HEWLETT PACKARD

Parallelization of Control Recurrences for ILP Processors

Michael Schlansker, Vinod Kathail, Sadun Anik
Compiler and Architecture Research
Computer Research Center
HPL-94-75
August, 1994

control dependences, recurrences, parallelism, control height reduction, back-substitution, blocked back-substitution, software pipeline, loop optimization

The performance of applications executing on processors with instruction level parallelism is often limited by control and data dependences. Performance bottlenecks caused by dependences can frequently be eliminated through transformations which reduce the height of critical paths through the program. While height reduction techniques are not always helpful, their utility can be demonstrated in an increasingly broad range of important situations.

This report focuses on the height reduction of control recurrences within loops with data dependent exits. Loops with data dependent exits are transformed so as to alleviate performance bottlenecks resulting from control dependences. A compilation approach to effect these transformations is described. The techniques presented in this report used in combination with prior work on reducing the height of data dependences provide a comprehensive approach to accelerating loops with conditional exits.

In many cases, loops with conditional exits provide a degree of parallelism traditionally associated with vectorization. Multiple iterations of a loop can be retired in a single cycle on a processor with adequate instruction level parallelism with no cost in code redundancy. In more difficult cases, height reduction requires redundant computation or may not be feasible.

1 Introduction

Control and data dependences are often a limiting factor in the performance of applications executing on processors with instruction level parallelism (ILP). Performance bottlenecks caused by dependences can frequently be eliminated through transformations which reduce the height of critical paths through the program. This report studies performance limitations imposed by control dependences present in a program and describes a collection of techniques which re-organize control dependences to reduce the height of critical paths. The objective is to retire sequences of dependent branches as quickly as possible, thus maximizing exploitable parallelism. Although the techniques are applicable to both loop and "scalar" code, this report focuses on control recurrences in loops with conditional exits.

The concept of control dependence has been defined in prior work [1]. Control dependences identify the relationship between each branch and the operations which depend upon its resolution. Control dependences correctly identify minimal conditions under which an operation may execute without speculation. Branches are a significant bottleneck in limiting ILP performance. A number of experimental measurements for ILP performance have been obtained [2-5]. In all cases, these measurements have not considered program transformations which alleviate bottlenecks caused by either data or control dependences.

Most of the prior work on height reduction relates to data height reduction, *i.e.*, height reduction of critical paths caused by data dependences. Early work in this area introduced techniques such as tree height reduction of arithmetic expressions [6]. More recent work demonstrated techniques, called symmetric and blocked back-substitution, for cyclic height reduction of arithmetic expressions in software pipelined loops [7]. A broad understanding of the utility of these techniques on ILP processors is not yet available.

There has been some prior work in alleviating the effects of control dependences. The use of speculative execution is one such technique. Speculative execution identifies operations whose side effects are reversible and moves these operations above branches on which they depend [8-12]. Speculative execution can significantly accelerate program performance but does not address the problem of parallel execution of branches and other non-speculative operations with problematic control dependences.

The ability to retire multiple branches in a single cycle can alleviate bottlenecks caused by chains of control dependences. The Multiflow Trace machine [13] is an example of a processor with such capability. While processors like the Trace can issue multiple branches per cycle, it is difficult to correctly guard non-speculative operations in between these branches. In general, it may be difficult to build hardware which retires multiple branches per cycle and executes operations between these branches under the correct branch condition. This problem is complicated by the need for a fast cycle time and short branch latency.

Predicated execution (see [14-17]) offers alternative approaches for removing performance bottlenecks due to control dependences. Predicated execution uses a boolean value to guard the execution of an operation. Predicated execution can be used to convert control dependences arising from a branch into data dependences between the computation of a predicate and conditionally executed operations which are guarded by this predicate. Control dependences, when converted to data dependences, are amenable to height reduction techniques such as symmetric and blocked back-substitution [7]. Predicate expressions as originally derived from programs may be too sequential in their construction. A set of properly defined machine

operations, *e.g.*, as in the PlayDoh architecture [18], can be used to parallelize the computation of these predicate expressions.

This report uses predicated execution to describe control height reduction. Similar control height reduction techniques can also be used to parallelize programs for architectures with no support for predicated execution. We use predicated execution as a means to describe control height reduction partly because it allows us to view both control and data dependences in an uniform manner. The conversion of control dependence to data dependece through use of predicates allows techniques, originally developed for the height reduction of data expressions, to be used for the height reduction of mixed data and control expressions.

The rest of the report is organized as follows. Section 2 introduces some of the basic concepts and notations that are used in this report. Section 3 illustrates the concept of control height reduction using an example. Section 4 identifies a class of loops, called *loops with separable stores*, that are amenable to control height reduction and describes the transformation process in detail. Section 5 illustrates the application of control height reduction to a class of conditional recurrences, called *write-overwrite recurrences*, that arise due to conditional assignments in a loop body. Section 6 describes the architectural support provided in the PlayDoh architecture for efficient evaluation of predicate expressions. Section 7 contains concluding remarks.

2 Overview of basic concepts and notations

This section describes some of the basic concepts and notations that are used throughout this report.

2.1 Predicated execution

Predicated execution has been implemented in the Cydra 5 computer and has been described in a number of papers [14-17]. Predicated execution refers to the conditional execution of an operation based on a boolean-valued operand, called a predicate. The operation executes if the predicate input is true and is nullified if it is false. Compare operations are used to calculate a boolean value which is subsequently used as a predicate. Predicated execution supports if-conversion which can eliminate program branches in conditional expressions. Predicated execution may also be used to generalize the rules of code motion across conditional branches.

We denote a predicated operation as follows: "r = op(a, b) if p". Here, p names the predicate operand which potentially nullifies the operation. Constants "T" and "F" are used to represent true and false boolean constants. The constant "T" may be used as the predicate operand for an operation. For example, "r = op(a, b) if T" executes unconditionally. Operations with no predicate specified also execute unconditionally and use the constant "T" for predicate. Thus, "r = op(a, b)" is the same operation as "r = op(a, b) if T". Predicates are complemented through the boolean complement operation "~".

At times we extend the notation to include predicated execution of a compound function. When we write "r = F(a, b, c) if p", we mean that all operations inside F execute using the predicate p or predicates derived from it (the function may contain embedded if-converted code).

2.2 Use of Expanded Virtual Registers and Dynamic Single Assignment form

Special notation is used to describe the flow of values between loop iterations. We use the concept of the *expanded virtual register* (EVR) as described by Rau [19]. Each EVR is a

linearly ordered set of virtual registers. An EVR t is referenced using the notation t[i] where i identifies the dynamic instance for the corresponding assignment. A special *remap* operation is defined on EVRs. Each time a remap(t) is executed for the EVR t, all values in the linearly ordered set are referenced through a new name. A value t[i] prior to the execution of a remap(t) is referenced as t[i+1] after the remap(t). A reference to the value t is defined to be exactly the same as a reference to t[0]. A value originally referenced as t, is referenced after the execution of a single remap(t) as t[1]. EVRs allow multiple values from a sequence of assignments to a common scalar to remain alive without terminating the lifetime of a value when the next member of the sequence is computed. This is used to eliminate anti and output dependences resulting from assignment to scalars within loops. EVRs can be used as an internal representation within a compiler and do not require specialized hardware support for register renaming.

EVRs are used to convert programs into *dynamic single assignment* (DSA) form. In DSA form, a value is written into a register only once. DSA form requires that for each program circuit, and for every write to some register r within the circuit, a remap(r) is traversed before revisiting the write to r. Anti-dependences and output dependences due to register assignment are eliminated, but flow dependences needed to complete the computation are preserved. The combination of EVRs and DSA form accurately describes dependences across zero, one, or more loop iterations.

2.3 Reduction operations

In the examples and the schemas presented in this report, we use certain high-level reduction operations in order to simplify the presentation. This section describes these reduction operations. Each reduction operation calculates a single result as a function of a variable number of inputs. Section 6 describes their implementation in terms of low-level operations in the PlayDoh architecture.

1. SELECT operation: result = SELECT(op1 if p1; op2 if p2; ... ; opn if pn). The value of the result is calculated by evaluating the operation (or function) corresponding to a true predicate. If none of the predicates, or more than one of the predicates is true, the result of a select is undefined. For the purposes of this report, we will assume that exactly one of the predicates in a SELECT statement is true, always giving a well defined result.
2. Logical AND operation: result = AND(e1, e2, ..., en). This operation represents a multiple input boolean conjunction of values computed by boolean expressions e1, ..., en.
3. Logical OR operation: result = OR(e1, e2, ..., en). This represents a multiple input boolean disjunction of values computed by boolean expressions e1, ..., en.

2.4 Function composition

In order to explain the height reduction schema, we need to describe components of a loop body in a functional form. As an example, the statement S = F(S[1]) means that the state variable S is calculated by applying the function F to the previous state variable S[1]. We use the notation $F^b(S[1])$ to represent the b-fold composition of F defined inductively as: $F^1=F$; and $F^b= F(F^{b-1})$ for $b > 1$.

3 An example to illustrate control height reduction

In this section, the concept of control height reduction is illustrated with an example. Figure 1 shows an example do-while loop in C. This loop performs the string copy operation while

counting the characters of the string. In this example, the strings pointed to by p and q are assumed not to overlap.

This loop exhibits several characteristics which are discussed in detail in Section 4. First the loop contains stores to memory. We assume that stores can not be speculatively executed. Thus, the loop contains a recurrence composed of the control dependence cycle containing the store operation and the conditional branch. The second property of the loop is that it contains a variable, count, which is live upon loop exit (to simplify the example, the variables p and q are assumed to be dead at loop exit). Finally, the number of iterations of the loop can not be calculated *a priori*, *i.e.*, the loop is not a counted loop.

```
count = -1 ;
do {
      *q++ = *p ;
      count++ ;
} while(*p++)
```

Figure 1: String copy with count

In its original form, this loop exhibits very limited ILP. Although the load operation can be executed speculatively, both the memory update operation and the calculation of values live at loop exit has to be executed under the correct control dependence. Figure 2 shows control and data flow dependences present in the loop which collectively form a number of potentially performance limiting recurrence cycles. In this figure, thick lines represent the data flow dependences and the thin lines represent the control flow dependences.

We can now analyze the dependences in this loop for the software-pipeline scheduling model [20, 21, 9, 22]. In software-pipelining, the control dependences for the increment operations for p and q, and for the load operation can be relaxed by speculative execution. On the other hand, the increment operation on count and the store to memory should be executed under the original control conditions.

Four dependence cycles that can effect the schedule length for software pipelining are shown in this figure. Three of these are data recurrences resulting from the increment operations on variables p, q, and count. These recurrences can be reduced by using data height reduction techniques like symmetric or blocked back-substitution [7]. The fourth recurrence in the loop is a control recurrence. The branch at the loop exit is control dependent on the branch from the previous iteration. This recurrence is the focus of this report. For an architecture where only one conditional branch can be executed at a time, this recurrence limits the software pipelined version of this loop to retire no more than one iteration within the latency of a single branch.

In the presence of architectural support for concurrent execution of multiple branches, this example presents another problem. When predicates cannot be used, control dependences constrain stores and assignments to live-out values to remain trapped between successive branches. On VLIW architectures supporting multiple branches per cycle it may be difficult to correctly guard operations trapped between concurrent branches. Control height reduction can be adapted to properly guard concurrent operations from adjacent iterations of a loop even without using predicated execution.

The control height reduction transformation consists of the following steps. First the loop is unrolled b times resulting in a loop where each iteration contains b iterations of the original loop.

We will call these iterations *major iterations* and *minor iterations*, respectively. The next step is preparation of the unrolled loop for control height reduction. The operations which can be executed speculatively are moved above the conditional branches and the operations which cannot be executed speculatively (*e.g.* stores and calculation of live-outs) are moved below the branches and also copied into the loop exit code. The predicates under which these operations execute are derived from the control dependences of the unrolled loop. Finally, the branches within the major iteration are collapsed resulting in the reduction of a chain of b control dependent branches into a single branch. After control height reduction, the loop traverses a single branch every b minor iterations reducing the effective latency imposed by control dependence by a factor of b. At this step, data recurrences which contain chains of data dependent calculations can be height reduced using existing techniques.

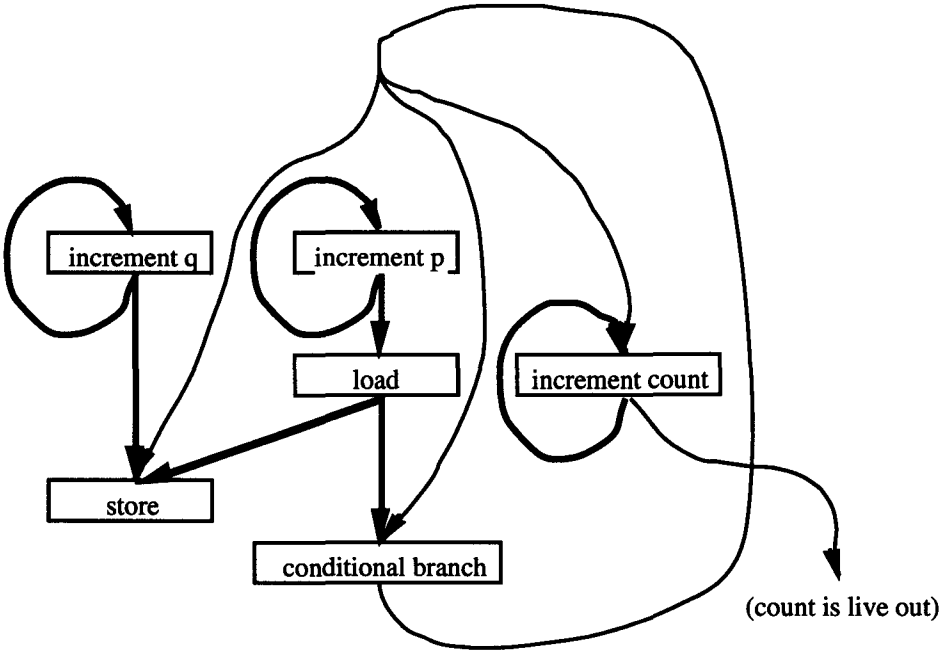

Figure 2: Dependence cycles in the example

Figure 3 presents the result of the combined data and control height reduction on the example in Figure 1. In this example, the degree of unroll used in this transformation is four. The original do-while loop is converted into a while loop. The loop kernel consists of three sections, speculative calculation of the loop exit condition, the loop branch for the loop exit which replaces four branches corresponding to the four minor iterations, and the stores to memory combined with the loop state update. The exit code for the loop contains the conditional stores to memory for the last four iterations of the original loop combined with the calculation of the final value of count. Within the loop kernel, data height reduction is applied to address calculation expressions to facilitate parallel execution of load and store operations.

3.1 Height reduction in intermediate representation

This section presents the treatment of the example loop in intermediate form. The intermediate form uses EVR notation and predicates to represent a program. Figure 4 shows the intermediate representation for the original do-while loop. The virtual registers vr1 and vr2 contain the address of the locations pointed by p and q; vr3 is the temporary register used to hold the data loaded from memory; vr4 holds the running value of count and cond is the variable used to store

the loop exit condition. The remap operation before the branch renames all the virtual registers as described in Section 2.2.

```
count = -1 ;
while(*p && *(p+1) && *(p+2) && *(p+3)) {
      *q = *p ;
      *(q+1) = *(p+1) ;
      *(q+2) = *(p+2) ;
      *(q+3) = *(p+3) ;
      p += 4 ;
      q += 4 ;
      count += 4 ;
}
*q = *p ;
count += 1 ;
if (*p) {
      *(q+1) = *(p+1) ;
      count += 1 ;
      if (*p+1) {
            *(q+2) = *(p+2) ;
            count += 1 ;
            if (*p+2) {
                  *(q+3) = *(p+3) ;
                  count += 1 ;
            }
      }
}
```

Figure 3: Height reduced string copy with count

```
      vr1[1] = p
      vr2[1] = q
      vr4[1] = -1    /* count */
loop: vr1 = vr1[1]+1
      vr2 = vr2[1]+1
      vr3 = load vr1[1]
      vr4 = vr4[1] + 1
      store vr2[1],vr3
      cond = (vr3 != 0)
      remap(vr1,vr2,vr3,vr4,cond)
      if cond[1] go to loop
exit: count = vr4[1]
```

Figure 4: Do while loop in intermediate representation

Figure 5 presents the intermediate code after applying the control height reduction transformation presented in Section 4. The reader is referred to Section 4 for the schema used in generating this code. The code is presented here for illustration purposes and to provide an example to the reader in the next section. This code corresponds to the height reduced state of the loop in Figure 4 after the application of the transformation in Figure 17 with a loop unroll factor of four. The example

contains two major components, the loop body and the loop exit code. The loop body contains four minor iterations where the computation in the fourth minor iteration is back substituted so that it is data dependent only on the fourth minor iteration from the previous major iteration. The address calculation of four load operations that set vr10-vr13 are also back-substituted.

```
      vr1[1] = p ; vr2[1] = q ; vr4[1] = -1 ; W[1] = TRUE
loop: vr1 = vr1[1]+1 ;              vr2 = vr2[1]+1
      vr3 = load vr1[1] ;           vr4 = vr4[1]+1
      remap(vr1,vr2,vr3,vr4,vr10,vr11,vr12,vr13,W)
      vr1 = vr1[1]+1 ;              vr2 = vr2[1]+1
      vr3 = load vr1[1] ;           vr4 = vr4[1]+1
      remap(vr1,vr2,vr3,vr4,vr10,vr11,vr12,vr13,W)
      vr1 = vr1[1]+1 ;              vr2 = vr2[1]+1
      vr3 = load vr1[1] ;           vr4 = vr4[1]+1
      remap(vr1,vr2,vr3,vr4,vr10,vr11,vr12,vr13,W)
      vr1 = vr1[4]+1+1+1+1 ;        vr2 = vr2[4]+1+1+1+1
      vr3 = load(vr1[4]+1+1+1) ;    vr4 = vr4[4]+1+1+1+1

      vr10 = load vr1[4] ;          vr11 = load (vr1[4]+1)
      vr12 = load (vr1[4]+2) ;      vr13 = load (vr1[4]+3)
      W = AND(W[4], (vr10 != 0), (vr11 != 0),
                    (vr12 != 0), (vr13 != 0) )
      store vr2[3],vr3[3]           if W
      store vr2[2],vr3[2]           if W
      store vr2[1],vr3[1]           if W
      store vr2,vr3                 if W
      remap(vr1,vr2,vr3,vr4,vr10,vr11,vr12,vr13,W)
      if W[1] goto loop
exit: W[4] = AND(W[5], (vr3[4] != 0) )
      W[3] = AND(W[4], (vr3[3] != 0) )
      W[2] = AND(W[3], (vr3[2] != 0) )
      store vr2[4],vr3[4]           if W[5]
      count = vr4[4]                if W[5]
      store vr2[3],vr3[3]           if W[4]
      count = vr4[3]                if W[4]
      store vr2[2],vr3[2]           if W[3]
      count = vr4[2]                if W[3]
      store vr2[1],vr3[1]           if W[2]
      count = vr4[1]                if W[2]
```

Figure 5: Intermediate code after transformation

In this figure two notations are adopted. The symbol ";" is used as a separator for multiple intermediate code statements that are on the same line. In our intermediate representation, load and store operations do not perform address calculation. However, to improve the readability of the example, address calculations whose only consumer is a single load or store operation may be folded into the arguments of load and store operations as has been done for back substituted addresses used in memory operations after the third remap in Figure 5. The loop in Figure 5

contains a significant amount of redundancy. Figure 6 shows the optimized version of this loop. In Figure 6, the remap operations have also been eliminated by renaming virtual registers.

```
      vr1 = p
      vr9 = q
      vr4 = -1
      W = TRUE
loop: vr2 = vr1 + 1   ; vr3 = vr1 + 2    ; vr4 = vr1 + 3
      vr5 = load vr1 ; vr6 = load vr2
      vr7 = load vr3 ; vr8 = load vr4
      W = AND(W, (vr5 != 0), (vr6 != 0),
                 (vr7 != 0), (vr8 != 0) )
      vr10 = vr9 + 1 ; vr11 = vr9 + 2   ; vr12 = vr9 + 3
      store vr9,  vr5                   if W
      store vr10, vr6                   if W
      store vr11, vr7                   if W
      store vr12, vr8                   if W
      vr1 = vr1 + 4                     if W
      vr9 = vr9 + 4                     if W
      vr4 = vr4 + 4                     if W
      if W  go to loop
exit: store vr9,  vr5                   if W
      count = vr4 + 1                   if W
      W = AND(W, (vr5 != 0) )
      store vr10, vr6                   if W
      count = vr4 + 2                   if W
      W = AND(W, (vr6 != 0) )
      store vr11, vr7                   if W
      count = vr4 + 3                   if W
      W = AND(W, (vr7 != 0) )
      store vr12, vr8                   if W
      count = vr4 + 4                   if W
```

Figure 6: Intermediate code after optimization

For an architecture which does not support predicated execution, reverse-if-conversion [23] can be used to obtain a branching version of this code. The branching intermediate code would correspond to the source code shown in Figure 3.

4 Framework for parallelization of loops with conditional exits

In this section, we formalize the transformations illustrated in the last section for the parallelization of control recurrences present in loops with conditional exits. As mentioned earlier, loops with conditional exits are inherently recurrences because control dependence between branches enforce the following rule: an iteration executes if the previous iteration has executed and did not exit. The approach described in this section uses predicated execution to convert the basic control recurrence threading through branches into a data recurrence. The control recurrence when converted to a data recurrence is amenable to height reduction techniques such as symmetric and blocked back-substitution described in [7]. As noted in

Section 3, the height reduction of control recurrence supports the execution of multiple loop iterations within a single cycle on ILP architectures where at most one branch can be processed in a single cycle.

4.1 Loops amenable to control height reduction

This section discusses the types of loops that are amenable to control height reduction and describes a canonical form for such loops, which will be used as the starting point for height reduction transformations. We consider do-while (also called repeat-until) loops with a single exit at the end of the loop. While loops can be converted to do-while loops by peeling off the first iteration test. Loops with multiple exits can also be converted into do-while loops with a single exit using the technique described in [9].

Figure 7 shows a do-while loop in a stylized form. We assume that conditional computations inside the loop body have been if-converted to predicated code. The loop state vector X represents all loop variant register state elements within the loop body. Functions F1, F2, etc. are applied to X in order to compute new values for some or all of the elements of the state vector X. These computations are interspersed with stores to memory. Functions A1, A2, etc., compute memory addresses for stores. Similarly, functions V1, V2, etc., compute the values to be stored in the memory. The function E calculates a boolean value (cond) to determine whether to continue the loop or not.

```
      X = LI(IN);              /* Initialize the state vector X */
Loop: X = F1(X);               /* Compute elements of X */
      Store(A1(X), V1(X)); /* Perform a store to memory */
      X = F2(X);
      Store(A2(X), V2(X));
      ...
      cond = E(X)            /* E(X) computes the branch condition */
      if cond go to Loop
Exit: OUT = LO(X);           /* Extract live-outs from X */
```

Figure 7: Original do-while loop in pseudo assembly code

Values that are live into the loop or live out of the loop must be carefully treated. IN and OUT represent scalar values which were live into or live out of the original source loop. In the stylized code shown in Figure 7, the function LI copies the live-in values from IN to the appropriate variables within the initial variant state. The function LO selects the live-out values from the final variant, which are then copied to OUT.

We define a class of loops, called *loops with separable stores*, that are amenable to control height reduction. Such a loop has the property that load operations used to resolve the loop-back branch, do not alias with store operations in previous iterations of the loop. Thus, the addresses of loads used to compute the condition E(X) in Figure 7 are guaranteed not to be the same as any of the store addresses A1(X), A2(X), ... in previous iterations. Separability can be defined over a finite number of previous iterations. The application of b-fold control back-substitution requires that stores move across all loads used to compute E(X) in the next b-1 iterations.

Another way to view this property can be seen within the dependence graph of the loop, *i.e.*, the graph that makes all control and data dependences explicit. The dependence graph of a loop with separable stores has the property that no recurrence cycle involving the branch has a store to

memory. The control height reduction technique described in this report relies on the ability to delay stores by moving them across one or more iterations. The separable store property guarantees that such code motions are permitted. Analysis techniques, such as vector dependence analysis, may be used. to prove that a loop is separable before control height reduction is performed. The loop may also have performance limiting load store dependences within the body. Techniques such as load and store elimination [19] may be used to eliminate these recurrences.

To illustrate the problem in accelerating loops that don't have the separable store property, consider the dependence graph shown in Figure 8. Although the branch in an iteration is control-dependent upon branches in previous iterations, the performance of the loop is limited by the recurrence cycle passing through all four operations. To accelerate this recurrence, it is necessary to move the load from an iteration to a previous iteration, or similarly, to delay the store by moving it across one or more iterations. Control and memory dependences in the figure prohibit such code motions.

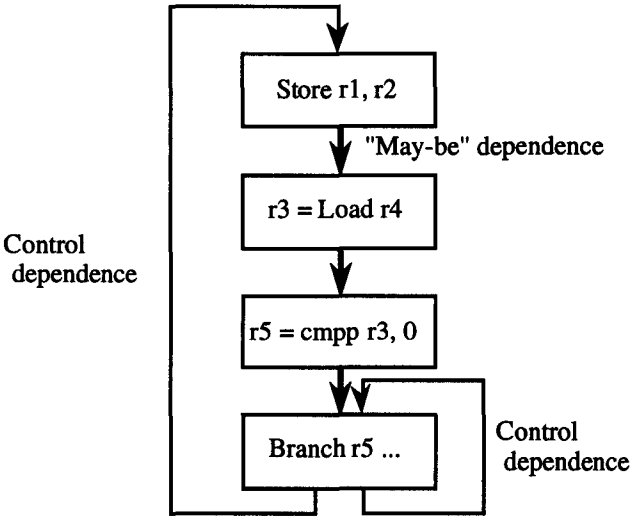

Figure 8: Precedence graph for a loop without separable store property

Implicit in the above discussion are certain assumptions about the target architecture. The first assumption is that the architecture permits all operations other than branches and stores to be issued speculatively. The second assumption is that the architecture provides no support to schedule a load before potentially aliasing stores that precede the load in the original program. If the target architecture permits speculative stores or if the architecture provides data speculative loads (*e.g.*, as in the PlayDoh architecture [18]), then the technique described in this report can be applied to loops without the separable store property; however, that is beyond the scope of this report.

Given a separable loop, the state vector X can be decomposed into two parts S and T such that the following holds. The state vector S contains all elements of X whose computation doesn't involve load operations that alias with the stores in the loop. The state vector T contains all other elements of X. The separable store property guarantees that both the computation of S and the computation of branch condition (cond) use only elements of S and don't use any elements of T. The computation of T may use elements of both S and T. Further, the computation may contain loads that alias with stores, in which case the load-store ordering implied in the source program

must be preserved. Similarly, stores may alias with each other and, if they do, they must be executed in their source order to preserve correct memory state.

With these definitions, a loop with separable stores can be put into a canonical form in which operations are separated into two groups. The first group contains operations that calculate the state vector S and the branch condition (cond). The second group contains operations related to the computation of T and all store operations. To separate the operations inside the loop into the two groups, it may be necessary to introduce new temporaries in the calculation of S in order to preserve certain intermediate values that are used in the calculation of the state vector T and the arguments to store operations. This can be done either by static renaming or by making use of EVRs.

Figure 9(a) shows the dependence graph of the canonical form of a loop with separable stores, and Figure 9(b) shows the functional representation used in later sections to explain height reduction. In Figure 9(b), T = U_T(S, T) represents all computation related to T as well as the effect of all stores to memory.

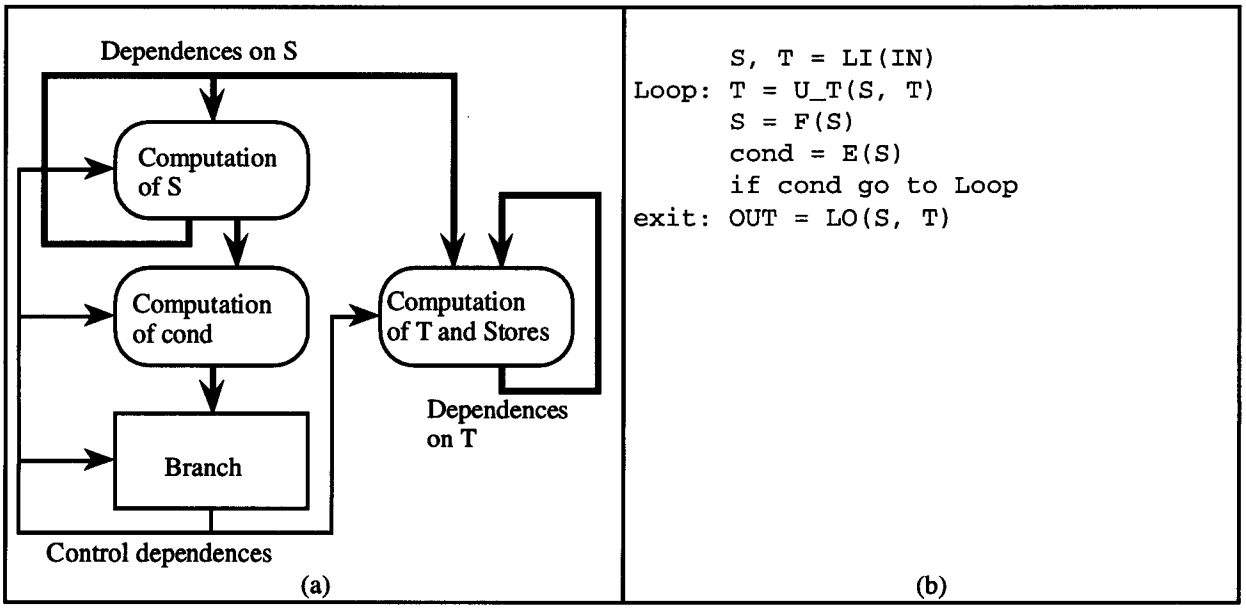

Figure 9: The canonical form for loops with separable stores. (a) dependence graph (b) textual form

The dependence graph in Figure 9(a) contains four types of recurrences.

1. Data recurrences that involve the state vector S. If necessary, data height reduction techniques such as symmetric and blocked back-substitution [7] can be applied to reduce the height of these recurrences.
2. Data recurrences that involve the state vector T. Data and control height reduction techniques are not applicable to these recurrences without special architectural support (*e.g.*, speculative stores, data speculative loads). In many important cases such as the example in Section 3, these recurrences don't exist.
3. Recurrences induced by control dependences between the branch and the computation of S. Results from S are in turn used to compute the branch condition. These recurrences can be eliminated by making these computations speculative.

4. The basic control recurrence induced by the control dependence from the branch in an iteration to the branch in the next iteration. That is, the branch is executed only if the branch in the previous iteration was executed and didn't exit the loop. The height reduction of this control recurrence is the focus of this report.

4.2 Converting to Dynamic Single Assignment form

To eliminate unnecessary anti- and output-dependences that arise because of repeated assignments to elements of state vectors S and T in each iteration of the loop, we convert the loop (see Figure 9(b)) into dynamic single assignment form. Figure 10 shows the result. The elements of S and T and the virtual register cond are now EVRs, which are remapped just before the branch.

```
      S[1], T[1] = LI(IN)
Loop: T = U_T(S[1], T[1])
      S = F(S[1])
      cond = E(S)
      remap(S, T, cond)
      if cond[1] go to Loop
exit: OUT = LO(S[1], T[1])
```

Figure 10: Dynamic single assignment form of the loop

4.3 Introducing the fully qualified predicate

As mentioned in Section 4.1, a branch in an iteration of a loop with conditional exits is control dependent upon the branch in the previous iteration. In this section, we introduce the notion of a *fully qualified predicate* and use it to convert this basic control recurrence into a data recurrence.

The fully qualified predicate associated with a branch takes into account not only the branch condition for the branch but also the branch conditions for all previous branches upon which the branch is control dependent. To calculate the sequence of fully qualified predicates W, we introduce a statement which performs a logical AND between W[1] from a previous iteration and the branch condition to calculate W for a current iteration (see Figure 11(a)). The predicate W is initialized to true before the loop.

All operations in an iteration that can't be executed speculatively (*e.g.*, stores) are also control dependent upon the branch in the previous iteration. Thus, to facilitate code motion across branches, we re-express the loop as a sequence of iterations whose operations are predicated with W[1], *i.e.*, the fully qualified predicate for the branch in the previous iteration. Note that the computation of W itself is not predicated, since the entire sequence of W must have well-defined values in order to properly guard the rest of the computation.

The branch condition for the branch at the end of the loop has been modified to W[1], which is the fully qualified predicate calculated in the current iteration (note the intervening remap operation). Thus like any other operation, the branch is predicated on the fully qualified predicate for the branch in the previous iteration except that the predication of a branch is expressed by modifying the branch condition. Each branch is no longer dependent on the branch from the previous iteration. Instead, the AND operation which calculates the branch condition is dependent upon the AND operation from the previous iteration. This permits branches to be

moved across each other and permits multiple branches to be scheduled in the same cycle on architectures that provide such a capability (*e.g.*, PlayDoh). However, we don't rely on such architectural capability in this report.

```
      S[1], T[1] = LI(IN)
      W[1] = TRUE
Loop: T = U_T(S[1], T[1])  if W[1]
      S = F(S[1])          if W[1]
      cond = E(S)          if W[1]
      W = AND(W[1], cond)
      remap(S, T, W, cond)
      if W[1] go to Loop
exit: OUT = LO(S[1], T[1])
```

(a)

```
      S[1], T[1] = LI(IN)
      W[1] = TRUE
Loop: T = U_T(S[1], T[1])  if W[1]
      S = F(S[1])
      W = AND(W[1], E(S))
      remap(S, T, W)
      if W[1] go to Loop
exit: OUT = LO(S[1], T[1])
```

(b)

Figure 11: (a) Loop with all operations guarded using fully qualified predicates (b) Loop with speculative computation of S and E(S), *i.e.* , cond.

As pointed out in Section 4.1, recurrences induced by control dependences between a branch and the computation of S and the branch condition may be the limiting factor in performance. Thus, we make the computation of S and cond speculative by changing their guarding predicate to TRUE. This allows their computation to be freely moved to previous iterations. The computation abbreviated by U_T, however, must be correctly guarded. Specifically, all stores must be executed under the correct control conditions. Figure 11(b) shows the code after this transformation. We have also eliminated explicit assignment to cond to simplify the presentation in later sections.

In the most optimistic scenario when there are no data recurrences other than the one that expresses W in terms of W[1], the loop execution takes n cycles for each iteration where n is the latency of the AND operation.

We can now decompose the problem of accelerating the loop into two disjoint problems. The first problem is to accelerate the control recurrence. The control recurrence when converted to a data recurrence (threading through W) is amenable to height reduction techniques such as symmetric and blocked back-substitution described in [7]. Both these techniques replace the loop body with an equivalent loop having substantially reduced critical path and may introduce redundant computation to achieve the height reduction. We discuss these techniques in the subsequent sections.

The second problem relates to actually exiting the loop. The loop shown in Figure 11 has the property that if the loop were to run for a finite number of additional iterations after the loop terminates, the semantics of the program will still be preserved–all operations in the body would execute using predicate FALSE and would have no effect on the program state. Thus, the branch out of loop need not be executed precisely on time. If the branch out of loop is delayed, a correct result is ensured through the action of the predicates. This permits a loop to be unrolled an arbitrary amount with only one branch at the end of the loop. This is particularly important when executing multiple loop iterations in a single cycle. In such cases, it may be difficult to exit between adjacent iterations of the original loop.

4.4 Symmetric back-substitution of loops with conditional exits

As mentioned earlier, we need to height reduce the data recurrence which threads through the variable W in order to accelerate the control recurrence. A symmetric height reduction as described in [7] would calculate the predicate W for every iteration using an identical expression appearing in a non-unrolled loop body. We could use symmetric height reduction in order to reduce the height of this recurrence and accelerate the loop. We will not discuss this technique in this report because it introduces more redundant computation than blocked back-substitution discussed in the next section.

4.5 Blocked back-substitution of loops with conditional exits

Blocked back-substitution begins by unrolling a loop (*e.g.*, b iterations). The last of the b unrolled iterations is height-reduced through substitution and expression optimization. It has been shown that this asymmetric form of the code is particularly efficient for the evaluation of recurrences (see [7]).

The starting point to explain the transformation process is the loop shown in Figure 11(b). To describe how live-outs are computed correctly, we temporarily move the assignment to OUT into the body of the loop and properly guard the assignment (see Figure 12). Note that the assignment is moved across a remap, and thus, S[1] and T[1] has been adjusted to S and T, respectively. The motion of the live-out computation inside the loop introduces scheduling constraints that were not present in the original loop. Since the loop repeatedly assigns to the same set of variables, *i.e.*, OUT, in each iteration, the order of the assignments must be preserved to get the correct live-out values. This ordering constraint appears as an output-dependence from the assignment in one iteration to the assignment in the next iteration. In the subsequent discussion, we ignore this output-dependence, since the purpose of moving the assignment inside the loop is to explain the transformation process. In the final transformed code, the computation of live-outs will be done outside the loop.

```
      S[1], T[1] = LI(IN)
      W[1] = TRUE
Loop: T = U_T(S[1], T[1])    if W[1]
      S = F(S[1])
      W = AND(W[1], E(S))
      OUT = LO(S, T)         if W[1]
      remap(S, T, W)
      if W[1] go to Loop
exit:
```

Figure 12: Loop with the computation of live-outs

4.5.1 Unrolling and back-substitution

The first step is to unroll the loop a number of times. The degree of loop unroll depends upon the amount of parallelism in the target machine. The ideal unroll factor is one that exposes as much parallelism as is available on the target machine. Figure 13 shows the loop unrolled b times.

The second step is to reduce the height of the control recurrence (*i.e.*, recurrence threading through W) using back-substitution and expression optimization. Although control height reduction is the focus of this report, a combined approach that addresses both control and data recurrences is usually necessary to expose parallelism. To reinforce this fact, the schema

presented in this section shows height reduction of not only the control recurrence but also data recurrences threading through the state vector S.

```
      S[1], T[1] = LI(IN)
      W[1] = TRUE
Loop:  /* First minor iteration */
      T = U_T(S[1], T[1])     if W[1]
      S = F(S[1])
      W = AND(W[1], E(S))
      OUT = LO(S, T)          if W[1]
      remap(S, T, W)
      if ~W[1] go to exit
      ...
/* bth minor iteration */
      T = U_T(S[1], T[1])     if W[1]
      S = F(S[1])
      W = AND(W[1], E(S))
      OUT = LO(S, T)          if W[1]
      remap(S, T, W)
      if W[1] go to Loop
exit:
```

Figure 13: b-way unrolled loop

The height reduction is accomplished by expressing the evaluation of both W and S in the last minor iteration directly in terms of variables W[b] and S[b]. Variables W[b] and S[b]. are produced in the b^{th} previous iteration, *i.e.*, the values in the last minor iteration of the previous major iteration (see Figure 14). Furthermore, the expressions calculating functions F, ..., F^b are simplified and height-reduced to enhance speedup. In the example illustrated in Figure 3, the evaluation of the F^b function has been simplified. This can be seen where the constant 4 is added to variables p, q, and count. Constants have been folded to allow every four iterations to traverse only a single irredundant summation on the recurrence path.

The third step removes the branches in the first b - 1 iterations. Note that all stores as well as the computation of live-outs are properly guarded using fully qualified predicates. Thus, the intermediate branches in the unrolled loop can be pushed to the bottom of the loop without affecting the semantics after which they can be eliminated, since their effect is subsumed by the loop-back branch. Thus, the back-substituted loop executes only a single branch every major iteration. Control dependences among b minor iterations are enforced using predicates. As mentioned in Section 4.3, the use of predicates permits execution of superfluous minor iteration within the last major iteration without changing the semantics. The stores and assignments to OUT in these extra minor iterations are simply nullified, and the final live-out values and the final memory state are correctly determined independent of the loop trip count. Figure 14 shows the code after back-substitution and elimination of branches.

Bounds on the interval between loop iterations can be derived due to resource limitations (ResMII) and recurrence path length limitations (RecMII) as described in [22]. One can calculate ResMII and RecMII bounds for the loop before and after the block back-substitution transformation. Ignoring data dependences, the control recurrence in the original loop implies

that each iteration takes n cycles where n is the latency of the AND operation. After back-substitution W is calculated in terms of W[b] again in n cycles, but the height reduced recurrence spans b iterations of the original loop providing a b-fold height reduction.

```
      S[1], T[1] = LI(IN)
      W[1] = TRUE
Loop:  /* First minor iteration */
      T = U_T(S[1], T[1])     if W[1]
      S = F(S[1])
      W = AND(W[1], E(S))
      OUT = LO(S, T)          if W[1]
      remap(S, T, W)
      ...
/* Form for back-substituted last minor iteration */
      T = U_T(S[1], T[1])     if W[1]
      S = F^b(S[b])
      W = AND(W[b], E(F(S[b])), ..., E(F^b(S[b])))
      OUT = LO(S, T)          if W[1]
      remap(S, T, W)
      if W[1] go to Loop
exit:
```

Figure 14: Loop after back-substitution of S and W and removal of intermediate branches

Several additional points should be noted. The first point relates to the implementation of AND reduction. As long as the path from W[b] to W involves only a single boolean operation, exactly how the AND reduction is implemented doesn't affect the achievable II (initiation interval) for the loop. When using unconditional boolean operations to evaluate W, the associative property can be used to reorganize the evaluation of W so that only a single operation is on the critical path. A concurrent implementation of AND can improve the performance of short trip count loops, by reducing the schedule length of a single iteration and hence the epilog stage count (see [24]) of the software-pipelined loop. Section 6 describes micro-architecture support for calculating AND reductions very efficiently.

Second, the performance of the back-substituted loop, in general, depends on the form of the function F. When F^b can be quickly and efficiently evaluated, speedups are substantial. In many cases, F^b can be evaluated at a cost similar to that of evaluating F and an unlimited amount of parallelism can be exposed with no penalty for redundant operations. This is true, for example, when an address strides by a loop invariant quantity (as shown in the example of Section 3). The b-fold address update corresponds to adding a loop invariant.

4.5.2 Removal of partial block code from loop body

The loop shown in Figure 14 contains excessive code which is required to treat the last major iteration in the loop, which may be executed only partially. In this section, we describe how this code can be removed from the loop body.

To gain an intuitive understanding of the transformation described in this section, consider the following two cases. First, consider the execution of a major iteration that does not exit the loop.

Since all minor iterations execute, it is not necessary to guard the computation in each minor iteration by a distinct predicate. Instead, the computations inside each minor iteration can be guarded by the predicate that captures the fact that the major iteration doesn't exit. This will allow the computation of b-1 predicates to be eliminated from the loop body. Also, the computation of live-out in this major iteration is unnecessary, since it is only the live-out computation in the last major iteration that is visible outside the loop.

Consider the last major iteration. The loop body must specify an exit condition which falls through after identifying a last major iteration. After the branch condition for the last major iteration is computed, all remaining computation from the last major iteration can be moved outside the loop. The transformations described in this section simplify the loop body by performing the conditional execution of minor iterations within the last major iteration outside of the loop.

The first step is to move the execution of U_T as well as the evaluation of the live-out function LO to the last minor iteration. Figure 15 shows the code after this step.

```
      S[1], T[1] = LI(IN)
      W[1] = TRUE
Loop: /* First minor iteration */
      S = F(S[1])
      W = AND(W[1], E(S))
      remap(S, T, W)
      ...
/* Back-substituted last minor iteration */
      S = F^b(S[b])
      W = AND(W[b], E(F(S[b])), ..., E(F^b(S[b])))
/* Computation of T, OUT and stores */
      T[b-1] = U_T(S[b], T[b])        if W[b]
      OUT = LO(S[b-1], T[b-1])        if W[b]
      ...
      T = U_T(S[1], T[1])             if W[1]
      OUT = LO(S, T)                  if W[1]

      remap(S, T, W)
      if W[1] go to Loop
exit:
```

Figure 15: Code with the computation of T, OUT and all stores moved at the end of the loop

The next step is to apply a transformation, which we call *predicate splitting*. This transformation replaces a computation guarded by predicate p by multiple copies of the computation guarded by predicates q1, ..., qn such that $p = q1 \vee ... \vee qn$. That is , the effect of the multiple copies of the computation under q1, ..., qn is the same as the effect of the original computation under p. This transformation in predicate domain is analogous to moving a computation below a branch or above a merge (join) in the control-flow domain.

To see how the predicate splitting transformation can be applied to simplify the loop, consider one of the statements of the form T [i-1] = U_T(S[i], T[i]) if W[i] ($1 \leq i \leq b$). We can replace this statement by the following two statements:

```
T[i-1] = U_T(S[i], T[i])    if W ∧W[i]
T[i-1] = U_T(S[i], T[i])    if ~W ∧W[i]
```

The assignments to T have all been collected into the last minor iteration. Here, W is the newly computed predicate in the last minor iteration. Since W is the conjunction of the predicates for all the previous iterations, W evaluates to true implies that each of W[b], ..., W[1] must also be true. Thus, the predicate expression guarding the first statement can be simplified to W, which simply states that the current major iteration doesn't exit the loop. In the predicate expression guarding the second statement, ~W states that the current major iteration is the last one. The other part of the conjunction, *i.e.*, W[i], describes if $(b\text{-}i)^{th}$ minor iteration corresponds to an executed iteration in the original loop or not. Since the second statement executes only in the last major iteration, it can be moved out of the loop. We apply the predicate splitting transformation to the evaluation of U_Ts and to the computation of the live-outs (LO). After the live-out computation for the last major iteration is moved out of the loop, the live-out computation remaining inside the loop is unnecessary and can be removed. Figure 16 shows the simplified code.

```
      S[1], T[1] = LI(IN)
      W[1] = TRUE
Loop: /* First minor iteration */
      S = F(S[1])
      W = AND(W[1], E(S))
      remap(S, T, W)
       ...
/* Back-substituted last minor iteration */
      S = F^b(S[b])
      W = AND(W[b], E(F(S[b])), ..., E(F^b(S[b])))
/* Computation of T and stores */
      T[b-1] = U_T(S[b], T[b])      if W
      ...
      T = U_T(S[1], T[1])           if W

      remap(S, T, W)
      if W[1] go to Loop

exit:  /* Handle live-outs and U_Ts in the last major iteration */
      T[b] = U_T(S[b+1], T[b+1])    if W[b+1]
      OUT = LO(S[b], T[b])          if W[b+1]
      ...
      T[1] = U_T(S[2], T[2])        if W[2]
      OUT = LO(S[1], T[1])          if W[2]
```

Figure 16: Removal of partial block code from loop body

The loop in Figure 16 can be further simplified. First, only every b^{th} member of the sequence of values for W are used within the body of the loop. We call this the sequence of block guards or major iteration guards. Each newly computed block guard W is true if the previous block guard

is true and none of the intervening minor-iterations required a loop exit. The other b-1 members of the sequences are used only after loop exit and can be evaluated out of loop.

Second, only a subset of the state vector sequence S is needed within the body of the loop. Some components of these state vectors are needed in the computation of T and in the arguments to stores. Computation of these components must remain inside the loop. However, some of the components may be needed only in the computation of live-outs, and their computation can be moved out of the loop. To accomplish this, we simply replicate the computation of the state vector S in the first (b - 1) minor iterations in the exit code, and rely on traditional optimizations to simplify the code both within the loop and within the exit code. Figure 17 shows the code after these transformations.

```
      S[1], T[1] = LI(IN)
      W[1] = TRUE
Loop: /* First minor iteration */
      S = F(S[1])
      remap(S, T, W)
      ...
/* Back-substituted last minor iteration */
      S = F^b(S[b])
      W = AND(W[b], E(F(S[b])), ..., E(F^b(S[b])))
/* Computation of T and stores */
      T[b-1] = U_T(S[b], T[b])        if W
      ...
      T = U_T(S[1], T[1])             if W

      remap(S, T, W)
      if W[1] go to Loop

exit:  /* Computation of W and replicated computation of S */
      S[b] = F(S[b+1])
      W[b] = AND(W[b+1], E(S[b]))
      ...
      s[2] = F(S[3])
      W[2] = AND (W[3], E(S[2])
/* Handle live-outs and U_Ts in the last major iteration */
      T[b] = U_T(S[b+1], T[b+1])      if W[b+1]
      OUT = LO(S[b], T[b])            if W[b+1]
      ...
      T[2] = U_T(S[3], T[3])          if W[3]
      OUT = LO(S[2], T[2])            if W[3]
      T[1] = U_T(S[2], T[2])          if W[2]
      OUT = LO(S[1], T[1])            if W[2]
```

Figure 17: Moving intermediate computations of W out of the loop and preparing for traditional optimizations

Note that the final code shown in Figure 17 needs to be simplified through traditional optimizations such as common sub-expression, dead code elimination, constant folding, etc., in order to remove unnecessary and redundant code. These optimizations must, however, be extended to operate correctly in the presence of remap and predication.

To summarize this section, we have used predicated execution to convert control dependences into data dependences and applied blocked back-substitution to reduce the height of the control recurrence (as well as data recurrences). The resultant code is optimized to produce the height reduced and simplified result.

The blocked back-substitution method for control height reduction can reveal unlimited parallelism in loops with conditional exits. However, this parallelism may come at the expense of some redundancy. The amount of redundancy depends upon the exact nature of the loop body. In many important cases such as the example in Section 3, the height reduced code is essentially irredundant. With proper attention to optimization, the back-substituted code with b-way unroll contains b times as many operations as in the original code, yet the recurrence height is divided by b. In fact, blocked back-substitution may reduce the number of operations executed per iteration in some cases. The reason is that some of the code to compute live-out values has been moved out of the loop. The example presented in Section 3 illustrates such a case. In general, the special treatment of the final major iteration, *i.e.*, moving some of the code out of the loop, may lead to some performance penalty on loops with very short trip count.

5 Height reduction of conditional recurrences

The blocked back-substitution technique for height reduction is applicable not only in the case of the basic control recurrence in a loop with conditional exits but also to a more general class of recurrences induced by control dependences. In this section, we apply the technique to a class of conditional recurrences, which we call *write-overwrite recurrences*. We will use the "first minimum" loop taken from the Livermore FORTRAN Kernels (loop 24) [25] as an example to illustrate the height reduction of such recurrences.

```
m = 1
do 24 k = 2, n
      if (X(k) .LT. X(m)) m = k
24 continue
```

(a)

```
m = 1
xm = X(1)
do 24 k = 2, n
      t = X(k)
      if(t .LT. xm) {
            m = k
            xm = t
      }
24 continue
```

(b)

Figure 18: (a) Original code for "write-overwrite" recurrence (b) Code after load elimination

```
m[1] = 1
k[1] = 2
xm[1] = load X(1)
do i = 1, n - 1
      t = load X(k[1])
      p = (t < xm[1])
      m = SELECT( k[1] if p; m[1] if ~p )
      xm = SELECT( t if p; xm[1] if ~p )
      k = k[1] + 1;
      remap(m,xm,p,t,k)
endo
```

Figure 19: Low level if-converted code

Figure 18(a) shows the original code for the loop. The purpose of the loop is to identify the index of the smallest element of an array X. The loop contains a recurrence because of the conditional assignment to m in an iteration and the use of m in computing the branch condition in the next iteration. The recurrence is an example of a write-overwrite recurrence, so called because it involves a conditional assignment. We can height reduce such a recurrence by unrolling the loop and re-associating the conditional assignment to m.

While it may appear that two load operations are necessary per loop iteration, we can eliminate the load from "X(m)" (see, for example, [19]). This can be done because X(m) always re-loads a minimal X(k) which was loaded in an earlier iteration. Figure 18(b) shows the code after load elimination. The re-written program contains a new variable xm which retains the current value of X(m). The variable xm is updated with value X(k) every time m is updated with k.

We apply if-conversion to the loop body in order to remove the branch and express control dependences as data dependences. This permits us to better understand the underlying recurrence which limits its performance and to apply the blocked back-substitution technique to reduce its height. Figure 19 shows the low level code after if-conversion and after conversion to dynamic single assignment form. Conditional assignments to m and xm are implemented using SELECT operation. In addition, the loop count has been normalized to start at 1.

We now consider the maximum sustainable rate if we were to software-pipeline the loop. Ignoring resource constraints, this limit is determined by the RecMII for the loop. Careful inspection reveals that the critical path traverses the compare operation that calculates predicate p, which is in turn used by a register-to-register copy within the SELECT operation to compute xm. The process repeats every loop iteration resulting in a RecMII equal to the sum of the compare and the copy latencies.

To reduce the height of the recurrence, we unroll the loop b times and re-associate SELECT operations. The re-association of SELECT operations relies on the transitive and anti-symmetric properties of the less-than (<) operator to reorder the sequence of comparisons. The basic idea is this. First, we find the first minimum within a block of minor iterations which form a single major iteration. The first minor iteration within a major iteration makes no reference to the minimum calculated in the previous major iteration. That is, the first minor iteration unconditionally assumes that the value for that iteration is the minimum. Each of b-1 subsequent minor iteration computes the minimum taking into account the minimum computed only in minor iterations from the current major iteration. Then, we select a minimum across all major iterations. A final code sequence within a major iteration uses the minimum value from all previous major iterations and the minimum value from the current major iteration to compute the minimum value for the next major iteration. Figure 20 shows the code after height reduction. To unroll the loop, we have "post-conditioned" the loop. The blocked loop executes an integral multiple of b source iterations. The remaining iterations are done by a second loop after the blocked loop. The loop bounds for the two loops, Q and R, are given by the following formulas.

$$R = (n-1) \bmod b$$
$$Q = (n-1) - R$$

The RecMII for the transformed loop is the sum of a compare latency and a copy latency. However, the transformed loop advances across b minor iterations. Thus, the height of the recurrence has been amortized across b iterations and effectively reduced b-fold. As we can see, this recurrence can be parallelized as desired without significant redundancy in the calculation by increasing the degree of back-substitution. This acceleration technique can be generalized to an

important class of computations where a variable is repeatedly overwritten and the overwrite test obeys the properties of an ordering relation.

```
m[1] = 1
k[1] = 2
xm[1] = load X(1)

do i = 1, Q, b
/*special first iteration */
      t = load X(k[1])
      p = (t < xm[1])
      m = k[1]
      xm = t
      k = k[1]+1
      remap(m,xm,p,t,k)

/*b-1 conventional iterations */
      t = load X(k[1])
      p = (t < xm[1])
      m = SELECT( k[1] if p; m[1] if ~p )
      xm = SELECT( t if p; xm[1] if ~p )
      k = k[1]+1;
      remap(m,xm,p,t,k)
      ...
/* special code sequence to compute minimum for the next
   major iteration */
      p = (xm[b] < xm[1])
      m = SELECT( m[b] if p; m[1] if ~p)
      xm = SELECT( xm[b] if p; xm[1] if ~p )
      remap(m,xm,p,t,k)
endo

/* Remaining iterations */
do i = 1, R
      t = load X(k[1])
      p = (t < xm[1])
      m = SELECT( k[1] if p; m[1] if ~p )
      xm = SELECT( t if p; xm[1] if ~p )
      k = k[1]+1;
      remap(m,xm,p,t,k)
enddo
```

Figure 20: Code after applying height reduction transformation

In previous sections, we have discussed high-level issues surrounding the transformation of an important class of loops with control recurrences. We have identified techniques which enhance available parallelism using control height reduction. In the following section, we describe low-level issues surrounding control height reduction. We illustrate how low-level operations such as AND, OR, and SELECT can be accelerated using appropriate architectural primitives.

6 Acceleration of predicate computation

The use of predicates allows us to convert control dependences into data dependences. The PlayDoh architecture [18] provides highly specialized semantics for efficient calculation of predicate expressions. This is discussed in the first two subsections. The rest of this section describes parallel implementation of the reduction operations introduced in Section 2 using the low-level operations in the PlayDoh architecture.

6.1 Semantics of simultaneous writes to a register

The PlayDoh architecture provides unusual semantics for simultaneous writes to registers. Unlike traditional architectures, multiple operations may write into a register in a cycle provided they all write the *same value*. In this case, the result stored in the register is simply the value being written. On the other hand, if multiple operations attempt to write different values into a register simultaneously, then the result stored in the register is undefined. In the case of writes to predicate registers, this atypical semantics is useful for efficient evaluation of boolean reductions discussed later in this section. Note that a predicated operation is conditionally executed and doesn't write into its destination register(s) if the guarding predicate is false.

6.2 Compare operations to calculate predicates

The calculation of predicates is performed through a family of compare operations. PlayDoh provides compare operations that target two predicate registers. The two results of a compare are typically used to guard operations under taken and not taken branch conditions from a single compare. To simplify the presentation, however, we describe compare operations as operations with a single destination. Each compare operation computes the value of a predicate as a function of another predicate and a compare condition. This is described, for example, using a statement of the form:

p_out = cmpp.<comp_cond>.<D-action>.(i1, i2) if p_in.

This statement computes the predicate p_out in terms of input boolean predicate p_in and input data arguments i1 and i2. The compare op-code "cmpp" has two associated modifiers: the compare condition <comp_cond> and the destination action modifier <D_action>. The compare condition serves to enumerate classical compare conditions such as compare for equality, inequality, less than, etc. The choice of compare conditions for PlayDoh mirrors HP PA-RISC. We introduce compare conditions informally as needed by examples.

The destination action modifier uniquely specifies the means by which the input predicate and the compare condition are combined to produce a final predicate result. To understand the destination action specifiers, consider each combination of the predicate value and the boolean result of the compare,. For each combination, there are three possible choices as to what can be done with a destination. The choices are as follows:

1. Write 0 into the destination register.
2. Write 1 into the destination register.
3. Leave the destination unchanged.

That is, there are three possible actions for each of the four combinations of the predicate input and compare result. A total of $3^4 = 81$ possible actions that can be performed on a destination.

Out of these, PlayDoh supports the ones described in Table 1. That is, the destination action may take on one of eight settings:

<D-action> = UN | CN | ON | AN | UC | CC | OC | AC

each of which corresponds to a single column in Table 1. These are sufficient to cover most requirements on predicate use. A brief explanation is given below for each of these eight destination actions;

Table 1: Destination action specifiers for compare-to-predicate operations and their semantics. An entry with -- means leave the target unchanged.

Predicate input	Result of comparison	On result				On the complement of result			
		UN	CN	ON	AN	UC	CC	OC	AC
0	0	0	--	--	--	0	--	--	--
0	1	0	--	--	--	0	--	--	--
1	0	0	0	--	0	1	1	1	--
1	1	1	1	1	--	0	0	--	0

In the subsequent discussion, we use the following terms to describe compare operations. The *unconditional* class refers to operations with UN and UC modifiers which always write the target predicate. The *conditional* class refers to operations with CN and CC modifiers which write the target predicate if the predicate input is true. The *OR* class refers to operations with ON and OC modifiers as they are used in OR reductions, and the *AND* class refers to the ones with AN and AC modifiers as they are useful in AND reductions .

First, we discuss the four actions grouped under the heading "on result". Unconditional operations (UN) always write into the destination register. If the predicate input is false, they clear the destination register; otherwise, they copy the result of the comparison into the destination register. In other words, these operations effectively compute the boolean conjunction of the input predicate and the result of the comparison.

Conditional operations (CN) behave like predicated compares. That is, if the predicate input is false, they leave the destination unchanged; otherwise they copy the result of the comparison into the destination. Note that both conditional and unconditional operations display identical functionality if the predicate input is true and differ only in the case when the predicate input is false.

The other two classes (OR and AND) are useful in efficient evaluation of boolean reductions (see sections 6.4 and 6.5). Operations in the OR class (ON) write a 1 into the destination register only if both the predicate input and the result of the comparison are true. Otherwise, they leave the destination unchanged. Operations in the AND class (AN) write a 0 into the destination register when the predicate input is true and the result of the comparison is false. Otherwise, they leave the destination unchanged.

The actions marked "on the complement of the result" are similar to the ones described above except that they implicitly complement the result of the comparison. For example, consider UN and UC modifiers. Operations with UN modifier write the result of the comparison into the destination register if the predicate is true, whereas operations with UC modifier write the complement of the result.

A point to note is that the semantics of compare-to-predicate operations is somewhat unique with respect to the conditions under which writes to the destination register are nullified.

Unconditional compare operations always write into their destinations even if the predicate input is false, *i.e.*, they use the predicate input like a regular data input. Conditional compare operations have semantics exactly similar to predicated data operations. Operations in the OR class and the AND class also behave like conventional predicated operations; that is, they perform no action if their predicate input is false. However, their operation semantics is unusual in that these operations conditionally update their destination even when the predicate is true as is illustrated in Figure 22.

6.3 Implementation of SELECT

The PlayDoh provides direct support for implementing a SELECT operation (see Section 2) in terms of predicated copy operations. Figure 21 shows the implementation of a SELECT operation.

```
r = SELECT(op1 if p1, ..., opn if pn) ≡   r = op1 if p1
                                          ...
                                          r = opn if pn
```

Figure 21: Implementation of SELECT

The result of SELECT is well-defined only when exactly one of the predicates is true. Then according to the semantics of predicated execution, only one of the operations in the low-level sequence writes into r and the result stored in r is the value computed by the operation. In all other cases, the result of SELECT is undefined and it is immaterial what value is written into r by the low-level code sequence. Note that predicated copies used to implement a SELECT operation can be executed concurrently or in any order. Although they all target the same destination register, there are no output dependences between them.

6.4 Implementation of AND

PlayDoh architecture provides the means for a very efficient implementation of the AND reduction operation. The two features that are used are the AND class of comparison operations and the PlayDoh semantics for the simultaneous writes to a destination register.

```
p = CMPP.W..UN(1, 1) /* Initialize p to 1 */
...
p = CMPP.W.<.AN(a, b)
p = CMPP.W.>.AN(c, d)
p = CMPP.W.<.AN(a, c)
```

Figure 22: Low-level code illustrating the implementation of AND reduction

The AND reduction operation can be implemented by first setting the destination register to 1, and then executing one compare operation per argument of the AND reduction with AN (or AC) as the destination modifier. When an argument evaluates to 1 then the corresponding compare operation takes no action. When an argument evaluates to 0, the compare operation sets the destination predicate to 0 forcing the result of the AND reduction to be 0. As an example, we illustrate the evaluation of the following AND-reduction. In the expression, p is a predicate register and a, b, c, d are general purpose registers.

```
p = AND( a < b, c > d, a < c)
```

The code for evaluating the reduction is given in Figure 22. The first statement in the code is a way to set the register p to 1. In the statement, 0 refers to the integer literal 0.

The semantics of simultaneous writes to a register permits these operations to be issued concurrently or in any order. The AND class of compare operations either leave the destination register unchanged or write 0 to it. Therefore whenever multiple AND class operations which have the same destination register execute concurrently, the value in the destination register is always well-defined. In the case where none of the operations modify the destination, the register contains its value prior to the execution. If at least one of the AND class compare operations modify the destination, the value is 0. Thus, the AND reduction of the results of an arbitrary number of compare operations can be done in the same amount of time as the latency of a single AND provided there is enough parallelism in the machine. Terms participating in an AND reduction are not output dependent on each other and can be executed in any order, in spite of the fact that they target a common register. The AND operation is used directly in the height reduction transformation.

6.5 Implementation of OR

The OR reduction can be implemented efficiently like the AND reduction. In this case, the destination predicate is initialized to 0, and the <D-action> modifier for the compare operations is ON or OC. When an argument evaluates to 0, then the corresponding compare operation takes no action. When an argument evaluates to 1, the compare operation sets the destination predicate to 1 forcing the result of the OR reduction to be 1. The OR operation is used in the if-conversion of conditionals [26] which has been performed prior to height reduction.

7 Conclusions

This work has demonstrated that loops with conditional exits provide substantial ILP when accelerated using height reduction. On processors with adequate parallelism, multiple iterations of a loop with conditional exits can be executed in a single cycle. A loop with exits contains an embedded control recurrence. Techniques presented here to reduce the height of this control recurrence are closely related to earlier techniques used to reduce the height of data recurrences within counted loops [7]. These techniques may be extended to multiple CPU parallelization, but this has not been attempted here.

Under restricted but commonly occurring conditions, loops with exits provide arbitrary amounts of parallelism while requiring no redundant computation. Most string operations are excellent examples of such computations. In fact, blocked back-substitution may reduce the number of operations executed per iteration in some cases, since some of the code to compute live-out values is moved out of the loop. The example presented in Section 3 illustrates such a case.

Height reduction techniques for loops with exits can also accommodate conditional branching within the body of the loop. In this work, such conditionals are executed using hardware that supports predicated execution. Predicated execution is also used to support the height reduction of the loop exit recurrence. However, for loops with no conditionals within the body, height reduction can be accomplished without the use of predicate hardware.

There are a number of potential obstacles to exposing parallelism within loops with exits. First, if the trip count is excessively short, little parallelism may be available. Second, a key obstacle to exposing parallelism regards the separability of stores to memory within the loop body. If stores are not separable, the control recurrence is strictly sequential and the techniques presented here

are not applicable. Finally, when loop performance is limited by the underlying data recurrences, the acceleration of control recurrences may provide no benefit. In general, we must apply techniques for the height reduction of both data and control recurrences to accelerate loop performance.

We summarize some results from earlier data recurrence work. Many data recurrences are readily accelerated without redundant code. A common example is the induction variable computation used to compute address sequences with loop invariant stride. Another important example is the reduction of a vector to a scalar using an associative operation, *e.g.*, summing the elements of a vector. When associative reductions are used within a loop with exit, the loop is fully parallelizable without need for redundant computation.

More complex data recurrences may require redundant computation. Some recurrences use add or add-multiply operations to compute a vector element in terms of previous vector elements. Here, the entire vector sequence is computed for storage into an array. For such sequences, a constant factor in redundant computation exposes unlimited parallelism. For example, a summation such as x(i) = x(i-1) + t(i) within the body of a loop provides unlimited parallelism at a cost of two additions per iteration. This represents a two fold redundancy in additions within the loop body. Because no redundancy is required for memory, address, and other calculations, the total required redundancy is less than two.

Some data recurrences, in particular some recurrences involving complex conditionals, are not readily accelerated. Techniques to accelerate them may generate excessively redundant code, in which case the acceleration of control recurrences is of no utility.

This work represents a step forward in our endeavor to liberate adequate program parallelism for execution on ILP machines. We feel that program transformations can substantially increase available parallelism in a broad range of codes which have been classically termed non-vectorizable or scalar. We are not yet able to measure the statistical utility of the techniques presented above. We leave this to future work.

The techniques presented can be generalized to provide acceleration within scalar code outside of loops. Here, the benefits may not be of the same magnitude. We do not understand limits for these techniques, and thus, are skeptical of the accuracy of measured "limits of parallelism" which have been derived from existing code without consideration for code transformation to enhance parallelism.

References

1. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Language Systems 9, 3 (1987), 319-349.

2. A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. IEEE Transactions on Computers C-33, 11 (1984), 968-976.

3. N. P. Jouppi and D. Wall. Available instruction level parallelism for superscalar and superpiplined machines. Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (1989), 272-282.

4. D. W. Wall. Limits of instruction-level parallelism. Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (1991), 176-188.

5. M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. Proceedings of the Nineteenth International Symposium on Computer Architecture (Gold Coast, Australia, 1992), 46-57.

6. D. J. Kuck. The structure of Computers and Computations. (Wiley, New York, 1978).

7. M. Schlansker and V. Kathail. Acceleration of first and higher order recurrences on processors with instruction level parallelism, in Sixth International Workshop on Languages and Compilers for Parallel Computing, U. Banerjee, *et al.*, Editor. 1993, Springer-Verlag: p. 406-429.

8. J. A. Fisher. Trace scheduling: A technique for global microcode compaction. IEEE Transactions on Computers C-30, 7 (1981), 478-490.

9. P. Tirumalai, M. Lee, and M. S. Schlansker. Parallelization of loops with exits on pipelined architectures. Proceedings of the Supercomputing '90 (1990), 200-212.

10. W.-M. W. Hwu, *et al.* The Superblock: An effective technique for VLIW and superscalar compilation. The Journal of Supercomputing 7, 1/2 (1993), 229-248.

11. S. A. Mahlke, *et al.* Sentinel scheduling: A model for compiler-controlled speculative execution. ACM Transactions on Computer Systems 11, 4 (1993), 376-408.

12. J. A. Fisher. Global code generation for instruction-level parallelism: trace scheduling-2. Technical Report HPL-93-43. Hewlett-Packard Laboratories, Palo Alto CA., 1993.

13. R. P. Colwell, *et al.* A VLIW architecture for a trace scheduling compiler. IEEE Transactions on Computers C-37, 8 (1988), 967-979.

14. P. Y. T. Hsu and E. S. Davidson. Highly concurrent scalar processing. Proceedings of the Thirteenth Annual International Symposium on Computer Architecture (1986), 386-395.

15. B. R. Rau. Cydra 5 directed dataflow architecture. Proceedings of the COMPCON '88 (San Francisco, 1988), 106-113.

16. B. R. Rau, *et al.* The Cydra 5 departmental supercomputer: Design philosophies, decisions and trade-offs. Computer 22, 1 (1989), 12-35.

17. J. C. H. Park and M. S. Schlansker. On predicated execution. Technical Report HPL-91-58. Hewlett-Packard Laboratories, Palo Alto CA, 1991.

18. V. Kathail, M. S. Schlansker, and B. R. Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80. Hewlett-Packard Laboratories, Palo Alto CA, 1993.

19. B. R. Rau. Data flow and dependence analysis for instruction level parallelism, in Fourth International Workshop on Languages and Compilers for Parallel Computing, U. Banerjee, *et al.*, Editor. 1992, Springer-Verlag: p. 236-250.

20. B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. Proceedings of the Fourteenth Annual Workshop on Microprogramming (1981), 183-198.

21. J. C. Dehnert, P. Y. T. Hsu, and J. P. Bratt. Overlapped Loop Support on the Cydra 5. Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (Boston, Mass, 1989), 26-38.

22. J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. The Journal of Supercomputing 7, 1/2 (1993), 181-228.

23. N. J. Warter, *et al.* Reverse if-conversion. Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation (Albuquerque, New Mexico, 1993), 290-299.

24. B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation schemas for modulo scheduled DO-loops and WHILE-loops. Technical Report HPL-92-47. Hewlett-Packard Laboratories, Palo Alto CA, 1992.

25. F. H. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745. Lawrence Livermore National Laboratory, 1986.

26. S. A. Mahlke, *et al.* Effective compiler support for predicated execution using the hyperblock. Proceedings of the 25th Annual International Symposium on Microarchitecture (1992), 45-54.