An Integrated Approach to Accelerate Data and Predicate Computations in Hyperblocks

Alexandre Eichenberger
North Carolina State University, Raleigh, NC
alexe@eos.ncsu.edu

Waleed Meleis and Suman Maradani
Northeastern University, Boston, MA
{meleis, smaradan}@ece.neu.edu

Abstract

To exploit increased instruction−level parallelism available in modern processors, we describe the formation and optimization of tracenets, an integrated approach to reducing the length of the critical path in data and predicated computation. By tightly integrating selective path expansion and path optimization within hyperblocks, our algorithm is able to produce highly optimized code without exploring the exponentially large number of paths included in a hyperblock. Our approach extracts more of the implicit predicate correlations in hyperblocks and uses a precise model of predicate correlations to aggressively accelerate data and predicate computations. Experimental results indicate that tracenets can significantly reduce the number of dynamic execution cycles.

1. Introduction

Advances in chip technology and computer microarchitecture have contributed to the increase in instruction level parallelism provided by modern microprocessors, as exemplified by Intel's IA−64 implementation of an Explicitly Parallel Instruction Computing (EPIC) architecture. However, sufficient parallelism to exploit these resources is rarely present in typical applications. Instead, hardware and compiler techniques are used to increase parallelism, including hardware techniques that support speculation and predication and compiler techniques that optimize regions of code such as traces [Fis81], superblocks [Hwu93], and hyperblocks [Ma92].

A successful compiler techniques is superblock scheduling [Hwu93], which selects frequently executed paths in the application and optimizes them under the assumption that each path is mostly executed from start to finish. Increased performance is achieved by overlapping the execution of operations from the consecutive blocks in the superblock and by optimizing the operations more effectively since optimizations are significantly more successful along a single path than in multiple paths of control flow.

However the effectiveness of superblock scheduling is reduced in highly branching code as the probability of executing any given path is reduced. Hyperblock scheduling [Ma92] alleviates this problem by using predicated execution [HD86] to include multiple paths within a single scheduling and optimization region. A hyperblock is formed by applying if−conversion [AKP83][DT93][Ma92], a process that eliminates the branches guarding each included path and guards the operations in each path by a predicate that evaluates to true precisely when the path would have been taken. Predicated operations have side effects only when their guarding predicate is true. Hyperblock scheduling achieves higher performance by eliminating branches and by enabling scheduling and optimizing within a region that covers a larger fraction of the execution time since they include multiple high−frequency paths.

Recent advances in predicated code optimizations reduce the length of the critical path in hyperblocks to increase instruction level parallelism. One technique eliminates the dependences among consecutive branches by moving some branches off trace [SMJ99]. Another technique accelerates the computation of the predicates by systematically restructuring their computation [Aug99]. A third technique eliminates false dependences in a hyperblock by renaming registers and replicating operations [Car99].

In this paper we describe tracenet formation, an integrated approach to reducing the length of the critical path in data and predicated computation. A *tracenet* is a hyperblock containing a network of traces expanded based on reaching values. By tightly integrating selective path expansion and path optimization, our algorithm produces highly optimized code without exploring the exponentially large number of paths in a hyperblock. Selective path expansion expands paths only when optimization opportunities are enabled. Paths that are determined to be infeasible are not considered further. Tracenets accelerate the data computations by increasing the effectiveness of classic optimizations like constant propagation, constant folding, and copy propagation. In addition, tracenets and these optimizations enable the algorithm to extract more of the implicit correlations that exist among predicates of the hyperblock. A precise model of the complete set of possible predicate correlations is used to define invariants which implicitly list all feasible combinations of correlated tests. These invariants allow the algorithm to aggressively minimize predicate computations and generate faster computations of predicate values.

Compared to other path expansion schemes [Car99], our approach targets not only elimination of false dependences but also height reduction along true dependences. Furthermore, while performing more aggressive path expansion, we may expand fewer paths by tightly integrating path expansion with optimizations like constant subexpression elimination that enable us to focus on values rather than on static assignments. Our approach considers the acceleration of data computation needed by predicate compares, extracts implicit correlation among predicates, and uses the predicate correlation data aggressively. These transformations enable the optimization of predicate computations performed by August et al. [Aug99]. While August considers the "implies" and "excludes" relationships among predicates, we classify predicate correlations using a precise model that covers all possible correlations.

The paper is organized as follows. In Section 2, we describe the architectural support for predicated execution. In Section 3, we motivate our approach to accelerating data and predicate computation. In Section 4, we describe the integrated path expansion and optimization algorithm. In Section 5, we present our approach to extraction of predicate correlation and its use in logic minimization. We present measurements in Section 6, compare our approach to related work in Section 7, and conclude in Section 8.

2. Architectural support

PlayDoh is a parameterized processor architecture designed to further research in instruction–level parallelism [KSR94]. Several features are provided to allow a compiler to fully exploit available ILP. The architecture supports both control and data speculation by adding speculative tag bits to each register, and by providing both speculative and non–speculative versions of many operations.

Predicated execution allows an operation to be conditionally executed based on the value of a boolean–valued register. In PlayDoh most operations can be predicated and a varied set of the compare–to–predicate operations that set the value of the predicate registers is provided. The syntax of these operations is:

$$p(\langle x\rangle),\ q(\langle x\rangle) = cond(a,b)\ \ ?\ \ r.$$

The boolean expression *cond(a,b)* and the guard predicate *r* are used to set destination predicates *p* and *q*. The final value of *p* and *q* depends on the actions specified by *<x>* and *<y>*, which can be one of un/uc, cn/cc, on/oc, or an/ac.

Inputs		**Outputs**							
r	**cond**	**un**	**uc**	**cn**	**cc**	**on**	**oc**	**an**	**ac**
0	**0**	0	0	–	–	–	–	–	–
0	**1**	0	0	–	–	–	–	–	–
1	**0**	0	1	0	1	–	1	0	–
1	**1**	1	0	1	0	1	–	–	0

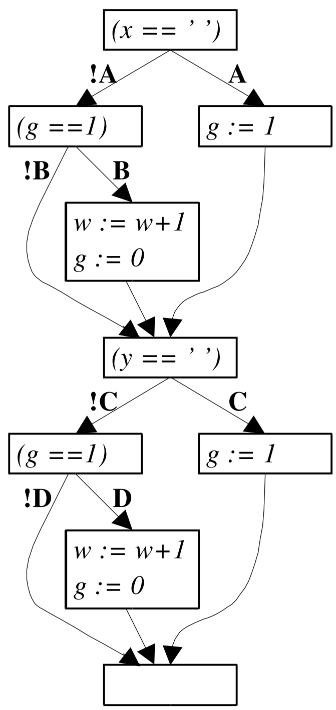
As a result of a compare–to–predicate operation, either a 1 or 0 is written into a destination predicate, or the destination predicate is left unchanged. These three outcomes (0,1,–) are indicated in the table above. In each case, the first letter of the action specifies the type of predicate setting operation, and the second letter indicates whether the normal or complemented version of the operation is being used.

The unconditional predicate types, un/uc, set the destination predicate equal to the logical conjunction of the expression (or its complement) and the guard predicate. The remaining predicate types leave the destination predicate unchanged if the source predicate equals 0. The OR predicate types, on/oc, set the destination predicate to 1 if the expression (or its complement) is true, and otherwise leave the destination unchanged. This operation can be used to compute the wired–or of a collection of expressions by initially setting the destination to 0, and then applying a compare–to–predicate operation with the same destination to each expression. In a similar way, the AND predicate types, an/ac, can be used to compute the wired–and of a collection of expressions. The conditional predicate types, cn/cc, are identical to the unconditional actions when the guard predicate is true. Otherwise the destination predicate is unchanged.

3. Motivation

Eliminating merge points along frequently executed paths in the control flow graph is critical to enabling aggressive optimization and extraction of instruction–level parallelism. A control merge is a point in the program where two or more execution traces merge. When the merging execution traces have distinct definitions of the same variable, *v*, the control merge is also called a data merge for variable *v*. Data merges disable useful optimizations because the compiler cannot determine which code was executed before the control merge. Examples of optimizations disabled by data merges include constant propagation and common subexpression elimination.

a) WC loop, unrolled once



b)

Paths	**Statements**	**Results**
AC	*(x==' ') g:=1 (y==' ') g:=1*	*g:=1*
A!CD	*(x==' ') g:=1 (y!=' ') (g==1) w++ g:=0*	*g:=0 w++*
A!C!D	*(x==' ') g:=1 (y!=' ') (g!=1)*	infeasible
!ABC	*(x!=' ') (g==1) w++ g:=0 (y==' ') g:=1*	*g:=1 w++*
!AB!CD	*(x!=' ') (g==1) w++ g:=0 (y!=' ') (g==1) w++ g:=0*	infeasible
!AB!C!D	*(x!=' ') (g==1) w++ g:=0 (y!=' ') (g!=1)*	*g:=0 w++*
!A!BC	*(x!=' ') (g!=1) (y==' ') g:=1*	*g:=1*
!A!B!CD	*(x!=' ') (g!=1) (y!=' ') (g==1) w++ g:=0*	infeasible
!A!B!C!D	*(x!=' ') (g!=1) (y!=' ') (g!=1)*	–

c)

Live–out values	**Tests**	**Simplified Tests ***
g:=1	AC + !ABC + !A!BC	C
g:=0	A!CD + !AB!C!D	A!C + B!C
w++	A!CD + !ABC + !AB!C!D	!AB + A!C
unfeasible	A!C!D + !AB!CD + !A!B!CD	

* use infeasible as "don't care"

d) Optimized code

```
<1>  x:=load[..]; y:=load[..]; p1:=1; p2:=1; p3:=1;
<2>  p1(an),p2(ac):=(x==' '); p2(an),p3(an):=(g==1);
     p1(ac),p3(ac):=(y==' '); p0(un):=(y==' ');
<3>  g:=1?p0; g:=0?p1; g:=0?p3; w=w+1?p1; w=w+1?p2;
```

Figure 1. All path optimization of the innermost loop of word count, unrolled once.

Modern scheduling algorithms eliminate data merge points by constructing and optimizing operation traces that do not contain control merges. For example, a superblock includes one path in the control flow and eliminates all data merges within this path. A hyperblock includes multiple paths from the original control flow, eliminates all control merges within the hyperblock, but still includes some data merges that are implicitly present in the predicated code.

Ideally, we would like to have a region that includes multiple predicated paths without any optimization–disabling data merges along its critical paths. This goal can be achieved by fully expanding each path in a region, optimizing each path individually, and recombining any operations that are redundant among the expanded paths. This approach is demonstrated below.

Figure 1 shows the innermost loop of wc, taken from the benchmark suite distributed with Trimaran [Tri99] and simplified for conciseness: we assume here that words are only separated by spaces (not tabs or newlines) and we do not count lines. The variable *w* contains the current number of words and the variable *g* is the boolean variable that indicates if the last characters were spaces. Variables *x* and *y* contain the characters being analyzed.

In Figure 1a we show the control flow graph for wc after the inner loop has been unrolled once. The graph contains operations on data values, e.g. *w++* and *g:=0*, and tests that affect the flow of control, e.g. *(g == 1)* and *(x == ' ')*. We let boolean variables *A, B, C*, and *D* represent the results of the four tests in the code, and the outgoing arcs from each test are either labeled with the appropriate variable or its complement. Figure 1b lists the operations and tests executed on each path. In many cases the sequence of operations performed along a path can be optimized. For example along path *AC*, one of the two *g := 1* operations is redundant. On the right side of Figure 1b we show the optimized code for each path. Note three out of nine paths are infeasible since they correspond to inconsistent sequences of operations and test results.

By computing predicates that correspond to each path and predicating the operations that are executed along that path, the optimized paths can be merged into a single code sequence. Further optimization is possible by observing that many of the paths execute identical operations. For example, variable *w* is incremented along paths *A!CD*, *!ABC*, and *!AB!C!D*. In Figure 1c we show the boolean expressions that correspond to paths that cause each optimized operation to be executed.

We can apply logic minimization techniques to find simplified conditions under which each operation is executed. Certain paths are infeasible and the expressions that correspond to these paths are considered to be don't–cares and can be set to whatever value gives the simplest expression. Using the infeasible paths *A!C!D*, *!AB!CD*, and *!A!B!CD* as don't–cares, the simplified conditions in Figure 1c fully eliminated test *D,* since test *D* is fully determined by *g*'s assignments and tests *A, B*, and *C*.

This transformation merges operations that correspond to the same static operation in the original code, such as operation *w++* in the first unrolled iteration that is executed along paths *!ABC* and *!AB!C!D,* as well as operations that correspond to different static operations in the original code, such as operation *w++* in the second unrolled iteration that is executed along path *A!CD*. This transformation preserves the semantics of the code since each instance of *w++* increments the same live–in value of variable *w*.

We show the optimized code in Figure 1d. In cycle 0, variables *x* and *y* are loaded; in cycle 1, the predicate values are computed; and in cycle 2, the data operations are issued with the simplified predicates. Computing *p=!AB+A!C* and executing *w++ ?p* ordinarily takes three cycles on an EPIC architecture. However, we can simultaneously issue one instance of the *w++* operation predicated on *p1=A!C,* and another instance predicated on *p2=!AB*. Since expressions *A!C* and *!AB* are disjoint (i.e. they cannot both be true), at most one instance will execute.

The resulting optimized code executes in 3 cycles for two loop iterations, compared to the 5 cycles obtained with traditional and predicate optimizations. Similarly, unrolling twice yields code that requires 3 cycles (one for loads, one for computing the predicate conditions, and one for executing the operations) while using traditional optimizations gives code that requires 7 cycles.

The drawback of this naive approach is that it individually investigates and optimizes a number of paths that is exponential in the number of tests. In practice it is useful to expand and optimize only a subset of these paths, or subsets of partial paths. Our algorithm generates the same optimized code without generating all paths by (1) tightly integrating path expansion and code optimization to detect infeasible paths and their associated reaching definitions early, and by (2) selectively expanding subsets of paths that generate distinct values to enable optimizations for operations in the critical path. Correlation among predicates is used to detect infeasible paths and reaching definitions as well as to minimize the code defining the predicates.

The first key insight is that we expand paths based on values, not on control flow or original operations. In Figure 1 we saw that variable *w* is incremented along three control paths by two distinct operations in the original control flow graph. If distinguishing between the original and the incremented values of *w* enables useful optimizations, our algorithm would expand two paths: one for the original value and another one for the incremented value. In contrast, a full control path expansion approach would expand all six distinct control paths. Similarly, a static operation path expansion [Car99] would expand three distinct paths: one for the original value of *w*, one for the value of *w* generated by the first increment operation, and one for the value of *w* generated by the second increment operation in the original control flow graph.

The second key insight is that we treat predicate compare operations like regular operations. By expanding paths that compute the values being compared and by optimizing along such paths, we extract more correlation among compares which in turn can lower the overhead to compute and set the predicates. Also, we can break dependences from predicate computation, to regular operation, back to predicate computation, as seen in Figure 1 by the elimination of the data–dependent computation of *D* from the final optimized code.

4. Tracenet formation

We give an overview of the algorithm in Section 4.1, illustrate how it works on the innermost loop from wc in Sections 4.2 and 4.3, and describe the algorithm in detail in Section 4.4.

4.1. Algorithm overview

We propose an algorithm that expands traces in a hyperblock and generates the same optimized code as full path expansion without having to actually consider each path. Paths are expanded as follows. Assume we are processing operation *g:=1* and that we want to distinguish this value of variable *g* from previous values of *g*. To do so, we rename the destination register of the operation, e.g. g_1*:=1*. Each subsequent operation that uses variable *g* may now have to be replicated, with one instance using the register containing the original value of variable *g,* e.g. g_0, and another instance using register g_1. We say that g_1 is a renaming of variable *g*, since g_1 is a distinct register that holds a value originally contained in variable *g*.

The algorithm processes each operation in turn, in program order. Consider that it is currently processing operation *x* which uses variable *v* and defines variable *w*. The algorithm first determines whether more than one register that is a renaming of v reaches operation *x.* If this is the case (e.g. when v_1 and v_2 are two renamings of *v* and when some definitions of v_1 and v_2 reach operation *x*) the algorithm replicates operation *x* to create one instance per renamed register. Replicated operations are added after *x* in the program order and are processed like any of the original operations. When replication occurs, the original operation *x* is removed and the algorithm tries to optimize the replicated operations by exploiting the specific definitions that reach them. After optimization, the algorithm decides whether to rename the destination register of *x* by evaluating whether renaming generates optimization opportunities along the critical path. Finally, the algorithm records the definitions generated by *x* and updates the reaching definition data accordingly.

While we replicate and optimize predicate compares as if they were regular operations, we never rename the predicate registers since the computations of the predicates are accelerated separately using an approach similar to the one proposed by August et al [Aug99]. The current value of the predicate registers is analyzed using an approach similar to the one proposed by Johnson and Schlansker [JS96].

```
(1)   p0(un), p1(uc) := (x==' ')
(2)   g := 1 ?p0
(3)   p2(un) := (g==1) ?p1
(4)   w := w+1 ?p2
(5)   g := 0 ?p2
(6)   p10(un), p11(uc) := (y==' ')
(7)   g := 1 ?p10
(8)   p12(un) := (g==1) ?p11
(9)   w := w+1 ?p12
(10)  g := 0 ?p12
```

Figure 2. Unrolled, if-converted innermost loop from wc.

4.2. Introductory example: expanding path and optimization

Consider the if-converted code in Figure 2 constructed from the control flow graph in Figure 1a. Because all operations are on the critical path, the destination register of each operation will be renamed, provided it generates a new value. Thus, in this example the terms "definition" and "renamed register" are used interchangeably.

The data gathered by our algorithm is shown in Table 1, which reflects the state after processing each operation. The first column contains the original operations that have been processed so far. The second column indicates the symbolic definitions that result from each operation. These include the association of new logical variables with the result of a test, the association of a predicate register with a logical expression, and the assignment of values to definitions. The remaining columns indicate the logical expressions under which each definition reaches the current operation. Reaching definitions for predicate definitions are computed elsewhere [JS96] and are therefore omitted from the table.

The first operation is not replicated and cannot be further optimized. We record in the second column that operation (1) defines a test *(x==' ')*. We let A equal the result of this test. Register *p0* is equivalent to A and *p1* is equivalent to the complement of *A,* as defined by the semantics of the predicate types un and uc described in Section 2. The recording of symbolic definitions for predicate compare operations is similar to the one proposed by Johnson and Schlansker [JS96]. The remaining columns indicate that the original live-in definitions of *g* and *w*, g_0 and w_0 respectively, reach this operation unconditionally.

Operation (2) is not replicated and variable *g* is renamed as g_1. We record in the table the symbolic definition and update the reaching definition expressions. Since operation (2) is predicated on *p0*, which equals *A*, the definition g_1 is generated under *A* and the original definition of *g* is killed under *A*. The reaching definition expression associated with w_o is unchanged (as depicted in the table by downward pointing arrows).

Operation (3) is a predicate compare of type *un* predicated on *p1*. Unconditional predicate compares are atypical because, although they produce a value unconditionally, they read their inputs (*g* and *1* here) only when their guarding predicate evaluates to true. Of the two available definitions of variable *g*, g_1 reaching under expression *A* and g_0 reaching under expression *!A*, only the later is used by operation (3) guarded under expression *!A*. Thus operation (3) can be rewritten as $p2(un){:=}(g_0{==}1)?p1$. We record in the second column the new $B{\equiv}(g_0{==}1)$ test and the equivalence of *p2* to *!AB*.

We now consider operations (4)-(7). First, operation *w:=w+1?p2* generates a renamed register w_1 under expression *!AB;* second, operation *g:=0?p2* generates a renamed register g_2 under expression *!AB*; and third operation *p10(un),p11(uc):=(y==' ')* generates a new test labeled *C≡(y==' ')*. Consider now the processing of operation (7) which, like operation (2), sets variable *g* to 1. No replication is required, and common subexpression elimination indicates that the value 1 is already available in the renamed register g_1. Instead of having a new definition of *g* and/or renaming the variable *g* again, we regenerate the same symbolic definition $g_1{\equiv}1$ but this time under expression *C*.

The processing of operation (8) is illustrated in Figure 3. Because three definitions of *g* reach operation (8) in three distinct registers, as shown in Figure 3a, the predicate compare operation is replicated three times. Figure 3b shows each replicated operation guarded by the conjunction of the expression associated with its reaching definition and the original guard of operation (8). Note that an unconditional predicate compare can only be replicated by transforming all but the first replicated operation into or-type predicate compares. In Figure 3c, the constant symbolic definitions are propagated into the replicated operations. Note that the test $(g_0{==}1)$ was previously assigned the label *B*. In doing so, we correlate this test with the earlier identical test seen when processing operation (3). Predicate compare operations are evaluated in Figure 3d; the first one sets *p12* to false,

Original operations	Symbolic definitions	Reaching definition expressions				
		g_0	g_1	g_2	w_0	w_1
*(1)*** p0(un),p1(uc):=(x==' ')*	A≡(x==' '), p0≡A, p1≡!A	true.	false.	false.	true.	false.
(2) g:=1?p0	g_1≡1	!A	A	↓	↓	↓
(3) p2(un):=(g==1)?p1	B≡(g_0==1), p2≡!AB	↓	↓	↓	↓	↓
(4) w:=w+1?p2	w_1≡w_0+1	↓	↓	↓	A+!B	!AB
(5) g:=0?p2	g_2≡0	!A!B	↓	!AB	↓	↓
(6) p10(un),p11(uc):=(y==' ')	C≡(y==' '), p10≡C, p11≡!C	↓	↓	↓	↓	↓
(7) g:=1?p10	g_1≡1	!A!B!C	A+C	!AB!C	↓	↓
(8) p12(un):=(g==1)?p11	p12≡A!C	↓	↓	↓	↓	↓
(9) w:=w+1?p12	w_1≡w_0+1	↓	↓	↓	!A!B+AC+!BC	!AB+ A!C
(10) *g:=0?p12*	g_2≡0	!A!B!C	C	A!C+B!C	↓	↓

Table 1. Algorithm state after processing each operation in Figure 2.

the second one sets *p12* to true whenever *A!C* evaluates to true, and the third one never writes anything (i.e. the operation is dead). As a result, *p12* correlates with expression *A!C* as shown in Table 1.

Processing operation (9) does not require replication as only one definition of the variable *w* reaches the operation predicated on expression *A!C*. Note that common subexpression elimination recognizes that w_0+1 has been computed before, and thus no new definition or renamed register is generated. When processing operation (10), common subexpression elimination recognizes that a move of the value zero has been performed before, and this previous definition can be reused.

4.3. Introductory example: code generation

In the case of wc, generating code from Table 1 is straightforward since each reaching definition is either a constant or is a function of the live−in values. The first step is to determine which variable is live−out on the branch, which comes just after operation (10). Here both variables *w* and *g* are live−outs. The next step is to overwrite the original values of *w* and *g* with those of the renamed registers that are reaching the branch. Since the last row of Table 1 indicates the reaching definition expression for each of the renamed register just before the branch, we must overwrite the original value of *w* with the value of w_1 under the expression *!AB+ A!C.* Similarly, we must overwrite the original value of *g* with the value of g_1 under the expression *C* and with the value of g_2 under the expression *A!C+B!C.* Since $w_1 \equiv w_0+1$, $g_1 \equiv 1$, and $g_2 \equiv 0$, we issue the following code: *w++?(!AB+A!C); g:=1?C; g:= 0?(A!C+B!C)* or the code with simpler and faster predicate conditions: *w++?(!AB); w++?(A!C); g:= 1?C; g:=0?(A!C); g:=0?(B!C)* just as in Figure 1d.

In summary, the algorithm processed each of the operations in the innermost loop of wc, performing an all−path expansion and optimization without having replicated any operations that could not be immediately simplified. It generated one renamed register/definition per value by aggressively using constant propagation and common subexpression elimination. Without tightly integrating such optimization within the path expansion algorithm, the algorithm would have generated many more renamed registers, processed many more replicated operations, and may not have generated such compact code.

4.4. Detailed algorithm

We introduce notation to describe registers in the presence of register renaming. We refer to the inputs and outputs of an operation as uses and defs, respectively. For any two−input/one−output operation we let u_1 and u_2 refer to the first and second uses, respectively, and let d_1 refer to the def. We let $reg(u_1)$ and $reg(u_2)$ be the registers that are used and let $reg(d_1)$ be the register that is defined. We further define $oreg(u_1)$, $oreg(u_2)$, and $oreg(d_1)$ as the registers that are used and defined in the *original* hyperblock, prior to any renaming and replicating. Prior to calling our path expansion algorithm, $oreg(x) = reg(x)$ for any use or def *x* by any operation in the hyperblock. As the algorithm renames registers, $oreg(d_x)$ and $reg(d_x)$ start to differ for each operation *x* that defines a renamed register. Similarly, when an

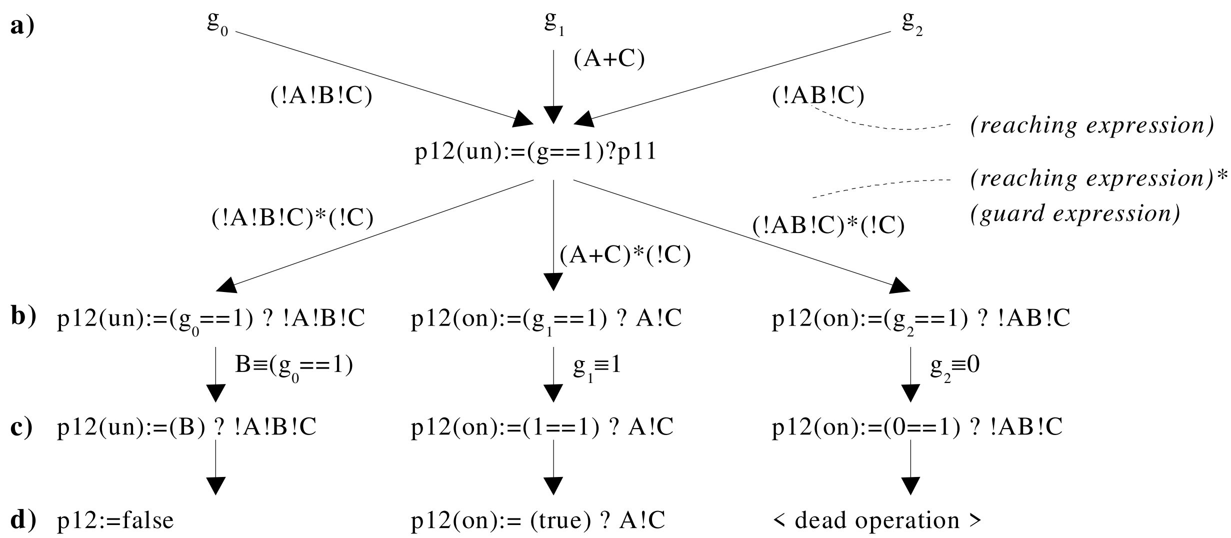


Figure 3. Replication and optimization for operation (8). a) Definitions of *g* that reach operation (8). b) Replicated operations with one instance per reaching definition in a). Replicated operation is predicated on the conjunction of the reaching definition expression and the guard expression of operation (8). c) Replicated operation after constant propagation of its symbolical definitions. d) Replicated operations after constant evaluation of the comparison.

operation is replicated to accommodate defs that reach the operation in renamed register, the replicated operation x is assigned uses u_x with $reg(u_x) \neq oreg(u_x)$. The above definitions are trivially extended to operations with different number of defs and uses.

We further define the expression under which a definition d_x reaches the current operation as *reach_expr*(d_x). Expressions are functions of boolean variables, where each boolean variable represents the test of one compare operation. Examples of boolean variables in Section 4.1 are A, B, and C which relate to test *(x ==' ')*, *(g_0==1)*, and *(y==' ')*, respectively. When manipulating expressions, summing is equivalent to oring, multiplying is equivalent to anding, and subtracting is equivalent to taking the difference. Actual expression manipulations are performed in this paper using the Binary Decision Diagrams [Bry86]. We also extend the notion of a reaching definition to registers: the expression under which a register r_x reaches the current operation is defined as *reach_expr*(r_x) and is equal to the sum of *reach_expr*(d_y) over all definitions d_y such that $reg(d_y) = r_x$. Finally, we let *guard_expr(op)* be the expression guarding the execution of *op*.

The algorithm that performs integrated path expansion and optimization is presented in Figure 4. It processes the current operation in three distinct phases. In the first phase, lines (2) through (10), it inspects the reaching definitions to determine whether to replicate the current op. While each operation of the introductory wc example of Section 4.1 used at most one register, the algorithm here handle operations with an arbitrary number of input registers.

In the second phase, lines (11) and (12), the algorithm attempts to simplify the current operation by performing constant propagation, copy propagation, constant folding, and common subexpression elimination. These optimizations are all forward optimizations, and thus fully benefit from any renaming that may have occurred while processing previous operations.

In the third phase, lines (13) through (21), the algorithm proceeds with renaming and analyzing the the side effect generated by the optimized version of the current operation. When the operation sets a predicate, it is analyzed as proposed by Johnson and Schlansker in [JS96] but using Binary Decision Diagrams as internal representation of the predicates and predicate expressions. This analysis is done on the fly since register renaming and optimizations affect the compared registers of predicate compare operations. We also build a predicate correlation database which we describe in more detail in Section 5. No reaching definition analysis is performed on predicate registers as it is done elsewhere [JS96]. For all other operations, the algorithm decides whether to perform register renaming and update the reaching definition expressions, one def at a time.

Some preprocessing is needed to properly handle live−in and live−out values. First, the live−in values are added into the initial reaching definitions under the expression true. Second, we must ensure that the live−out values are maintained in the presence of renaming. We do this by introducing dummy move operations guarded

```
process(op)
    // phase 1: perform replication if needed
    if op is an original operation then
        for each distinct use of op: u_i do
            rset(u_i) = {reg(d_j) | def d_j reaches u_i and oreg(d_j) = oreg(u_i)}
        for each unique tuple (r_1,..., r_n) where r_1∈rset(u_1),...,r_n∈rset(u_n) do
            expr = reach_expr(r_1)* ... *reach_expr(r_n)
            if (expr*guard_expr(op) ≠ false) then
                replicate op with renamed uses (r_1,...,r_n) and guarded by expr*guard_expr(op)
        if (replicated some operations) then remove original and return
    // phase 2: apply optimizations on op
    constant propagation, constant folding, common subexpression elimination,...
    // phase 3: perform renaming, predicate analysis, and predicate reaching analysis
    if (op is an operation that sets predicates) then
        analyze op as in PQS [JS96]
    else
        for each def of op: d_i do
            if (beneficial to rename d_i) then reg(d_i) = new reg name
            expr = expr under which def d_i is generated by op
            reach_expr(d_i) += expr
            for each def d_j with oreg(d_i) = oreg(d_j) do reach_expr(d_j) −= expr
    return

tracenet_formation(hyperblock)
    preprocess live−in and live−out
    for each op of hyperblock in program order do process(op)
    optimize live−out and perform speculation
    return
```

Figure 4. Tracenet formation algorithm.

by the same predicate that guards the branch. For example, if variable *v* is live along branch *branch L1 ? p1,* we insert prior to the branch the dummy move operation *v:=v ? p1*. If the variable *v* is renamed while processing the hyperblock, the dummy move will be replicated once per renamed register of *v*. If *v* is renamed twice, as v_1 and v_2, *we* replicate the original dummy move into $v{:=}v_1?p1$ and $v{:=}v_2?p1$ which precisely gathers the values of variable *v* back into the live−out register. Note that such move operations introduce a one cycle latency penalty, but such copy operations can be removed during the post−processing phase by either performing back−copy propagation or some additional operation replication.

5. Optimization of predicate computation

In this section, we present our approach to minimizing predicate computations, focusing on how to minimize the boolean expressions that define the predicate values. Once the boolean expressions are minimized, the actual generation of predicate compare operations is performed as proposed by August et al. [Aug99] and will not be further discussed here. The extraction of explicit correlation is presented in Section 5.1, the formulating of global predicate invariant is shown in Section 5.2, the minimization of predicate expression using invariant is described in Section 5.3, and additional sources of correlation are explored in Section 5.4

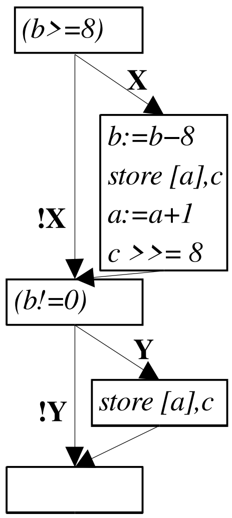
5.1. Extraction of explicitly correlated predicate compares

The wc example in Figure 1 allowed one predicate compare operation to be eliminated from the final optimized code (i.e. test *D* fully correlated with expression *A!C*). In this section, we expand on how predicate correlation is extracted using a more typical example from the inner loop from compress, shown in Figure 5a. The code here is from the output function where a compressed code (variable *c*) is stored in consecutive bytes of an array (variable *a*), depending on the number of interesting bits of data (variable *b*).

While the two branches in Figure 5a are clearly correlated, this correlation is implicit since both branches use different versions of variable *b* (i.e. the first test uses the initial value of *b* and the second test uses a value of *b* which might have been decremented by 8). Because of the dependences from the first test to the decrement operation to the second test, a minimum of 4 cycles are needed to execute this code fragment.

We now show how path expansion enables us to extract explicit correlation from implicitly correlated code. The three operations that contribute to determining the control flow are illustrated in the left column of

a) Code fragment



b) Decision logic operations

original operations	after renaming	symbolic definitions
(1) p1(un):=(b>=8)	*(1') $p1(un):=(b_0>=8)$*	$A\equiv(b_0\geq 8)$, $p1\equiv A$
(2) b:=b−8?p1	*(2') $b_1:=b_0-8?p1$*	$b_1\equiv b_0-8$
(3) p2(un):=(b!=0)	*(3') $p2(un):=(b_0!=0)?(!p1)$*	$B\equiv(b_0\neq 0)$, $p2\equiv !AB$
	(3'') $p2(or):=(b_1!=0)?\ p1$	$C\equiv(b_1\neq 0)\equiv(b_0\neq 8)$, $p2\equiv !AB+AC$

c) Correlation among test symbols

correlation:	A⇒B, A+C=true, B+C=true
feasible paths:	(!A+B)*(A+C)*(B+C)
infeasible paths:	A!B+!A!C

d) Optimized code with p2=(!AB+AC) simplified to BC

<1> *p1(un):=(b>=8); p2(an):=(b!=0); p2(an):=(b!=8);*

a1:=a+1; b1:=b−8; c1:= c>>8;

<2> *store [a],c?p1; store[a1],c1?p2;*

Figure 5. Fragment of the function output called in the inner loop of compress.

Figure 4b. The middle column of Figure 5b depicts the same operations after the renaming of the def register of operation (2) and the replicating of operation (3), with one instance using b_1 under predicate predicate *p1* and another instance using b_0 under the complement of predicate *p1*. Consider now the symbolic definitions in the left column of Figure 5b. We extract from operation (1') and (3') the tests $A\equiv(b_0\geq 8)$ and $B\equiv(b_0\neq 0)$, respectively. We also extract from operation (3'') the test $(b_1\neq 0)$, which can be rewritten as $C\equiv(b_0\neq 8)$ by folding the definition $b_1\equiv b_0-8$ into $(b_1\neq 0)$. As a result, all three tests are a function of the same original value of *b*. This transforms two implicitly correlated tests into three explicitly correlated tests.

5.2. Gathering correlation data

We expand in this section on the type of correlation data that we gather, expanding on previous work [Aug99]. Consider operation *x* which uses register r_x and operation *y* which uses register r_y. We say that both operations *yield the same* value if both operations get the same reaching definitions under the same reaching definition expressions.

In our work, we correlate any two tests that satisfy one of the following criteria.

- Both operations compare one register to one constant, and both uses of their respective registers yield the same value.
- Both operations compare two registers, and the first use by the first operation yields the same value as one of the uses by the second operation and the second use by the first operation yields the same value as the other use by the second operation.

Relation	Logical invariant	Example 1		Example 2	
		test A	test B	test A	test B
A **implies** B	!A + B	$x > y$	$x \geq y$	$x > 3$	$x > 2$
A and B are a **partition**	A ⊕ B	$x > y$	$x \leq y$	$x > 3$	$x \leq 3$
A and B are **disjoint**	!(AB)	$x > y$	$x = y$	$x = 3$	$x = 2$
A and B **cover_all**	A+B	$x \geq y$	$x \neq y$	$x > 3$	$x < 7$

Table 2. Four correlation types among tests (*x* and *y* yield the same value for both tests *A* and *B*).

When a pair of compare operations satisfies one of these criteria, the two operations are correlated and their correlation falls into one of the following four types: *imply, partition*, *disjoint*, and *cover_all*, as described in Table 2. "Imply" means that if the first test is true, the second test must be true; "partition" means that exactly one of the two tests must be true; "disjoint" means that at most one test can be true; "cover_all" means that at least one of the tests must be true.

In general, stronger correlations can be obtained by jointly considering relations of three or more tests. For example, every pair of the *(x<y)*, *(x=y)*, and *(x>y)* tests are disjoint, but they only form a partition when considering the three of them jointly. Such opportunities can be identified using the "families of comparisons" concept proposed by August et al [Aug99].

While each of the correlation types is useful in some context (e.g. partition and disjoint can eliminate false output dependence, cover_all can determine when a live range is killed, etc.), we use a more systematic representation of these relations. We compute an invariant expression that must be true regardless of the particular outcome of any tests. For example, "*A* implies *B*" is the same as stating that *!A+B* is always true. The second column of Table 2 indicates the logical invariant associated with each of the four correlation types.

Consider our example in Figure 5. Given the $A \equiv (b_0 \geq 8)$, $B \equiv (b_0 \neq 0)$, and $C \equiv (b_0 \neq 8)$ symbolic definitions, we conclude that *A* implies *B, A* and *C* cover_all, and *B* and *C* cover_all. The logical invariants associated with the correlations between tests are *(!A+B)*, *(A+C)*, and *(B+C)*, respectively. Each relation must hold regardless of whether *A, B*, and *C* are true or false during one execution trace. Thus, we may compute the global invariant as the conjunction of each of the individual logical invariants, i.e. the global invariant is *(!A+B)*(A+C)*(B+C)=AB+!AC* in Figure 5b.

5.3. Minimizing boolean expressions

To minimize boolean expressions, we proceed as follows. First, we gather all the correlation data and summarize the correlation in a global invariant, say *i*. In effect, invariant *i* describes all the feasible ways in which the logical variables representing the tests may be combined. When minimizing an expression *f*, we only need to compute this expression for the subset of expression *f* that does not contradict the invariant, i.e. we may minimize only the expression *f*i*.

Conversely, it is acceptable to include some of the infeasible logical variable combinations when minimizing an expression if they enable us to derive a simpler expression. Thus, we may treat the complement of the invariant, *!i,* as don’t–cares while minimizing the *f*i* expression.

For example in Figure 5b, the expression for predicate *p2* is (*!AB+AC*), which includes the terms *!AB!C, !ABC, A!BC,* and *ABC*. Both the *!AB!C* and *A!BC* terms do not intersect with the invariant *AB+!AC* since the first term contradicts *A* and *C* cover_all, and the second term contradicts *A* implies *B*. Thus we can minimize the smaller (*!ABC + ABC)* expression instead of the larger (*!AB+AC*) expression. Using the complement of invariant *AB+!AC* as don’t–cares, minimizing the expression for *p2* yields expression *BC*, which can be computed in one cycle. The final optimized code is shown in Figure 5d: in cycle 1, the predicates are computed and all but the store operations are speculatively executed; and in cycle 2, the two store operations are executed. This contrasts to the 4 cycles that are needed when the correlation between tests are left implicit.

5.4. Additional sources of correlation data and performance asserts

The more correlation data gathered, the more restrictive the global invariant is, and the more concise the minimized expressions will be. So far, correlation data was directly gathered from the tests in the predicate compare operations, but correlation data can be found elsewhere as well.

One source of correlation data is implicitly defined by branches. Consider a branch operation *b* and its guarding predicate *p.* Since the code below branch *b* is executed only if the branch is not taken, we can assume when minimizing predicates that are only used below branch *b* that the test defining predicate *p* must be false. Other types correlations may be gathered from the code. When identifying a boolean variable, we can use this information to strengthen correlation data (e.g. if *b* is a boolean variable, the correlation between tests *(b=0)* and *(b=1)* can be strengthened from disjoint to partition). A third source of correlation can be the programmer itself. Defensive programmers often use asserts in their code to catch programming errors. We can provide a similar construct, e.g. termed "passert" for performance assert, to convey more global, program wide invariant to the logic optimizer. Like traditional assert statements, which cause the program to stop when their asserted boolean expression is violated, "passert" statements convey a boolean expression that is guaranteed to be true, or else the correctness of the program becomes irrelevant.

6. Case study

We implemented our algorithms in the Trimaran System [Tri99]. We first added a Binary Decision Diagram [Bry86] predicate representation into the interface of the predicate query system [JS96] to handle complex predicates accurately. We then integrated a two level boolean expression minimization package, Espresso [Bra84], to minimize small expressions optimally and larger expressions heuristically. We also implemented a predicate code generation scheme similar to August et al [Aug99] to generate fast and compact predicate code from the optimized predicate expressions. Our algorithm then uses these components to expand and optimize the paths, simplify predicate expressions, and generate predicated code.

The baseline measurements are obtained by: (1) applying unrolling, traditional optimizations, hyperblock formation, and predicate promotion within Impact [Cha91]; (2) performing program decision logic minimization, scheduling (with speculative execution), and register allocation in Elcor [Rau98]; (3) by simulating the executable within the ReaCT_ILP cycle simulator. The tracenet measurements are obtained by inserting the proposed tracenet formation algorithm before the logic minimization step in Elcor.

machine width	benchmarks		
	wc	129.compress	130.li
16	1.200	1.073	1.087
32	1.500	1.077	1.087
64	1.599	1.077	1.087

Table 3. Dynamic cycle count speedup of tracenet over baseline

The dynamic cycle count speedups of tracenet versus baseline for the full version of wc, 129.compress and 130.li is shown in Table 3 for machines with varying numbers of fully−pipelined general−purpose functional units. Latencies are 1 cycle for ALUs, stores, and branches, 2 cycles for loads, 3 cycles for floats and integer multiplications, and 8 cycles for divisions. At most one branch is scheduled per cycle. Some dynamic effects such as branch mispredictions, cache and TLB misses are not measured to highlight more precisely the effect of the proposed optimizations.

The results indicate that tracenets reduces the number of dynamic cycles for wc, 129.compress, and 130.li by up to 59.9%, 7.7%, and 8.7%, respectively. Note that tracenets currently requires wide machines to achieve good speedup. Manual inspection of the optimized code indicates that some operation replication is not needed for height reduction. Such superfluous replication can be reduced by using a finer model where decisions concerning optimization and replication are made based on a cycle by cycle cost benefit analysis.

7. Conclusions

We have described the creation of tracenets, an extension of hyperblocks that contain a network of expanded traces. Interleaving path expansion with optimization aggressively reduces the length of the critical path and a detailed correlation model reduces the complexity of predicate computations. We are able to extract more implicit predicate correlations that were previously hidden by layers of interacting data and predicate computations. A case study indicates that tracenets can significantly reduce the number of dynamic cycles , e.g. up to 59.9% for wc, 7.7% for 129.compress and 8.7% for 130.li. Future work includes a finer resource and dependence model to guide optimization and renaming decisions during tracenet formations.

Acknowledgments

We thank B. Ramakrishna Rau and the Compiler and Architecture Research Group at HP Labs, Wen−mei W. Hwu and the IMPACT Research Group at the University of Illinois at Urbana−Champaign, and Krishna Palem and the ReaCT−ILP group at the New York University for making their research compiler available.

References

[AKP83] J.R. Allen, K. Kennedy, C. Porterfield, J. Warren. Conversion of control dependences to data dependences, *Conf. Record of POPL−10*, 1983.

[Aug99] D. August et al., The Program Decision Logic Approach to Predicated Execution, *Proc. of the 26th Intl. Symp. on Computer Architecture*, 1999.

[Bra84] R. Brayton et al., Logic Minimization Algorithms for VLSI Synthesis, Kluwer Acad. Pub., 1984.

[Bry86] R.E. Bryant. Graph−Based Algorithms for Boolean Function Manipulation, IEEE Transaction on Computers, C35(8), 1986.

[Car99] L. Carter et al., Predicated Static Single Assignment, in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 1999.

[Cha91] P. Chang et al., IMPACT: An Architectural Framework for Multiple−Instruction−Issue Processors, *Proc. 18th Int'l Symp. Computer Architecture*, 1991.

[DT93] J. Dehnert and R. Towle. Compiling for the Cydra−5. *Journal of Supercomputing*, 7(1), 1993

[Fis81] J. Fisher, Trace scheduling: a technique for global microcode compaction, *IEEE Trans. on Comp*, C−30:478−490, 1981.

[HD86] P. Y. T. Hsu and E. S. Davidson, Highly concurrent scalar processing, in *Proc. 13th Ann. Int'l Symp. Computer Architecture*, 1986.

[Hwu93] W. Hwu et al., The superblock: an effective technique for VLIW and superscalar compilation, *The Journal of Supercomputing*, 1993, 229−248.

[JS96] R. Johnson, M. Schlansker. Analysis techniques for predicated code. *Proc. 29th Ann. IEEE/ACM Intl. Symp. on Microarchitecture*, 1996.

[KSR94] V. Kathail, M. Schlansker, B. R. Rau, HPL PlayDoh Architecture Specification: Version 1.1, Computer Systems Laboratory, *Hewlett Packard Technical Report HPL−93−80(R.1)*, 2000.

[Ma92] S. Mahlke et al., Effective compiler support for predicated execution using the hyperblock, *Proc. of the 25th Intl Symp. on Microarchitecture (MICRO−25),* pp. 45−54, 1992.

[Ma96] S. A. Mahlke, Exploiting Instruction Level Parallelism In the Presence of Conditional Branches, *Ph.D. Thesis, University of Illinois at Urbana−Champaign*, 1996.

[Rau98] B. R. Rau, V. Kathail, S. Aditya, Machine Description Driven Compilers for EPIC Processors, *Technical Report HPL−98−40, Hewlett−Packard Laboratories,* 1998.

[SMJ99] M. Schlansker, S. Mahlke, R. Johnson. Control CPR: A Branch Height Reduction Optimization For EPIC Architectures. *PLDI*, 1999.

[Tri99] The Trimaran System, www.trimaran.org, 1999.