Using Hammock Graphs to Structure Programs

Fubo Zhang and Erik H. D'Hollander, *Member*, *IEEE*

Abstract—Advanced computer architectures rely mainly on compiler optimizations for parallelization, vectorization, and pipelining. Efficient code generation is based on a control dependence analysis to find the basic blocks and to determine the regions of control. However, unstructured branch statements, such as jumps and goto's, render the control flow analysis difficult, time-consuming, and result in poor code generation. Branches are part of many programming languages and occur in legacy and maintenance code as well as in assembler, intermediate languages, and byte code. A simple and effective technique is presented to convert unstructured branches into hammock graph control structures. Using three basic transformations, an equivalent program is obtained in which all control statements have a well-defined scope. In the interest of predication and branch prediction, the number of control variables has been minimized, thereby allowing a limited code replication. The correctness of the transformations has been proven using an axiomatic proof rule system. With respect to previous work, the algorithm is simpler and the branch conditions are less complex, making the program more readable and the code generation more efficient. Additionally, hammock graphs define single entry single exit regions and therefore allow localized optimizations. The restructuring method has been implemented into the parallelizing compiler FPT and allows to extract parallelism in unstructured programs. The use of hammock graph transformations in other application areas such as vectorization, decompilation, and assembly program restructuring is also demonstrated.

Index Terms—Program transformation, structured programming, compilers, optimization, parallel processing, software/program verification, correctness proofs.

1 Introduction

A structured program is a block-oriented program in which each block of statements has a single entry, a single exit, and the blocks are properly nested. Using block-oriented control constructs, such as while and if-then-else statements, most imperative languages enforce a structured style. However, branches such as jumps and goto's, interacting with block-oriented statements, make these statements unstructured, because there is no single locus of control.

The work presented here describes a method to convert programs with unstructured branches into structured form. The motivation is that structuring is necessary for a compiler to perform optimizations such as parallelization and vectorization. To this end, we present a new restructuring algorithm which limits code expansion and maintains a simple control structure. The algorithm eliminates all branches, maintains the basic block lengths, preserves structured loops and if-tests, and introduces only a few logical variables. The new approach is based on the creation of nested hammock graphs. These single-entry, single-exit graphs create well-defined nested control regions which can be separately optimized and parallelized. The method has been implemented in a parallelizing compiler and was able to generate a significant number of extra parallel loops. Furthermore, examples illustrate that the resulting code is generally more readable.

This research grew out of the need to parallelize and vectorize unstructured programs. The resulting restructuring method is simple, implementable in a compiler, provably correct, and generates a less complex control flow. A major benefit is that hammock graph restructuring builds a hierarchy of structured code regions. This enlarges the granularity and makes global code optimization possible, such as loop transformations, data locality enhancements, and other techniques used to find parallelism [14], [21], [38]. It is well known that unstructured code such as irreducible loops, i.e., loops with more than one entry point [16], may block optimization because there is no information on the control flow [15]. Existing front-end compiler methods are directed towards loop normalization [3] and goto elimination [13]. Back-end methods repair irreducible loops by node splitting (replication) and DJ-graphs [19], [32]. The present method is a front-end approach which repairs unstructured regions by creating hammock graph structures in the control flow graph.

Hammock graph restructuring operates on the control flow graph of the source code and eliminates all branches while minimizing the number of new control variables and associated control expressions. As a consequence, the basic blocks keep the same length as in the original program and the branching conditions maintain the same complexity. The conversion of the program is based on three elementary transformations, namely, the backward copy, forward copy, and cut operations. The transformations reorganize the program into a set of nested hammock graphs, each having a single entry and a single exit. As an example, consider the program in Fig. 1. The loop created by `goto 100` has two entries and is therefore irreducible. After a backward copy Fig. 1b and a simple if-conversion Fig. 1c, the resulting

• *F. Zhang is with Platform Computing Inc., 3760 14th Ave., Markham, Ontario L3R 3T7, Canada. E-mail: zfb@platform.com.*
• *E.H. D'Hollander is with the Department of Electronics and Information Systems, Ghent University, St.-Pietersnieuwstraat 41, B-9000 Ghent, Belgium. E-mail: dhollander@elis.ugent.be.*

Manuscript received 26 Aug. 2003; revised 23 Jan. 2004; accepted 29 Jan. 2004.
Recommended for acceptance by J. Knight.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0121-0803.

```
    if (x) goto 200
100 S1
200 S2
    if (y) goto 100
```

(a)

```
    if(x) goto 200
100 S1
200 S2
    do while(y)
      S1
      S2
    enddo
```

(b)

```
if(.not. x) then
  S1
endif
S2
do while(y)
  S1
  S2
enddo
```

(c)

Fig. 1. (a) Irreducible loop with two statements S1, S2, (b) after backward copy, (c) after single branch to if-conversion.

```
1:    S0
2:    if (X) then
3:10    S1
4:    else
5:      S2
6:      if (Y) goto 10
7:    endif
8:    S3
```

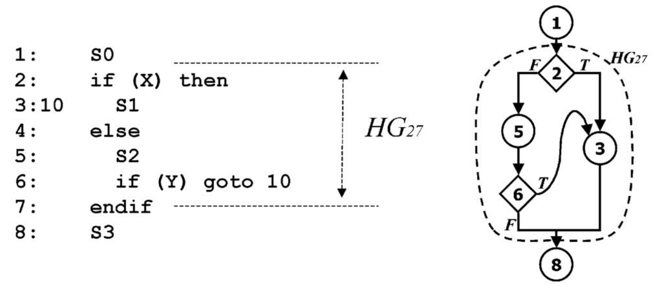


Fig. 2. A program and its control flow graph. The subgraph bounded by the dashed line is hammock graph HG_{27} with entry node 2 and exit node 7.

program contains two hammock graphs. In contrast to the original program Fig. 1a, the structured program enables optimizations which are otherwise impossible due to lacking control flow information. For example, the while loop may be converted into a parallel loop subject to a dependence analysis [38]. Further examples of optimization are given in Section 5.

The correctness of the three transformations has been proven using an axiomatic proof rule system. The remainder of this text is organized as follows: In Section 2, definitions and a basic corollary are given. The methods for converting branches into block structured control statements are developed in Section 3. Here, program transformations for single branches, multiple interacting branches, and branches interacting with structured control statements are developed. The correctness of the program transformations is formally proven in Section 4 using axiomatic proof rules for branches. A number of experimental results in the area of parallel processing are reported in Section 5. In Section 6, the related work is described and compared with the proposed approach. Concluding remarks are given in Section 7.

2 Definitions

A program consists of sequential statements and control statements. During the execution, a *sequential statement* is followed by the next statement in the program whereas the statement following a *control statement* is selected by a conditional jump. The statement ordering during the execution of the program is described by the control flow graph.

Definition 1: Control flow graph of a program. *The control flow graph of a program, $CFG =< N, E, n_0, n_e >$, is a directed graph $< N, E >$ in which the nodes N represent the statements and the edges E indicate the conditional transfer of control between the statements. There is a path from node $n_0 \in N$ to all other nodes, and all paths end with node $n_e \in N$. n_0 is called the initial node and n_e is called the terminal node.*

A path in the control flow graph traversing nodes n_i, e.g., $\{n_1, n_2, \cdots, n_k\}$ is called an *execution trajectory* [33]. The control flow graph represents the possible execution trajectories. In the control flow graph of a program, regions with a single entry and a single exit are hammock graphs. Several formal definitions for hammock graphs exist, e.g., see [14], [20]. Here, the definition from Ferrante et al. [14] is used.

Definition 2: Hammock graph. *Let $CFG = < N, E, n_0, n_e >$ be a control flow graph and consider a subgraph $H = < N', E', u, v >$ with a distinguished node u in H and a distinguished node v not in H such that 1) all edges from $(CFG - H)$ to H go to u, 2) all edges from H to $(CFG - H)$ go to v. H is called a hammock graph. Node u is called the entry node and v is called the exit node of H.*

Fig. 2 shows a program, the control flow graph and a hammock graph. This example will be used to illustrate the key steps of the algorithm.

The single-entry, single-exit property of hammock graphs allow us to confine the restructuring region. Since a hammock graph transformation doesn't generate side effects on the rest of the program, the following corollary holds.

Corollary 1. *A correct transformation of the control flow in a hammock graph maintains the correctness of the program.*

Hereby, we remind that that a restructuring transformation does not depend on data outside the hammock graph and that code duplication and new variables as a result of the transformation are invisible for code outside the hammock graph.

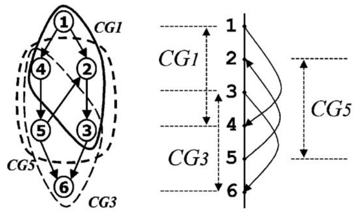
Ill-structured code is identified by the type and interaction of the control statements. There are two types of control statements: block-oriented statements and branches.

Definition 3: Block-oriented and branch statements. *A* block-oriented *control statement governs the execution of a sequential block of statements delineated by language dependent markers, e.g., braces in the C-language or* `do`, `enddo` *in Fortran. Block-oriented statements can be nested, and the* nesting level *of a statement is the* number of block-oriented statements guarding its execution. *A* branch *control statement is a statement of the type:* `if (<bcond>) goto <label>`, *where* `<bcond>` *is the branch condition and* `<label>` *identifies the target statement. The* `goto` *statement is called the branch, and the* `<label>` *is called the target. This definition includes unconditional jumps, and statements with implicit targets, such as* `stop` *and* `return`.

Definition 4: Branch types. *A branch is classified as* backward *or a* forward *if its target occurs, respectively, before or after the branch in lexical order. The lexical order is defined as the statement ordering, also called the program ordering [2], [22]. A branch in a block-oriented statement is called* incoming *or* outgoing *when the branch and its target are not in the same block and the target is respectively inside or*

```
1:          if(X) goto 200
2:   100    S1
3:          goto 300
4:   200    S2
5:          if(Y) goto 100
6:   300    S3
```

(a)



(b) (c)

Fig. 3. An example program (a) and its control flow graph (b). The three subgraphs in (b) are the control graphs of respectively statements 1, 3, and 5. Graph (c) is a CFG where the nodes are vertically aligned according to the statement order in the program.

outside the block. For example, a branch in a while-loop with a target outside the loop body is an outgoing branch.

Definition 5: Branch order. *Let i_f be a forward branch. If no forward branch* lexically *occurs before i_f, then branch i_f is called the* initial *forward branch. Let i_b be a backward branch. If no backward branch lexically occurs before i_b, then branch i_b is called the* initial *backward branch.*

Definition 6: Structured control statement. *A control statement p is structured if it is block-oriented and each block of statements controlled by p is a* hammock graph.

Note that, according to this definition, a *block-oriented statement* such as a loop or an if-then-else statement is structured *only if* the loop body or the if-blocks are hammock graphs. For example, the block-oriented `if`-statement in Fig. 2 is *not* structured because the inner if-statement jumps from the `else` part to the `then` part. In general, whenever a block contains an incoming jump, the block becomes unstructured.

Definition 7: Control graph of a control statement. *Let $CFG = < N, E, n_0, n_e >$ be a control flow graph and let node $p \in N$ represent a control statement. The* control graph *of p is the subgraph $CG_p = < \Gamma, \Delta, p, q >$, where q is defined as follows: 1) If p is a block-oriented control statement, then q is lexically the last node of the block of statements controlled by p; 2) if p is a branch, then q is the target node of p. Furthermore $\Gamma = \{a | a \in N$ and a is a statement between p and q in the source program, including p and q $\}$ and $\Delta = \{(a, b) | a, b \in \Gamma$ and $(a, b) \in E\}$.*

Informally, the control graph CG_p describes the region of the program directly controlled by statement p. Fig. 3 shows the control graphs of three branches. In Fig. 2, the control graph of the if-then-endif statement is the nodes 2-7, i.e., HG_{27}, while the control graph of the if-goto statement is the nodes 3-6. Control graphs may be disjoint, fully nested or they may *interact*.

Definition 8: Interacting control statements. *Let $CG_{i_0} = < \Gamma, \Delta, i_0, i_e >$ and $CG_{m_0} = < \Gamma', \Delta', m_0, m_e >$ be the control graphs of two control statements i_0 and m_0, where $i_0 \neq m_0$. If the set of nodes Γ and Γ' partially intersect, i.e., $(\Gamma \cap \Gamma' \neq \phi) \wedge (\Gamma' \not\subseteq \Gamma) \wedge (\Gamma \not\subseteq \Gamma')$, then control statements i_0 and m_0* interact. *If i_0 interacts with m_0 and m_0 interacts with control statement k_0, then i_0 and k_0 interact indirectly.*

Interacting control statements create an indeterminate control flow which at compile time prevents optimizing loop transformations because it is difficult to determine the dataflow and the dependencies. A minimal hammock graph of a branch is found by including all control graphs which interact directly or indirectly with the control graph of the branch.

Definition 9: Minimal hammock graph of a branch. *Let n_b be a branch statement in a program with control flow graph $CFG = < N, E, n_0, n_e >$. The* minimal hammock graph *of n_b, denoted $MHG_{n_b} = < N', E', m_0, m_e >$, is the smallest hammock subgraph of CFG, containing n_b.*

The minimal hammock graph MHG_{n_b} of a branch n_b defines its region of control. An algorithm to find the minimal hammock graph of a branch statement n_b is given below.

Algorithm 1
(find the minimal hammock graph of branch n_b)

input: n_b	–	a branch;
output: MHG_{n_b}	–	the minimal hammock graph containing n_b.

```
{ $MHG = CG_{n_b}$;
  while ($\exists$ branch $i_b$ | $(CG_{i_b} \cap MHG \neq \phi) \wedge$
      $(CG_{i_b} \not\subseteq MHG) \wedge (MHG \not\subseteq CG_{i_b}))$
        $MHG = MHG \cup CG_{i_b}$;
  return($MHG$);
}
```

In this algorithm, the while condition selects the branches i_b of which the control graph CG_{i_b} *partially* intersects MHG. Applying the algorithm to the if-statement at line 5 in Fig. 3 gives iteratively statements (nodes) $\{2-5\} \cup \{1-4\} \cup \{3-6\} = \{1-6\}$.

Definition 10: Structured program and control flow graph. *If all control graphs of a control flow graph are structured, i.e., are hammocks, then the contro flow graph is called* complete hammock *and the program is* well-structured.

3 The Conversion of Branch Statements

The block-oriented control structures used are if-then-else-endif, while, and repeat. In this section, branches are converted into structured control statements.

A single branch. A single branch is a branch not interacting with other branches or block-oriented control statements. As such its control graph is a hammock graph. A single forward branch is replaced by a *block-if* and a single backward branch is converted into a *while* or a *repeat* statement. In this way, the branch is replaced by a structured control statement.

Multiple interacting branches. Interacting branches are restructured by type: first backward branches are converted into loops containing no incoming branches. In the process forward branches out of a loop are replaced by an "exit" statement followed by a jump to the target. This is repeated until no backward branches remain. Next the forward branches are converted one by one into structured if-statements.

Branches interacting with block-oriented statements. A programing language usually contains block-oriented control statements such as do loops or if-then-else statements. When a block-oriented control statement interacts with a branch, it prevents hammock graph restructuring. As an example the if statement at line 6 in Fig. 2 interacts with the block-if statement. Therefore, first, the block-oriented if-statements are converted into branches. Then, branches out of loops are cut into exit statements inside the loop and conditional branches outside the loop. This makes the loops single-entry, single-exit hammocks. Next, the branches are removed.

3.1 Algorithm Outline

The hammock graph restructuring algorithm is based on three types of code transformations, respectively, for outgoing, backward, and forward branches.

The outline of the algorithm is as follows:

1. **Preprocessing**. Replace if-then-else statements interacting with branches by if-goto statements.
2. **Single branches**. Replace single branches by structured block-if or loop statements.
3. **Outgoing branches**. Replace an outgoing branch in a loop by an exit statement and put a conditional jump to the original target after the loop. This is the Cut transformation (Algorithm 2).
4. **Backward branches**. For each backward branch, determine the minimal hammock graph enclosing the branch. Starting with the initial backward branch i_b in this graph, convert each backward branch into a loop and move the targets of incoming branches outside the loop body (Algorithm 3). Cut each outgoing forward branch in the loop into an exit-jump followed by a conditional jump to the target of the branch
5. **Forward branches**. For each forward branch, determine the minimal hammock graph enclosing the branch. Starting with the initial forward branch, convert each forward branch into a structured if-statement (Algorithm 4).

After removal of the branches, the structuring may offer extra opportunities to remove dead code (e.g., see Fig. 7e) or to beautify the result, e.g., by using alternative looping constructs. This could be taken care of in a postprocessing step, based on specific compiler optimizations requested by the user. The following subsections present these techniques in more detail.

3.2 Cut Transformation

Loops with outgoing forward branches are unstructured. Therefore the *Cut* conversion is applied: 1) a new variable br_i is used to register the state of the branch condition and then the loop control expression is modified using the variable br_i, 2) the long jump is cut into two parts, one is within the loop and jumps to the end of the loop using the `exit` statement, the other is located outside the loop and conditionally jumps to the original target. Algorithm 2 realizes this transformation.

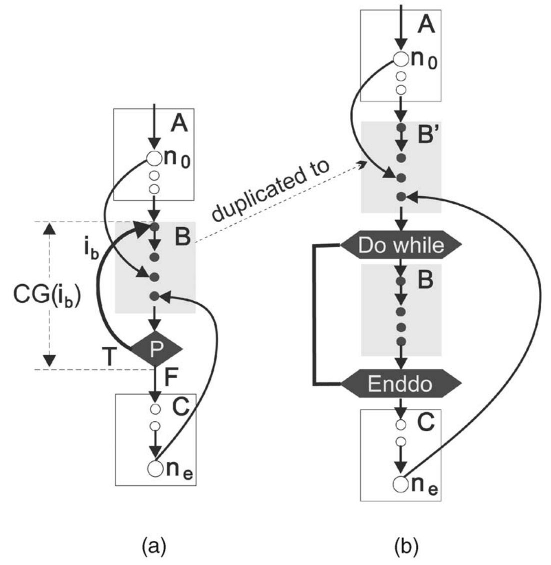


Fig. 4. Illustration of the backward branch elimination. (a) The minimal hammock graph MHG_{i_b}, initial backward branch i_b, beginning and end nodes n_0 and n_e. (b) After converting i_b into a while statement, there are two nested hammock graphs: the body of the while loop, B, and the blocks {A, B', do while and C}.

Algorithm 2 (elimination of outgoing branches)
input: loop_nest – a loop nest with outgoing branches;
output: loop_nest – a loop nest without outgoing branches.

```
Cut(loop_nest){
for $l =$ the innermost to the outermost loop
  { $n =$ the number of exit branches in loop $l$;
    Before loop $l$, insert:
      1. $fp_i$ = .false. $(1 \leq i \leq n)$
    For each outgoing branch $i$ with guard condition $B_i$
        $(1 \leq i \leq n)$:
    { replace branch $i$ to target $t_i$ by:
      2. if $(B_i)$ then {$fp_i$ =.true.; exit }
    } After loop $l$ insert:
      3. if $(fp_i)$ goto $t_i$ $(1 \leq i \leq n)$
    }
  }
```

3.3 Backward Copy

Let i_b be an arbitrary backward branch. Principally, a backward branch can be represented by a repeat-statement if there are no incoming branches into the loop body. An incoming branch prevents the backward branch from being directly converted into a loop. Since the body of this loop is executed at least once, we unroll the loop one time and modify the repeat loop into a while loop. The labels of incoming branches are moved out of the loop. As Fig. 4 illustrates, after unrolling the loop, the incoming branches n_0 and n_e are moved outside the loop and the backward branch is converted into a while-loop. This process is called the *Backward copy*. However, we cannot take an arbitrary backward branch and use Backward Copy in order to remove this branch. For example, suppose in Fig. 4 we convert n_e first instead of i_b into a repeat-loop. Because loop overlapping is not allowed, i_b cannot be directly converted

```
Algorithm 3 (elimination of backward branches)
input:  $CFG$ – a control flow graph with backward branches;
output: $CFG'$ – a control flow graph without backward branches.

Backward_copy($CFG$ ) {
 while (there exists a backward branch $b$ in $CFG$ )
   { Find the minimal hammock graph of $b$, $MHG_b$;
     Let $i_b$ be the initial backward branch in $MHG_b$;
     Let $B_b$ be the conditional expression of branch $i_b$;
     $block\_st = CG_{i_b} - \{i_b\}$; /* statements controlled by $i_b$, including labels */
     /* create a loop */
     if (there exist incoming branches into $block\_st$) /* convert into while */
     { $loop\_body = block\_st$ without labels of incoming targets;
                     labels of internal branches are renamed.
       $CG_{i_b}$ is replaced by
       1. $block\_st$;
       2. while ($B_b$) $loop\_body$ enddo;
     }
    else /* no incoming targets, convert into repeat until */
     { $loop\_body = block\_st$;
       $CG_{i_b}$ is replaced by
       1. repeat $block\_st$ until ($\neg B_b$);
     }
     /* cut long outgoing branches in the new loop */
    while (there exists an outgoing forward branch $f$ in $loop\_body$)
     { /* the cut transformation is applied */
       Let $t_f$ be the target of $f$ and $B_f$ the guard of branch $f$;
      Before the loop, insert:
       1. $br_f$ = .false.
      Replace branch $f$ with:
       2. if ($B_f$) then {$br_f$ = .true.; exit}
      After the loop insert:
       3. if ($br_f$) goto $t_f$
     }
   } /* end of while there exist backward branches */
}
```

Fig. 5. Algorithm 3: Elimination of backward branches.

into either a repeat-loop or a while-loop. When the *initial* backward branch is converted first, there are no internal backward branches going out of the loop.

The *loop_body* contains only target labels of internal branches. Labels used as targets of both internal and external branches are renamed, and the internal branches are redirected to the new internal labels.

When a backward branch is converted into a loop, the only remaining outgoing branches are forward branches. These are removed using the Cut transformation. After removing the first initial backward branch, the next initial backward branch is selected and removed, and so on, until no backward branches remain. The steps are shown in Algorithm 3 (Fig. 5).

3.4 Forward Copy

At this point, the backward branches are converted into loops without incoming branches by Algorithm 3. Outgoing branches are eliminated using `exit` statements. As a consequence, the only remaining branches are forward branches branches both of whose end points appear at the same nesting level. A forward branch transfers control to the target or to the next statement, depending on a Boolean condition. The first trajectory is called the *true_part*, the other trajectory is called the *false_part*. Forward branches are eliminated starting with the initial forward branch. Then, for each initial branch i_f, the surrounding *minimal hammock graph* is determined and the initial branch is structured.

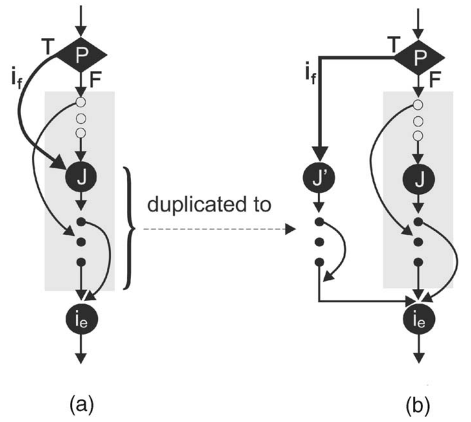


Fig. 6. (a) The minimal hammock graph MHG_{i_f}. i_f is a forward branch whose target is J, and i_e is the terminal node. (b) The shared statements are duplicated into the true part of i_f. The brace indicates the shared statements.

Given the initial forward branch i_f, the minimal hammock graph $MHG_{i_f} = < N, E, i_f, i_e >$ contains all forward branches interacting with i_f (see Fig. 6). The statements on the path between the target of i_f and the terminal node i_e are called the *shared statements* in MHG_{i_f}. The shared statements are determined as follows: Let CG_{i_f} be the control graph of forward branch i_f. Furthermore, let J be the target of the branch i_f, then one has $shared\ statements = (MHG_{i_f} - CG_{i_f} - \{i_e\}) \cup J$. Branch i_f is converted into a block-if where the shared statements are duplicated into the *true* part and the statements inside the minimal hammock graph MHG_{i_f} are moved into the *false* part. The following algorithm eliminates forward branches in a program where backward branches and outgoing branches have been eliminated.

```
Algorithm 4 (elimination of forward branches)
input:   $CFG$ – a flowgraph with only forward branches;
output:  $CFG'$ – a flowgraph without branches.

Forward_copy($CFG$) {
 { while (there exists an initial forward branch $i_f$)
   { Find the minimal hammock graph of $i_f$, $MHG_{i_f}$,
     with end node $i_e$;
     if ($CG_{i_f}$ interacts with other forward branches)
     { /* a forward-copy transformation is applied */
       true_part = the shared statements =
         $(MHG_{i_f} - CG_{i_f} - \{i_e\}) \cup J$;
       false_part = $MHG_{i_f} - \{i_f, i_e\}$;
       $MHG_{i_f}$ is replaced by:
```

```
     if(.not.x) goto 20
10     S1
     goto 30
20     S2
     if(y) goto 10
30     S3
```

(a)

```
     if(.not.x) goto 20
       S1
     goto 30
20     S2
     do while (y)
       S1
       goto 30
       S2
     enddo
30   s3
```

(b)

```
     if(.not.x) goto 20
       S1
     goto 30
20     S2
     do while (y)
       S1
       exit
       S2
     enddo
30   S3
```

(c)

```
if(x) then
  S1
else
  S2
  do while (y)
    S1
    exit
    S2
  enddo
endif
S3
```

(d)

```
if(x) then
S1
else
S2
if(y) then
    S1
endif
endif
S3
```

(e)

Fig. 7. Restructuring steps of the unstructured code inside the hammock graph HG_{27} in Fig. 2. (a) Conversion into branches, (b) backward copy, (c) cut outgoing branch, (d) forward copy, (e) unconditional exit optimization.

```
      1. if ($B$) then true_part else false_part endif;
    } /* $B$ is the Boolean expression of $i_f$.*/
    else /* a single forward branch */
     {true_part = $MHG_{i_f} - \{i_f, i_e\}$;
       $MHG_{i_f}$ is replaced by:
       1. if ($\neg B$)  then true_part endif; $\{i_e\}$;
     }
    Eliminate_forward(true_part);
    Eliminate_forward(false_part);
  }
}
```

Example. The algorithm is applied to the hammock graph H_{27} of Fig. 2. Fig. 7 shows the successive steps:

1. First, the block-if is converted into branches because the then-part contains an incoming branch.
2. Next, the backward copy creates a while loop without incoming branch.
3. The cut transformation converts the outgoing branch into an exit. Since the branch and while loop have the same successor, no logical variable is necessary.
4. The forward copy removes the last branch. The shared statements in the then-part are eliminated due to dead code removal.
5. Finally, the compiler replaces the single while-loop iteration by a block-if statement.

In the following section, the correctness of the basic hammock graph transformations is formally proven using axiomatic proof rules.

4 Correctness Proof of the Elementary Program Transformations

The elementary transformations used to construct hammock graphs are limited to the *Forward copy*, the *Backward copy*, and the *Cut transformation*. These transformations operate on a branch by branch basis and each handles a particular type of interacting branch. Since no other transformations are used, the basic cases cover all branches. In this section, the correctness proofs of these transformations are developed based on an axiomatic program specification [18], [17], [24], [1]. The formulas of the inference system have the form $\{P\}\ S\ \{Q\}$, where S is a block of statements representing a hammock graph. Branches are either within S or are expressed explicitly. P is a precondition and Q is a postcondition. The proof rules for structured statements are given in Table 1. Unstructured statements containing a **goto** to label L have the form:

$$\{P\}\ \textbf{if}\ (B)\ \textbf{goto}\ L\ \{Q\}\ \{L : R\}. \qquad (1)$$

There are two postconditions: $\{Q\} = \{P \wedge \neg B\}$ for the normal exit and $\{L : R\} = \{L : P \wedge B\}$ for the jump to the statement labeled $L :$. If the computation state satisfies

TABLE 1
Axiomatic Proof Rules for Structured Control Statements

consequence rules	$\dfrac{\{P\}\ S\ \{R\},\ R \Rightarrow Q}{\{P\}\ S\ \{Q\}}$ $\dfrac{\{P\} \Rightarrow R,\ \{R\}\ S\ \{Q\}}{\{P\}\ S\ \{Q\}}$
compound rule	$\dfrac{\{P_{i-1}\}\ S_i\ \{P_i\},\ \text{for } i = 1 \ldots n}{\{P_0\}\ S_1;\ S_2;\ \ldots;\ S_n\ \{P_n\}}$
if rule	$\dfrac{\{P \wedge B\}\ S_1\ \{Q\},\ \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ (B)\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{endif}\ \{Q\}}$ $\dfrac{\{P \wedge B\}\ S\ \{Q\},\ \{P \wedge \neg B\} \Rightarrow \{Q\}}{\{P\}\ \textbf{if}\ (B)\ \textbf{then}\ S\ \textbf{endif}\ \{Q\}}$
while rule	$\dfrac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{do while}\ (B)\ S\ \textbf{enddo}\ \{P \wedge \neg B\}}$
repeat rule	$\dfrac{\{P\}\ S\ \{Q\},\ \{Q \wedge \neg B\} \Rightarrow \{P\}}{\{P\}\ \textbf{repeat}\ S\ \textbf{until}\ (B)\ \{Q \wedge B\}}$

precondition P, then on a normal exit, Q is asserted, while on the jump **goto** L, the computation state satisfies R. In addition, R is an invariant condition which is satisfied for all computing states reaching label L. The inference rules for goto's are given in Table 2.

The case **goto** L is treated as **if** $(true)$ **goto** L. Applying the goto rule, this gives: $\{P\}$ **goto** L $\{false\}$ $\{L : P\}$.

Definition 11: Semantic equivalence. *Let $\{P\}$ be an arbitrary precondition. S and S' are programs, and $\{P\}\ S\ \{Q\}$, $\{P\}\ S'\ \{Q'\}$. The program S is semantically equivalent to the program S', if and only if $Q = Q'$.*

Definition 12: Invariant condition. *An invariant condition U is a predicate at label L in the program such that all incoming computation states satisfy U. If an incoming computation state s at L satisfies predicate R, then one has $R \Rightarrow U$.*

Theorem 1: Backward copy. *The Backward copy transformation preserves the semantics of the program.*

Proof. A *Backward copy* is applied when there exists a target for an incoming branch within the control region of an initial backward branch, such as label L in the program shown in Fig. 8a. Denote L the target of an incoming branch. The Backward copy converts the program (a) (Fig. 8) into (b) (Fig. 8). Suppose $\{P\}$ (Fig. 8a) $\{Q\}$ and $\{P\}$ (Fig. 8b) $\{Q'\}$. Let U and R_2 be the invariant conditions respectively before S_1 and after S_2 in program (a) (Fig. 8). Now, location L has two or more incoming states: the state after S_1 and the state of an arbitrary incoming branch to target L. Let R_1 be a condition which is satisfied by those states. Hence, $\{U\}\ S_1\ \{R_1\}$, $\{R_1\}\ L : S_2\ \{R_2\}$. Furthermore, U is an invariant condition at the label L_1 and all incoming states must satisfy U, therefore $R_2 \wedge B \Rightarrow U$ and $P \Rightarrow U$. The proof rules of the program (a) (Fig. 8) are given in Fig. 9. With $P \Rightarrow U$ and $R_2 \wedge B \Rightarrow U$, the consequence and the forward labeling rules give: $\{P\}\quad L_1 : S_1 L : S_2$ **if** (B) **goto** $L_1 \quad \{R_2 \wedge \neg B\}$. Hence, $\{Q\} = \{R_2 \wedge \neg B\}$. Using U and R_i as defined in Fig. 9, rewriting the clauses and applying the axiomatic proof rules, program (b) (Fig. 8) is derived as shown in Fig. 10: Since the postcondition of program (b) (Fig. 8b), $Q' = R_2 \wedge \neg B = Q$, the transformed program is semantically equivalent to the original. Furthermore, the incoming target is moved out of the loop. □

TABLE 2
Inference Rules for Labels and Goto's

goto rule	$\{P\}$ **if** (B) **goto** $L\ \{P \wedge \neg B\}\ \{L : P \wedge B\}$
compound (1)	$\dfrac{\{P\}\ S_1\ \{Q_1\}\ \{L : R\},\ \{Q_1\}\ S_2\ \{Q_2\}\ \{L : R\}}{\{P\}\ S_1;\ S_2\ \{Q_2\}\ \{L : R\}}$
compound (2)	$\dfrac{\{P\}\ S_1\ \{Q_1\}\ \{L_1 : R_1\},\ \{Q_1\}\ S_2\ \{Q_2\}\ \{L_2 : R_2\}}{\{P\}\ S_1;\ S_2\ \{Q_2\}\ \{L_1 : R_1\}\ \{L_2 : R_2\}}$
backward labeling	$\dfrac{\{P\}\ S_1\ \{R\}\ \{L : R\}}{\{P\}\ S_1\ L\text{:}\ \{R\}}$
forward labeling	$\dfrac{\{R\}\ S_1\ \{Q\}\ \{L : R\}}{\{R\}\ L\text{:}\ S_1\ \{Q\}}$

```
L_1:  S_1 ;                           S_1
L  :  S_2 ;                     L  :  S_2
      if (B) goto L_1                 do while (B)
                                        S_1
                                        S_2
                                      enddo
            (a)                             (b)
```

Fig. 8. Elementry backward copy transformation: (a) backward branch, (b) do while conversion.

Theorem 2 (Cut transformation). *The Cut transformation preserves the semantics of the program.*

Proof. The *Cut transformation* breaks a branch out of a loop into two branches. The first branch just exits the loop, the second is located after the loop and transfers the control to the target of the original far jump. In this way, the program (a) (Fig. 11) is transformed into (b) (Fig. 11). To prove that the transformed program (b) (Fig. 11) is semantically equivalent to the program (a) (Fig. 11), let $\{P\}$ (Fig. 11a) $\{Q\}$ and $\{P\}$ (Fig. 11b) $\{Q'\}$. Assume U is an invariant condition before the loop. Furthermore, define R_i, $i = 1, \ldots, 3$, such that $\{U \wedge B_1\}\ S_1\ \{R_1\}$, $\{R_1 \wedge \neg B_2\}\ S_2\ \{U\}$ and $\{U \wedge \neg B_1\}\ S_3\ \{R_3\}$. Because U is an invariant condition before the loop, one has $\{P\} \Rightarrow \{U\}$. Now, the program (a) (Fig. 11) is annotated with the assertions shown in Fig. 12. Hence, the postcondition at L_2 is $Q = R_3 \vee R_1 \wedge B_2$.

In program (b) (Fig. 11), the instruction **exit** is introduced. Semantically, this instruction modifies the postcondition of the **dowhile** construct, $\{U \wedge \neg B_1\}$, by adding the disjunction of the exit condition, R_e, giving rise to the new postcondition, $\{U \wedge \neg B_1 \vee R_e\}$. A new logical variable br_2 is introduced to register the state of the Boolean expression B_2. br_2 is outside the state space of the original program. Therefore, neither S_1 nor S_2 nor S_3 change the state of br_2, i.e.,

$$\{R_{i-1}\}\ S_k\ \{R_i\} \Rightarrow \{R_{i-1} \wedge br_2\}\ S_k\ \{R_i \wedge br_2\}, (1 \le k \le 3).$$

$P \Rightarrow U$, $\{U\}$	$L_1 : S_1$	$\{R_1\}$
$\{R_1\}$	$L :\ S_2$	$\{R_2\}$
$\{R_2\}$	**if** (B) **goto** L_1	$\{R_2 \wedge \neg B\}$ $\{L_1 : R_2 \wedge B \Rightarrow U\}$
$P \Rightarrow U$, $\{U\}$	$L_1 : S_1$ $L : S_2$ **if** (B) **goto** L_1	$\{R_2 \wedge \neg B\}$
		$\{L_1 : U\}$ (compound rule)

Fig. 9. Computing the postcondition of Fig. 8a.

Using the proof rules of program (a) (Fig. 11), program (b) (Fig. 11) can be derived next (see Fig. 13). L_w is a placeholder for the exit-node of the while-loop.

The postcondition of program (b) (Fig. 11) is $Q' = R_3 \wedge \neg br_2 \vee R_1 \wedge B_2 \wedge br_2$. Since the branch variable br_2 can take either value $.true.$ or $.false.$, depending on the program, one has $Q' \Rightarrow R_3 \vee R_1 \wedge B_2$. Therefore, $Q' \Rightarrow Q$ and the transformation is valid. □

Since a repeat statement can be represented by a while statement, it can be proven similarly that the *Cut* transformation in the case of a repeat statement is valid.

Theorem 3: (Forward copy). *The Forward copy transformation preserves the semantics of the program.*

Proof. Consider program (a) (Fig. 14) containing interacting forward branches. The forward branches have targets L_1 and L_2, respectively. The minimal hammock graph of the initial forward branch extends from the first statement to target label L_2. The *Forward copy* converts the initial forward branch into a block-if statement by duplicating the shared statement S_3 into the *true* part of the block-if (see Fig. 6). The resulting program is given in (Fig. 14b). The second noninteracting forward branch is similarly converted into a block-if, resulting program (c) (Fig. 14). Let $\{P\}$ (Fig. 14a) $\{Q\}$ and $\{P\}$ (Fig. 14c) $\{Q'\}$. Now, define R_i in program (a) (Fig. 14) as follows: Suppose $\{P \wedge \neg B_1\}\ S_1\ \{R_1\}$, denote R_2 and R_3 the invariant preconditions at the label L_1 and L_2 and let $\{R_3\}\ S_4\ \{R_4\}$. Then, one has $\{R_1 \wedge \neg B_2\}\ S_2\ \{R_2\}$, $P \wedge B_1 \Rightarrow R_2$, $\{R_2\}\ S_3\ \{R_3\}$, and $R_1 \wedge B_2 \Rightarrow R_3$.

The proof rules of (Fig. 14a) are shown in Fig. 15. Hence, the postcondition of program (a) (Fig. 14a) is $Q = R_4$. From these proof rules, the transformed program (c) (Fig. 14c) is derived as shown in Fig. 16. By applying the if-rule twice, the labels are removed and the postcondition of the transformed program becomes $Q' = R_4 = Q$. Consequently, the forward copy transformation is correct. □

$P \Rightarrow U$, $\{U\}$	S_1	$\{R_1\}$ (Fig. 8a)
$\{R_1\}$	L: S_2	$\{R_2\}$ (Fig. 8a)
$\{R_2 \wedge B\}$	S_1	$\{R_1\}$ (consequence)
$\{R_1\}$	S_2	$\{R_2\}$ (Fig. 8a)
$\{P\}$	S_1 $L : S_2$	
	do while (B)	
	S_1 S_2 **enddo**	$\{R_2 \wedge \neg B\}$

Fig. 10. Computing the postcondition of Fig. 8b.

5 Results

The algorithms have been implemented in the research parallelizing compiler FPT [12], [37] with the aim to structure programs and to extract the parallelism in ill-structured programs. Four file sets were selected, respectively, from the single-precision Linpack suite, the single-precision Lapack suite, the double-precision Eispack suite, and the single-precision Slap benchmark. Table 3 lists a statistical analysis of the benchmarks, showing the relative code expansion (excluding comments and format directives), the new control variables and the parallel loops. The average code expansion is 32 percent, which is comparable to Erosa's 22 percent code expansion obtained for a different benchmark [13]. The Linpack suite has a very modest code increase. A closer look reveals that every branch target is a *continue* statement, which is removed after restructuring. In addition, most goto's are used to implement if-then-else-endif constructs. This requires an extra

```
        do while ($B_1$)
          $S_1$
          if ($B_2$) goto $L_2$
          $S_2$
        enddo
        $S_3$
$L_2$:
                (a)
```

```
        $br_2$ = .false.
        do while ($B_1$)
          $S_1$
          if ($B_2$) {$br_2$=.true.; exit}
          $S_2$
        enddo
        if ($br_2$) goto $L_2$
        $S_3$
    $L_2$:
                (b)
```

Fig. 11. Elementry cut transformation: (a) loop with outgoing branch, (b) conversion using the **exit** statement.

$\{P \Rightarrow U\}$	**dowhile** (B_1)	$\{U \wedge B_1\}$
$\{U \wedge B_1\}$	S_1	$\{R_1\}$
$\{R_1\}$	**if** (B_2) **goto** L_2	$\{R_1 \wedge \neg B_2\}$ $\{L_2 : R_1 \wedge B_2\}$
$\{R_1 \wedge \neg B_2\}$	S_2	$\{U\}$
	enddo	$\{U \wedge \neg B_1\}$
$\{U \wedge \neg B_1\}$	S_3	$\{R_3\}$
$\{R_3 \vee R_1 \wedge B_2\}$ L_2 :		

Fig. 12. Computing the postcondition of Fig. 11a.

$\{P\}$		$br_2 = .false.$	$\{P \wedge \neg br_2\}$
$\{P \wedge \neg br_2 \Rightarrow U \wedge \neg br_2\}$		**dowhile** (B_1)	$\{U \wedge B_1 \wedge \neg br_2\}$
$\{U \wedge B_1 \wedge \neg br_2\}$		S_1	$\{R_1 \wedge \neg br_2\}$
$\{R_1 \wedge \neg br_2\}$		**if** (B_2) **then**	$\{R_1 \wedge B_2 \wedge \neg br_2\}$
$\{R_1 \wedge B_2 \wedge \neg br_2\}$		$br_2 = .true.;$ **exit**	$\{L_w : R_1 \wedge B_2 \wedge br_2\}$
		endif	$\{R_1 \wedge \neg B_2 \wedge \neg br_2\}$
$\{R_1 \wedge \neg B_2 \wedge \neg br_2\}$		S_2	$\{U \wedge \neg br_2\}$
		enddo	$\{U \wedge \neg B_1 \wedge \neg br_2\}$
$\{U \wedge \neg B_1 \wedge \neg br_2 \vee R_1 \wedge B_2 \wedge br_2\}$	L_w :	**if** (br_2) **goto** L_2	$\{U \wedge \neg B_1 \wedge \neg br_2\}$
$\{U \wedge \neg B_1 \wedge \neg br_2\}$		S_3	$\{R_3 \wedge \neg br_2\}$
$\{R_3 \wedge \neg br_2 \vee R_1 \wedge B_2 \wedge br_2\}$	L_2 :		

Fig. 13. Computing the postcondition of Fig. 11b.

jump from then then-part to the end of the if-block, which is also removed after restructuring. Since the Linpack routines contain only reducible loops and are quite structured, there is almost no code expansion. More than half of the routines has a slight code reduction, while, in 80 percent of the routines, the code expansion is limited to eight lines. A few programs contain large "computed goto's" with interfering targets, causing a lot of code duplication. The largest increase of 48 percent is for the routine `ss.for`. This routine contains a computed-goto (multijump) statement with 19 targets, pointing to overlapping code blocks. On the other hand, the code expansion is highest for the Eispack benchmark, with an average of 85.3 percent. This is generated by a handful routines with highly interacting branches and many target labels creating a maximal code increase of 5.5, while the median of the code increase is eight lines. Many Lapack routines have no goto's and were left out of the test suite. The Lapack benchmark routines used here have an average code size of 192 lines and generate a limited code increase of 26.2 percent, with a median of seven lines. The Slap benchmark has an average code size of 303 lines, with a median expansion of 11 lines.

After removing the branches, while- and repeat-loops are generated. For parallelization, these loops need to be converted into DO-loops. This conversion requires two steps: 1) the definition of an induction variable which serves as index of the loop and 2) the determination of the loop count. A technique for the conversion of while into do-loops was developed, which is described in [38]. The FPT compiler does not parallellize programs containing goto's because of the potential side-effects prohibiting the dependency analysis. After removing the goto's, normally a substantial number of parallel loops is detected. In order to estimate the impact of branch removal on the detection of additional parallelism, the parallel loops are classified into two groups. Parallelizable loops not interacting with branches are classified as "parallel loops before restructuring." Loops parallelized after branch removal are called "parallel loops after restructuring." The Table 3 shows that the present restructuring technique allows between 10 percent and 30 percent extra loops to be parallelized in the Linpack, Eispack, and Lapack routines. Slap is already better structured and, therefore, the gain is only 2 percent.

Code size increase can reduce performance, e.g., loop unrolling is known to increase cache misses and register pressure which may offset the benefit of the transformation. This is especially important when the amount of memory available is limited, e.g., in embedded systems. However, in most cases, the code size increase is limited and the parallelism outweighs the code size penalty. In extreme cases, rather than reverting to unstructured code, new alternatives at the link level offer efficient and uniform code compaction techniques [10].

The structuring of ill-structured programs is recognized as a necessary condition for many compiler optimizations, especially for the exploitation of instruction level parallelism [19], vectorization [2], predicated instruction scheduling [8], contemporary VLIW architectures [32], and the

	if (B_1) **goto** L_1		**if** (B_1) **then**	**if** (B_1) **then**
	S_1		S_3	S_3
	if (B_2) **goto** L_2		**else**	**else**
	S_2		S_1	S_1
L_1:	S_3		**if** (B_2) **goto** L_2	**if** $(\neg B_2)$ **then**
L_2:	S_4		S_2	S_2
			S_3	S_3
		L_2 :		**endif**
			endif	**endif**
			S_4	S_4
	(a)		(b)	(c)

Fig. 14. Elementry forward copy transformation: (a) interacting forward branches, (b) applying the forward copy eliminates the initial forward branch, (c) simple conversion of the remaining branch.

$\{P\}$		**if** (B_1) **goto** L_1	$\{P \wedge \neg B_1\}$ $\{L_1 : P \wedge B_1 \Rightarrow R_2\}$
$\{P \wedge \neg B_1\}$		S_1	$\{R_1\}$
$\{R_1\}$		**if** (B_2) **goto** L_2	$\{R_1 \wedge \neg B_2\}$ $\{L_2 : R_1 \wedge B_2 \Rightarrow R_3\}$
$\{R_1 \wedge \neg B_2\}$		S_2	$\{R_2\}$
$\{R_2\}$	L_1 :	S_3	$\{R_3\}$
$\{R_3\}$	L_2 :	S_4	$\{R_4\}$

Fig. 15. Computing the postcondition of Fig. 14a.

$P \wedge B_1 \Rightarrow R_2$, $\{R_2\}$		S_3	$\{R_3\}$ (Fig. 14a)
$\{P \wedge \neg B_1\}$		S_1	$\{R_1\}$ (Fig. 14a)
$\{R_1\}$		**if** (B_2) **goto** L_2	$\{R_1 \wedge \neg B_2\}$ $\{L_2 : R_1 \wedge B_2 \Rightarrow R_3\}$ (Fig. 14a)
$\{R_1 \wedge \neg B_2\}$		S_2	$\{R_2\}$ (Fig. 14a)
$\{R_2\}$		S_3	$\{R_3\}$ (Fig. 14a)
$\{R_3\}$	L_2 :	S_4	$\{R_4\}$ (Fig. 14a)
$\{P\}$		**if** (B_1) **then**	(apply the if-rule)
		S_3	
		else	
		S_1	
		if $(\neg B_2)$ **then**	(apply the if-rule)
		S_2 S_3	$\{R_3\}$
		endif	
		endif	
$\{R_3\}$	L_2 :	S_4	$\{R_4\}$

Fig. 16. Computing the postcondition of Fig. 14c.

handling of control flow in general. In this respect, the results of our hammock graph restructuring technique has a number of characteristics which compare favorably with other approaches. This comparison is elaborated in the section about related work.

6 Related Work

Structuring programs has been an active research topic for many years, with shifting methods and objectives, see Table 4. In the earlier papers by Boehm et al. [7], Peterson et al. [29], Ashcroft and Manna [5], and Williams [34], [36], a number of techniques were introduced based on flowcharts. Boehm et al. [7] define a set of functional and predicative boxes, similar to sequential and control statements and show that any flowchart (a two-dimensional programming language) can be normalized by three basic constructs, composition, selection and iteration, in such a way that all loops are properly nested. The results are applied to the theory of Turing machines, but no algorithm for ordinary programs is given. Ashcroft and Manna [5] introduce two algorithms to remove goto's in a flowchart using while statements. The first algorithm (not described in the paper) duplicates code, the second algorithm tries to find cut-sets within the flow graph, to eliminate the duplication using extra logical variables. The hammock graph approach uses selectively repeat and while loops, leading to no code replication and no extra logical variables in the example flow charts of [3]. Peterson et al. [29] have shown that for any given flowchart, a program can be written using if-statements, repeat statements, and multilevel exit statements. The chart may have to be modified by node splitting. A "well-formed program" is defined as one in which loops and conditional statements are properly nested and entered only at their beginning. Starting from a flowchart, an algorithm is given which transforms a flowchart into a well-formed flowchart. A second algorithm produces a well-formed program from a well-formed flowchart. Besides starting from a flowchart, the difference with our approach is that multilevel exits are actually goto's jumping to the

TABLE 3
Restructuring Four Benchmark Suites

	Linpack	Lapack	Eispack	Slap	Total
Lines B	7379	24546	4557	3039	39521
Lines A	8255	30980	8444	4481	52160
Lines Incr.	11.9 %	26.2 %	85.3 %	47.4 %	32.0 %
Gotos B	714	563	602	71	1950
Par. loops B	322	476	323	91	1212
Par. loops A	355	561	419	93	1428
Par. loops Incr.	33	85	96	2	216
Par. loops Incr. %	10.2 %	17.9 %	29.7 %	2.2 %	17.8 %
New vars	7	161	150	32	350
Files	60	128	67	10	265

B=before, A=after, Incr. = increment, Par. = parallel loops, Par. loops Incr. = unstructured loops parallelized after restructuring, New vars = new logical variables.

TABLE 4
Chronological List of Restructuring Methods

Input	Approach	IMP
Turing machine [7]	composition, iteration	
Flowchart [5]	while statements	
Flowchart [29]	repeat, multi-level exits, RP	
Flow diagram [36]	5 unstructured base diagrams	
Program [2]	if-conversion	PFC
Schema [26]	6 basic unstructured forms	
Program [30]	multiple exit statements, RP	
Program [3]	factoring continuations	MIPRAC
Program [13]	goto relocation and elimination	Mc-CAT
Binary [9]	interval analysis	dcc
Program (this paper)	nested hammock graphs	FPT

CDG = control dependence graph, RP = reducible program only, IMP = compiler implementation.

start of an arbitrary enclosed loop. This may confuse the reader or complicate the dependence analysis, and is indeed considered a cause of unstructuredness in [36]. Furthermore, very few languages support the multilevel exit construct. Williams and Ossher [36] state that a structured flow diagram consists of three base diagrams: the simple sequence, the selection (if-then-else), and the while loop. Unstructuredness is identified by five flow graph structures, one structure of interacting forward branches called abnormal selection path, and four loops structures, denoted respectively, multiple entry, multiple exit, overlapping, and parallel loops. A detailed algorithm is given to find the loop and selection structures and substitute the anomalous structures by structured ones. This mechanical restructuring doesn't always produce the most efficient or most pleasing solution. In particular, the bare flowchart of a program looses information about the structured loops and if-statements available in the source. In a later paper [35], goto's are removed from a Pascal program, taking advantage of the limitations of the use of goto's in this language and considering three basic sets of transformations. Besides being related to a particular programming language, the examples of the paper use more temporary variables (flags) than in our approach. Furthermore, no benchmark results such as code expansion are given. The if-conversion technique developed by Allen et al. [2] is aimed at vectorizing Fortran loops. The idea is to remove control statements and replace them by guarded Fortran 8X vector instructions, e.g., `WHERE (BR1(1:N)) A(1:N) = B(1:N) + 10`. A number of transformations on branches are applied. Branch relocation moves branches out of loops until the branch and its target are at the same loop nesting level; in this way, they become a forward or backward branch. Branch removal eliminates forward branches by computing guard expressions for statements under their control and conditioning execution on these expressions. Backward branches are left in place because they imply loops which cannot be expressed by guarded statements. If-conversion has two limitations: First, it is only oriented towards vectorization, and second, backward branches representing loops with potential parallelism are not converted. In our approach, we follow a similar classification of branches into forward, backward, and exit branches; however, the control restructuring is done at the hammock graph level, instead of the guarded statement level. This allows a coarse grain parallelization as well as vectorization. In addition, backward branches are converted into loops amenable to parallelization. Ramshaw [30] wants to eliminate goto's while maintaining the program structure. He proves that removing goto's without code duplication requires a reducible program. Based on a stricter set of ground rules than Peterson et al. [29], goto's are eliminated for a conforming program. However, the stricter rules imply that a major cause of unstructured loops, i.e., irreducible loops with multiple entries, is unaccounted for.

A notably different approach is the use of continuations as presented by Ammarguellat [3]. Informally, a continuation is the remaining execution trajectory at any particular point in the program. Ammarguellat uses continuations as unknowns at the branch points in the program. This results in a set of continuation equations which can be solved recursively in a Gaussian elimination style. When the solution is found, code factorization allows to replace multiple identical continuation codes executable under different execution conditions by a single continuation code, executed under a generalized condition. This factoring allows to avoid all code replication, except for the cases of irreducible graphs. The approach by Ammarguellat is quite elegant and feasible, but rather complex compared to the approach presented here. Moreover, by factoring the continuation code, the execution conditions become complex and the associated overhead might become noticeable, especially in small inner loops. In contrast, our technique may create a relatively small code replication, but the branch expressions remain essentially the same as in the original program. Furthermore, the goal to avoid code replication generates complex predicate expressions because the same code, executed under different conditions, will appear only once in the program. While this indeed creates a more compact code, the execution performance can be hampered by a lengthy predicate analysis. Consider the following example from Ammarguellat (Fig. 17a). The Lisp program is rewritten into Fortran, in order to use our FPT compiler, and the original condition `(i.gt.0)` is replaced by `(i.gt.5)`, to avoid dead code between labels 20 and 40. Ammarguellat's normalized flow program (Fig. 17b) contains a double repeat loop, three predicates, and seven conditional expressions representing in total 12 logical operations. In contrast, the hammock graph transformation generates two loops, two predicates, four conditional expressions, and three logical operations in total. In addition, instead of the outer repeat loop in Fig. 17b, a do while loop without predicates in the loop condition is generated (see Fig. 17c). The simple loop condition, derived from the if-statement, allows the loop to be converted into a parallel do-loop Fig. 17d, using induction variable detection and dependence analysis [38]. This example illustrates that reducing and simplifying branch conditions enhances the granularity and may allow more optimization, an observation already made by Allen et al. [2].

Oulsnam [25] has an interesting paper in which he uses Kleene's algebra to describe an unstructured flow graph and reduce it to a structured form represented by structured regular expressions. He extended Brainbridge's reduction rules, by introducing logical variables to remember the values of branch conditions in the unstructured flowgraph. In this way, the flowgraph can be structured by applying the reduction rules to a system of so-called "endset" equations. No algorithm is given, but there is a follow-up paper on the algorithmic transformation of schemes [26] and the regular expression paradigm was recently revisited [23]. The problem with this approach is that a lot of extra logical variables are introduced, which unnecessarily increase the loop nesting level and lower the granularity of the resulting program. As an example, consider Oulsnam's treatment of the generalized Flynn's problem No. 5 (Fig. 7 in [25]). For the comparison, the flowchart is

```
    i=1
    j=i
10  if(i.gt.5) goto 40
20  x=i
30  i=i+1
    if(x.gt.i) then
      if(x.gt.j) goto 20
      goto 10
    else
      goto 50
    endif
40  j=1
50  j=2
```

(a)

```
i=1
j=i
repeat
   pred1=i.gt.5
   if(.not. pred1)
   repeat
      x=i
      i=i+1
      pred2=x.gt.i
      if(pred2) then
         pred3=x.gt.j
      endif
   until (pred2 .and. pred3)
until(.not.pred1.and.pred2.and.not.pred3)
if(pred2.and.(pred1.or..not.pred3)) then
    j=1
endif
j=2
```

(b)

```
i=1
j=i
DO WHILE (i.le.5)
  REPEAT
    x=i
    i=i+1
    IF (x.le.i) THEN
      br1=.True.
      EXIT
    ENDIF
  UNTIL (x.le.j)
  IF (br1) THEN
    br2=.True.
    EXIT
  ENDIF
ENDDO
IF (.not.br2) THEN
  j=1
ENDIF
j=2
```

(c)

```
i=1
j=i
DOALL k1 = 1,5
  i=k1
  REPEAT
    x=i
    i=k1+1
    IF (x.le.i) THEN
      br1=.True.
      EXIT
    ENDIF
  UNTIL (x.le.j)
  IF (br1) THEN
    br2=.True.
    EXIT
  ENDIF
ENDDO
IF (.not.br2) THEN
  j=1
ENDIF
j=2
```

(d)

Fig. 17. (a) Program with three branches, (b) minimizing code replication using predicates [3], (c) limited code replication using hammock graph restructuring reduces predicate proliferation, and (d) allows induction variable detection and parallel loop conversion.

rephrased in a Fortran style, with case sensitive variable names. From left to right one has the original program, the code derived from Oulsnam's flowchart and the code generated by our transformer, FPT, see Fig. 18. The unstructured program contains 16 lines; the programs restructured by regular expression reduction and by hammock graph transformations, respectively, contain 33 and 43 lines. Although Oulsnam's code is 24 percent shorter, it is not necessarily more efficient. This is because the control structure is more deeply nested and requires more conditions to be checked before executing code. For example, the maximum nesting depth of the different assignment statements is given in Table 5.

A large nesting depth means more jumps and less efficient pipelining, especially in loops. Moreover, the basic blocks are smaller, e.g., the average size of an execution block is 1.18 statements in Oulsnam's version and 1.85 statements in the hammock graph transformation. A large granularity is necessary to extract parallelism for wide issue architectures [8].

```
1   a=1
    if(.not.p) goto 33
    f=1
2   b=1
    if(q) goto 11
    g=1
3   c=1
    if(r) goto 22
    stop
11  d=1
    goto 1
22  h=1
    goto 2
33  e=1
    goto 3
    end
```

(a)

```
LOGICAL p,q,r,P,Q,R
Q=.True.
REPEAT
    REPEAT
        IF(Q) THEN
            a=1
            P=p
            IF(P) THEN
                f=1
            ELSE
                e=1
            ENDIF
        ELSE
            h=1
        ENDIF
        IF(P) THEN
            b=1
            Q=q
            IF(.not.Q) THEN
                g=1
            ELSE
                d=1
            ENDIF
        ELSE
            Q=.False.
        ENDIF
    UNTIL(.not.Q)
    c=1
    IF(r) THEN
        Q=.False.
    ENDIF
UNTIL(.not.r)
END
```

(b)

```
LOGICAL p,q,r,br1
br1=.False.
a=1
IF (p) THEN
    f=1
    b=1
    DO WHILE (q)
        d=1
        a=1
        IF (.not.p) THEN
            e=1
            br1=.True.
            EXIT
        ENDIF
        f=1
        b=1
    ENDDO
    IF (.not.br1) THEN
        g=1
    ENDIF
ENDIF
c=1
DO WHILE (r)
    h=1
    b=1
    DO WHILE (q)
        d=1
        a=1
        IF (.not.p) THEN
            e=1
            br1=.True.
            EXIT
        ENDIF
        f=1
        b=1
    ENDDO
    IF (.not.br1) THEN
        g=1
    ENDIF
    c=1
ENDDO
END
```

(c)

Fig. 18. (a) example from Oulsnam [25], (b) code restructuring using Kleene's algebra, and (c) Hammock graph restructuring.

TABLE 5
Maximum Nesting Depth with Two Restructuring Approaches

Max. nesting depth	a=1	b=1	c=1	d=1	e=1	f=1	g=1	average
Regular expressions [25]	3	3	1	4	4	4	4	3.28
Hammock graphs	2	2	1	2	3	2	2	2.00

With respect to implementation, Allen and Ammarguellat have implemented the branch removal into a Fortran compiler, whereas Eroasa and Hendren [13] describe an implementation for a C-compiler. Erosa and Hendren follow basically the same branch classification as Allen, but offer a different implementation for the exit-jumps: They are moved in or out of the loop or if-block such that branch and target are at the same nesting level. Incoming jumps are cut into hops, and a conditional jump is placed at the *start* of each block. Whereas the treatment of exit jumps is the same as our approach, we treat the problem of irreducibility differently: Loops and blocks with incoming jumps are immediately converted into a set of interacting branches. This allows the backward and the forward

```
      jmp @5
@4:
      jmp @9
@8:
      jne @19
      jmp @10
@19:
      jmp @14
@13:@14:
      jg  @13
@9:@10:
      jge @20
      jmp @8
@5:@20:
      jge @21
      jmp @4
@21:
```

(a)

```
si=0
DO WHILE (si.lt.n)
  di=0
    DO WHILE (di.lt.n)
      IF (di.ne.si) THEN
        f=a(si,di)/a(si,si)
        k=si+1
        DO WHILE (k.lt.n+1)
          a(di,k)=a(di,k)-f*a(si,k)
          k=k+1
        ENDDO
      ENDIF
      di=di+1
    ENDDO
    si=si+1
ENDDO
```

(b)

```
si=0
DO k1 = 0,n-1
  di=0
  DOALL k2 = 0,n-1
    IF (k2.ne.k1) THEN
      f=a(k1,k2)/a(k1,k1)
      DOALL k=k1+1,n
        a(k2,k)=a(k2,k)-f*a(k1,k)
      ENDDO
    ENDIF
    di=k2+1
  ENDDO
  si=k1+1
ENDDO
```

(c)

Fig. 19. Decompiling and parallelizing an assembler program. (a) Branches and labels from the assembler listing of the Gauss-Jordan elimination. (b) After restructuring with this paper's algorithm, the backward branches become while loops. (c) By removing the induction variables two while loops are converted into parallel loops.

transformations to eliminate the irreducibility without the need for extra control variables.

A recent application area is structuring decompiled binary programs. Cifuentes [9] developed the decompiler dcc, able to convert 8086 executable code into a C program, with a limited number of goto's. To illustrate the use of hammock graph restructuring in this area, we take the assembly output from the Gauss-Jordan linear system solver, generated by Turbo-C. The elimination loop contains 10 branches and 15 labels, which are depicted in Fig. 19a (other code is omitted). After rewriting the program in Fortran, taking into account the array dimensions and applying the hammock graph restructuring with FPT, the program in Fig. 19b is obtained in which the goto's are removed and the backward jumps become while loops. In order to parallelize the nested loop, two further steps are needed: the detection of an induction variable and a data dependence analysis. Using the technique described in [37], [38], FPT finds two parallel DOALL-loops, see Fig. 19c.

Except for the algebraic treatment of continuations by Ammarguellat, none of the methods to remove branches have been formally proven. This is quite understandable since a formal proof for all but simple programs has been recognized to be quite difficult [17]. On the other hand, the quality of the compiler, as ubiquitous translation layer to steer the machine hardware, is of utmost importance. Fortunately, the painstaking effort to devise a rigorous proof can now be alleviated with the advent of increasingly capable automatic theorem provers. PVS, the prototype verification system of SRI [28], has been successfully used to verify the correctness of program transformations [31], mathematical theorems [6] as well as live-critical aviation software [27]. In our approach of program restructuring, the actual program transformation is carried out by three basic transformations. We were able to prove that the transformations are correct and the correctness proofs were verified using PVS [11]. The proof uses Arbib et al.'s extension of the axiomatic rules for goto's [4].

7 Conclusion

Structured programs are necessary for readability and for effective compiler optimizations. In this paper, simple, efficient and provable correct basic program transformations are presented to remove all branches in a program. A restructuring algorithm using the three elementary transformations converts the control flow graph of a program into a nest of hammock graphs.

From the comparison with related work it becomes apparent that there is a trade off between conditional expression complexity, code granularity and code replication. The transformations presented allow a limited code replication in order to keep the sequential blocks intact and to minimize the pressure on predicate resources and branch prediction logic. The effect has been measured on several benchmarks, using an implementation of the hammock graph transformations in the research parallelizing compiler FPT [37].

References

[1] S. Alagic and M.A. Arbib, *The Design of Well-Structured and Correct Programs.* Springer, 1978.

[2] J.-R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," *Proc. 10th Ann. ACM Symp. Principles of Programming Languages,* pp. 177-189, Jan. 1983.

[3] Z. Ammarguellat, "A Control-Flow Normalization Algorithm and Its Complexity," *IEEE Trans. Software Eng.,* vol. 18, no. 3, pp. 237-251, 1992.

[4] M.A. Arbib and S. Alagic, "Proof Rules for Gotos," *Acta Informatica,* vol. 11, pp. 139-148, 1979.

[5] E. Ashcroft and Z. Manna, "The Translation of Goto Programs into While Programs," *Proc. IFIP Congress 71,* C. Freiman, J. Griffith, and J. Rosenfeld, eds., vol. 1, pp. 250-255, 1972.

[6] B. Dutertre, "Elements of Mathematical Analysis in PVS," *Proc. Ninth Int'l Conf. Theorem Proving in Higher Order Logics TPHOL,* J. Von Wright, J. Grundy, and J. Harrison, eds., Aug. 1996.

[7] C. Boehm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Comm. ACM,* vol. 9, no. 5, pp. 366-371, May 1966.

[8] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante, "Path Analysis and Renaming for Predicated Instruction Scheduling," *Int'l J. Parallel Programming,* vol. 28, no. 6, pp. 563-588, 2000.

[9] C. Cifuentes, "Structuring Decompiled Graphs," *Lecture Notes in Computer Science,* vol. 1060, pp. 91-104, 1996.

[10] B. De Bus, D. Kästner, D. Chanet, L. Van Put, and B. De Sutter, "Post-Pass Compaction Techniques," *Comm. ACM,* vol. 46, no. 8, pp. 41-46, 2003.

[11] E.H. D'Hollander and F. Zhang, "Restructuring Program Transformations with Proof Verification Using PVS," Technical Report RUG/ELIS01, p. 47, 2001.

[12] E.H. D'Hollander, F. Zhang, and Q. Wang, "The Fortran Parallel Transformer and Its Programming Environment," *Information Sciences,* vol. 106, nos. 3-4, pp. 293-317, 1998.

[13] A.M. Erosa and L.J. Hendren, "Taming Control Flow: A Structured Approach to Eliminating Goto Statements," *Proc. Fifth Int'l Conf. Computer Languages,* pp. 229-240, 1994.

[14] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Programming Languages and Systems,* vol. 9, no. 3, pp. 319-349, 1987.

[15] P. Havlak, "Nesting of Reducible and Irreducible Loops," *ACM Trans. Programming Languages and Systems (TOPLAS),* vol. 19, no. 4, pp. 557-567, 1997.

[16] M.S. Hecht and J.D. Ullman, "Characterizations of Reducible Flow Graphs," *J. ACM,* vol. 21, no. 3, pp. 367-375, 1974.

[17] C.A.R. Hoare, "An Axiomatic Basis of Computer Programming," *Comm. ACM,* vol. 12, pp. 576-580, 1969.

[18] C.A.R. Hoare, "An Axiomatic Definition of the Programming Language Pascal," *Acta Informatca,* vol. 2, pp. 335-355, 1973.

[19] J. Janssen and H. Corporaal, "Making Graphs Reducible with Controlled Node Splitting," *ACM Trans. Programming Languages and Systems,* vol. 19, no. 6, pp. 1031-1052, 1997.

[20] V.N. Kas'janov, "Distinguishing Hammocks in a Directed Graph," *Soviet Math. Doklady,* vol. 16, no. 5, pp. 448-450, 1975.

[21] A. Klauser, T. Austin, D. Grunwald, and B. Calder, "Dynamic Hammock Predication for Non-Predicated Instruction Set Architectures," *Proc. 1998 Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '98),* pp. 278-285, Oct. 1998.

[22] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. Computers,* vol. 28, no. 9, pp. 690-691, 1979.

[23] P. Morris, R.A. Gray, and R.E. Filman, "GOTO Removal Based on Regular Expressions," *J. Software Maintenance—Research and Practice,* vol. 9, no. 1, pp. 47-66, Jan./Feb. 1997.

[24] H.R. Nielson and F. Nielson, *Semantics with Applications.* John Wiley and Sons, 1993.

[25] G. Oulsnam, "Unravelling Unstructured Programs," *The Computer J.,* vol. 25, no. 3, pp. 379-387, Aug. 1982.

[26] G. Oulsnam, "The Algorithmic Transformation of Schemas to Structured Form," *The Computer J.,* vol. 30, no. 1, pp. 43-51, Feb. 1987.

[27] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Trans. Software Eng.,* vol. 21, no. 2, pp. 107-125, Feb. 1995.

[28] S. Owre, J.M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," *Proc. 11th Int'l Conf. Automated Deduction (CADE-11),* D. Kapur, ed., June 1992.

[29] W.W. Peterson, T. Kasami, and N. Tokura, "On the Capabilities of While, Repeat, and Exit Statements," *Comm. ACM,* vol. 16, no. 8, pp. 503-512, Aug. 1973.

[30] L. Ramshaw, "Eliminating Goto's While Preserving Program Structure," *J. ACM,* vol. 35, no. 4, pp. 893-920, Oct. 1988.

[31] N. Shankar, "Steps Towards Mechanizing Program Transformations Using PVS," *Science of Computer Programming,* vol. 26, nos. 1-3, pp. 33-57, May 1996.

[32] S. Unger and F. Mueller, "Handling Irreducible Loops: Optimized Node Splitting Versus DJ-Graphs," *ACM Trans. Programming Languages and Systems (TOPLAS),* vol. 24, no. 4, pp. 299-333, 2002.

[33] M. Weiser, "Program Slicing," *IEEE Trans. Software Eng.,* vol. 10, no. 4, pp. 352-357, July 1984.

[34] M.H. Williams, "Generating Structured Flow Diagrams: The Nature of Unstructuredness," *The Computer J.,* vol. 20, no. 1, pp. 45-50, Feb. 1977.

[35] M.H. Williams and G. Chen, "Restructuring Pascal Programs Containing Goto Statements," *The Computer J.,* vol. 28, no. 2, pp. 134-137, 1985.

[36] M.H. Williams and H.L. Ossher, "Conversion of Unstructured Flow Diagrams to Structured Form," *The Computer J.,* vol. 21, no. 2, pp. 161-167, 1978.

[37] F. Zhang, "FPT: A Parallel Programming Environment," PhD thesis, Dept. of Electrical Eng., Univ. of Ghent, 1996.

[38] F. Zhang and E.H. D'Hollander, "Extracting the Parallelism in Programs with Unstructured Control Statements," *Proc. Int'l Conf. Parallel and Distributed Systems, (ICPADS '94),* pp. 264-270, Dec. 1994.

Fubo Zhang graduated from Fudan University of Shanghai (1986). He received the PhD degree from the University of Ghent in 1996. He studied in the Electronic Engineering Department of Gent University, Belgium, majoring in parallel processing and distributed computer systems (1991-1996). After graduating from the University of Ghent, he worked at Platform Computing Canada as a product architect and development team manager (1996-2001). In 2001, he relocated to Beijing, China, and worked in the Platform Beijing office as a senior professional service and support manager for A/P. His current focus is on technical computing grid technology and distributed computing resource management.

Erik H. D'Hollander graduated from the universities of Ghent, (electrical engineering) in 1972 and Leuven (computer science) in 1976. He did research on parallel processing at the Computer Science Department of the University of California at Los Angeles (1979-1980) and received the PhD degree from the University of Ghent, Belgium in 1980. In the summers of 1983 and 1985, he was visiting researcher, respectively, at the Department of Computer Science of UCLA and at the Center of Supercomputing Research and Development (CSRD) of the University of Illinois at Urbana-Champaign. His research interests include automatic parallelization of ordinary programs, performance computing and compiler techniques for explicitly parallel computing, multimedia and embedded system architectures. He is a member of the IEEE.