*Research Monographs in*
*Parallel and Distributed Computing*

DAVID GELERNTER, ALEXANDRU NICOLAU
and DAVID PADUA (Editors)

# Languages and Compilers for Parallel Computing

The MIT Press

RESEARCH MONOGRAPHS IN PARALLEL AND DISTRIBUTED COMPUTING

Edited by
**David Gelernter**, Yale University,
**Alexandru Nicolau**, University of California, Irvine, and
**David Padua**, University of Illinois at Urbana-Champaign

# Languages and Compilers for Parallel Computing

Pitman, London

The MIT Press, Cambridge, Massachusetts

PITMAN PUBLISHING
128 Long Acre, London WC2E 9AN

First published 1990

Available in the Western Hemisphere and Israel from
The MIT Press
Cambridge, Massachusetts (and London, England)

ISSN 0953-7767

**British Library Cataloguing in Publication Data**
Languages and compilers for parallel computing.—(Research
monographs in parallel and distributed computing,
ISSN 0953-7767)
1. Computer systems. Parallel-processor
systems. Programming languages
I. Gelernter, David II. Nicolau, Alexandru III. Padua,
David IV. Series
005.4'5

ISBN 0-273-08820-3

**Library of Congress Cataloging-in-Publication Data**
Languages and compilers for parallel computing / edited by David
Gelernter, Alexandru Nicolau, and David Padua.
p. cm.—(Research monographs in parallel and distributed
computing, ISSN 0953-7767)
Includes bibliographical references.
ISBN 0-262-57080-7
1. Programming languages (Electronic computers) 2. Compilers
(Computer programs) 3. Parallel processing (Electronic computers)
I. Gelernter, David Hillel. II. Nicolau, Alexandru. III. Padua,
David A. IV. Series.
QA76.7.L38 1990
0054'53—dc20

Reproduced and printed by photolithography
in Great Britain by Biddles Ltd, Guildford

# Contents

1 **Fine-grain Parallelization and the Wavefront Method** 1
Alexander Aiken, *IBM Almaden Research Center*
Alexandru Nicolau, *University of California, Irvine*

2 **Visual Extensions to Parallel Programming Languages** 17
Duane A. Bailey, *Williams College*
Janice E. Cuny, *University of Massachusetts*

3 **A Status Report on CONSUL** 37
Douglas Baldwin, *University of Rochester*

4 **A Theory of Loop Permutations** 54
Utpal Banerjee, *Intel Corporation*

5 **Software Engineering of Parallel Programs in a Computationally Oriented Display Environment** 75
James C. Browne, *University of Texas at Austin*

6 **A Future-based Parallel Language for a General-purpose Highly-parallel Computer** 95
David Callahan, *Tera Computer Company*
Burton Smith, *Tera Computer Company*

7 **Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler** 114
Nicholas Carriero, *Yale University*
David Gelernter, *Yale University*

8 **COOL: a Language for Parallel Programming** 126
Rohit Chandra, *Stanford University*
Anoop Gupta, *Stanford University*
John L. Hennessy, *Stanford University*

9 **Code Generation and Separate Compilation in a Parallel Program Debugger** 149
Jong-Deok Choi, *University of Wisconsin, Madison*
Barton P. Miller, *University of Wisconsin, Madison*

10 **Blackboard Programming in Shared Prolog** 170
Paolo Ciancarini, *Universita di Pisa*

11 **Experiences U**
Ron Cytron, *I*
Jeanne Ferran
Vivek Sarkar,

12 **A New Compi**
**Branches on a**
Kemal Ebciog
Toshio Nakata

13 **Compile Time**
Christine Eise
William Jalby
Alain Lichnew

14 **Static Analysis**
Dennis Ganno
Vincent A. Gu
Jenq Kuen Le

15 **Compiling for**
Guang R. Gao
Zaharias Para

16 **A Comparison**
**Boyer–Moore**
Williams Ludw
Zahira Amma

17 **Refined C: an**
David Klapph
Apostolos D.
Xiangyun Kor

18 **Improving Par**
Andrew W. K
Lubomir Bic,
Daniel D. Gaj

19 **Some Results**
Zhiyuan Li, *U*
Pen-Chung Ye

11 **Experiences Using Control Dependence in PTRAN** 186
Ron Cytron, *IBM T. J. Watson Research Center*
Jeanne Ferrante, *IBM T. J. Watson Research Center*
Vivek Sarkar, *IBM T. J. Watson Research Center*

12 **A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture** 213
Kemal Ebcioglu, *IBM T. J. Watson Research Center*
Toshio Nakatani, *IBM Tokyo Research Laboratory*

13 **Compile Time Optimization of Memory and Register Usage on the Cray 2** 230
Christine Eisenbeis, *INRIA, Paris*
William Jalby, *INRIA and Universite Rennes*
Alain Lichnewsky, *INRIA, Paris*

14 **Static Analysis and Runtime Support for Parallel Execution of C** 254
Dennis Gannon, *Indiana University*
Vincent A. Guarna Jr., *University of Illinois at Urbana-Champaign*
Jenq Kuen Lee, *Indiana University*

15 **Compiling for Dataflow Software Pipelining** 275
Guang R. Gao, *McGill University*
Zaharias Paraskevas, *McGill University*

16 **A Comparison of Automatic versus Manual Parallelization of the Boyer–Moore Theorem Prover** 307
Williams Ludwell Harrison III, *University of Illinois at Urbana-Champaign*
Zahira Ammarguellat, *University of Illinois at Urbana-Champaign*

17 **Refined C: an Update** 331
David Klappholz, *Stevens Institute of Technology*
Apostolos D. Kallis, *Stevens Institute of Technology*
Xiangyun Kong, *Stevens Institute of Technology*

18 **Improving Parallel Program Performance Using Critical Path Analysis** 358
Andrew W. Kwan, *University of California, Irvine*
Lubomir Bic, *University of California, Irvine*
Daniel D. Gajski, *University of California, Irvine*

19 **Some Results on Exact Data Dependence Analysis** 374
Zhiyuan Li, *University of Illinois at Urbana-Champaign*
Pen-Chung Yew, *University of Illinois at Urbana-Champaign*

20 **Compiling Programs with User Parallelism** 402
Samuel P. Midkiff, *University of Illinois at Urbana-Champaign*
David A. Padua, *University of Illinois at Urbana-Champaign*
Ron Cytron, *IBM T. J. Watson Research Center*

21 **The Structure of Parafrase-2: an Advanced Parallelizing Compiler for C and Fortran** 423
Constantine D. Polychronopoulos
Milind B. Girkar
Mohammad R. Haghighat
Chia L. Lee
Bruce P. Leung
Dale A. Schouten
*all at University of Illinois at Urbana-Champaign*

22 **Concurrentization: Mapping Parallel Loops onto a Cluster of Control Data's Advanced Parallel Processors** 454
Tom Rochat, *Control Data Corporation*

23 **The XYZ Abstraction Levels of Poker-like Languages** 470
Lawrence Snyder, *University of Washington*

24 **The PARSEQ Project: an Interim Report** 490
Jon A. Solworth, *University of Illinois at Chicago*

25 **Desirable Code Transformations for a Concurrent Machine** 511
Augustus K. Uht, *University of California, San Diego*

26 **Loop Rotation** 531
Michael Wolfe, *Oregon Graduate Center*

# 12 A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture

**Kemal Ebcioglu and Toshio Nakatani**

**Abstract**

*Enhanced pipeline-percolation scheduling* is a new compilation technique we developed to extract fine-grain parallelism uniformly from nested loops in general software. The compilation technique generates code for execution on the IBM Very Long Instruction Word (VLIW) machine, which is now being built at the IBM T. J. Watson Research Center. The IBM VLIW architecture has features that facilitate high performance on general sequential-natured code with unpredictable branches: a decision-tree shaped instruction capable of performing multiple conditional branches emanating from arbitrary paths in the sequential code, a conditional execution feature that reduces pathlengths, multiple functional units and multiple ports to data memory, and a shared register file which eliminates communication delays between functional units. The preliminary version of the parallelizing compiler for the IBM VLIW machine is now operational both at the IBM Tokyo Research Laboratory and the IBM T. J. Watson Research Center. It is shown that a pathlength reduction of 5-11X over a RISC processor can be achieved on some of the Stanford integer benchmarks and other sequential-natured C programs.

## 1. Introduction

While vector supercomputers and multiprocessors are capable of achieving very high performance on vectorizable or inherently parallel, scientific code fragments, a considerable portion of typical, existing programs may not be vectorizable, and may not have the coarse-grain parallelism to allow separation into different tasks for execution on different processors. Typical sequential-natured, non-numerical programs do have a degree of fine-grain parallelism; but realistic multiprocessor architectures cannot exploit such parallelism, because communication and synchronization delays can greatly reduce the speedup.

The Very Long Instruction Word (VLIW) approach has a potential to fill this gap, and may achieve considerable speed-up on general, sequential-natured programs uniformly. But the application domain of VLIW architectures has so far been restricted to code with predictable branches (mainly scientific code), because of the nature of the parallelism extraction techniques developed to date for such architectures, and because of the assumptions made about the predictability of conditional branches during the design of such machines ([7] and [8]).

*Enhanced pipeline-percolation scheduling* is a new compilation technique we developed to extract fine-grain parallelism uniformly from general nested loops with arbitrary unpredictable branches. This technique generates code for the IBM VLIW machine, which is now being built at the IBM T. J. Watson Research Center. The IBM VLIW architecture is intended to achieve good performance on a very wide range of sequential-natured programs, such as systems, commercial, or A.I. code.

To facilitate high performance on general code, the architecture has features such as decision-tree shaped instructions capable of performing multiple conditional branches emanating from arbitrary paths in the sequential code, a conditional execution feature that reduces pathlengths, multiple functional units and multiple ports to data memory, and a shared register file which eliminates communication delays between functional units.

Our new parallelism extraction technique is robust, since it does not require any restriction to the form of the loops, and avoids the branch prediction problem by not doing any prediction, i.e. by executing operations on all paths instead, whenever resources permit.

The preliminary version of the parallelizing compiler for the IBM VLIW machine that takes the sequential intermediate code of a standard optimizing high-level language compiler ([15]) as input, and produces compacted long instruction words, is now operational both at the IBM Tokyo Research Laboratory and the IBM T. J. Watson Research Center. It is shown that the average of 5-11X pathlength reduction is obtained (versus a RISC Processor) on some of the Stanford integer benchmarks and other sequential-natured C programs.

In this paper, we will first give an overview of our architecture and the related compilation techniques. Second, the *enhanced pipeline-percolation scheduling technique* will be described in some detail, and will be compared with previous compiler research in this area. Finally, some benchmark results will be summarized to show the effectiveness of

the new paralleliza

## 2. Overview c

Our machine has n
condition code reg
branch and conditi
The instruction
tree is encoded in
tree there are label
the tree, there is a
code registers). C
register arithmetic
single machine cyc
selection phase an
At the path se
the tree to a tip n
that were set in th
given node is true,
to the right.
At the executic
that are on the sel
from the previous
taneously since th
sets the same des
node determines t
Finally the re
branches to the in
current instructio
For the specifi
arithmetic operat
particular instruc
on its edges can b
how the seemingl
fast cycle time, re
to be only 1.3 tin
In Figure 1, w
a fragment of a p
instruction-tree la
code registers cc1
the tree will be s
ai+4→ai will be
the previous instr

the new parallelization technique.

## 2. Overview of the Architecture

Our machine has multiple functional units all of which share a register file and multiple condition code registers that assume binary values (true or false). It supports multiway branch and conditional execution.

The instructions of the machine have the form of a decision tree (see Figure 1). The tree is encoded in binary form in the instruction word. At the terminal nodes of the tree there are labels, which this instruction can branch to. At each non-terminal node of the tree, there is a test on a condition code register (the machine has multiple condition code registers). On each directed edge of the tree there can be zero or more three-register arithmetic operations, or memory loads/stores. An instruction is executed in a single machine cycle. Conceptually, there are two phases in the machine cycle: the path selection phase and the execution phase.

At the path selection phase, the machine determines a unique path from the root of the tree to a tip node of the tree, based on the *old* values of the condition code registers that were set in the previous instructions, in a decision tree like fashion. If the test on a given node is true, the taken path branches to the left, otherwise the taken path branches to the right.

At the execution phase, only the three-register arithmetic operations and loads/stores that are on the selected path are executed, using the *old* values of the registers available from the previous instruction as operands or storage addresses. They are executed simultaneously since they have no data dependence on each other. If more than one operation sets the same destination register on the selected path, the operation closest to the tip node determines the final value in the destination register.

Finally the results of the operations are written into the register file, and control branches to the instruction whose label is indicated at the tip of the taken branch of the current instruction.

For the specific implementation, there will be a finite limit on the number of ***distinct*** arithmetic operations, loads/stores, and the number of branch target addresses in a particular instruction. Otherwise, the shape of the tree and the placement of operations on its edges can be arbitrary. For a more detailed discussion on the architecture, and on how the seemingly complex instruction semantics described here is implemented with a fast cycle time, readers should refer to [5]. The cycle time of this machine is estimated to be only 1.3 times that of a pipelined RISC machine designed in the same technology.

In Figure 1, we show an example of an instruction in our VLIW architecture, which is a fragment of a program for finding the minimum of an array. Just before executing the instruction-tree labeled L1 in the Figure 1, if for example the initial values of condition code registers cc1 and cc0 are false and true, respectively, then the rightmost path in the tree will be selected. The operations (t<min)→cc1, t→t', (ai<lim)→cc0, A(ai)→t, ai+4→ai will be executed in parallel, using the old values t, min, ai, lim, available from the previous instructions. The instruction will branch to L1 again. The next instruction

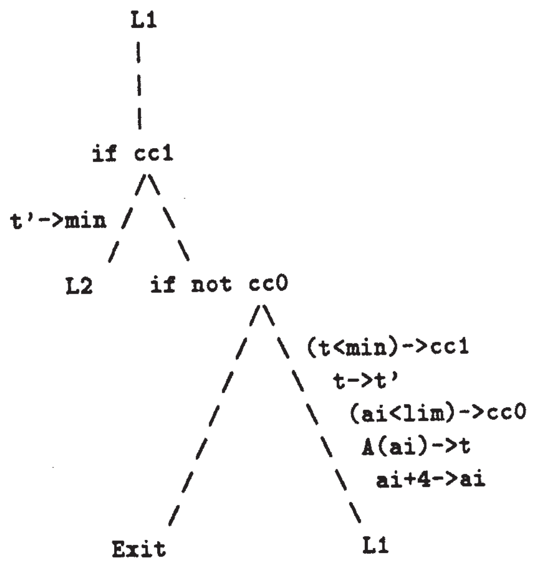

Figure 1: An example of an instruction in our VLIW architecture

will then observe t
not be executed si

The difference l
ELI architecture (
that conditional ju
be executed in par
conditional jumps

Our instruction
percolation schedu
conditionally depe
regardless of the p
delays over the ori

## 3. Outline of

The purpose of th
quential code prod
([15])) for a RISC
and produce a sequ
register arithmetic
give an overview o

In our project,
compiler ([15]) int
sary procedure cal
translates the inter
take as input to ou

The parallelize
Nicolau ([11]) and
two techniques ex
called *enhanced pi*
technique is given

The major dist
lau's percolation s
of our architectur
up, to eliminate i
dependence), and
and move-cj). We
loops (ignoring ba

Because only fi
of enhanced perco
optimal execution
operations are exe
But the compacte

will then observe the updated values of cc1, t', cc0, t and ai. The operation t'→min will not be executed since it is not on the selected path.

The difference between the instruction semantics of our VLIW machine and J. Fisher's ELI architecture ([7]), which was the basis for the Multiflow VLIW machines ([8]), is that conditional jumps that are not necessarily on a single path through the code can be executed in parallel in our machine, whereas the ELI allows the parallel execution of conditional jumps on a single "most probable" path.

Our instruction semantics is also different from A. Nicolau's original formulation of the percolation scheduling programming model ([11]), since our machine executes operations conditionally depending on where the instruction branches to, and thus gains headway regardless of the path taken. The conditional execution feature reduces critical path delays over the original percolation scheduling model.

## 3. Outline of the Compiler

The purpose of the compilation techniques for our machine is to take as input the sequential code produced by a traditional optimizing compiler (presently the PL.8 compiler ([15])) for a RISC machine, find groups of operations that can be executed in parallel, and produce a sequence of compacted instructions, each of which contains multiple three-register arithmetic operations, loads/stores, and multiway conditional jumps. We will give an overview of the compilation techniques in this section.

In our project, a frontend code generator has been developed, which takes the PL.8 compiler ([15]) intermediate code for each procedure as an input, inlines all the necessary procedure calls, solves the register save/restore problem by renaming registers, and translates the intermediate code to the sequential (abstract) RISC machine code that we take as input to our parallelizing compiler, using a rule-based code generation technique.

The parallelizer in our project is based on percolation scheduling derived from A. Nicolau ([11]) and software pipelining derived from K. Ebcioğlu ([4]). We modified the two techniques extensively and merged them together as a new parallelizing technique called *enhanced pipeline-percolation scheduling*. A more detailed description on this new technique is given in the next two sections. Here we will briefly describe this technique.

The major distinction of our enhanced percolation scheduling technique from Nicolau's percolation scheduling is: 1) reformulation into the conditional execution semantics of our architecture, 2) renaming of destination registers of operations that are moved up, to eliminate inhibition of parallelism due to read-before-write dependence (anti-dependence), and 3) new integration of core transformations (we have only move-op and move-cj). We apply this technique for loop-free code (DAG), like bodies of inner loops (ignoring backward edges).

Because only flow dependences remain after register renaming, a greedy application of enhanced percolation scheduling in the presence of unlimited resources, guarantees optimal execution of the original acyclic program. That is, in the compacted code, operations are executed as soon as their operands are ready, regardless of the path taken. But the compacted code stops executing the remaining operations on a path, as soon as it is known that the path is not to be taken by the original sequential code, so resources are conserved. Optimal execution on acyclic code is not achievable with trace scheduling or standard percolation scheduling, even with infinite resources.

The following is an overview of enhanced pipeline scheduling. First, the top level procedure picks an innermost loop as an instruction list. Here the instructions in the list are sorted in topological order of the control flow graph, ignoring backedges. Enhanced percolation scheduling is applied to move operations and conditional jumps to the first instruction of the instruction list, until the resource constraint limitation is encountered. We call this process "filling" the first instruction. Motion of operations is always in the upward direction of the instruction list and, in this initial stage, motion across backedges is inhibited. Motion across the original backedges is subsequently enabled and software pipelining occurs.

Once the first instruction is filled, it is placed at the bottom of the instruction list. It now represents the initiation of the second iteration of the loop. The successors (the level 2 instructions) of the first instruction, which have not already been filled and which are within the instruction list, are now filled, with the motions always in the upward direction of the instruction list. We note here that, in this scheme, operations from the first instruction of the second iteration can move into the level 2 instructions of the first iteration; thus, software pipelining is being done.

This procedure will then be repeated by moving down the level 2 instructions to the bottom of the instruction list, making the unfilled successors (the level 3 instructions) of the level 2 instructions the new set of instructions to be filled, and so on. This process will be repeated until no unfilled successors of the current level exist. A special rule applies for upward code motion out of already filled instructions to guarantee termination: if any operation or test is moved up on a given path from the filled instruction, then all operations and tests in the filled instruction must be moved up on that path.

At the end of this process, the innermost loop will have been transformed into the compacted tree instructions, which execute the loop in software pipelined fashion. Typically a number of additional instructions will be generated that start up the software pipeline (loop "prelude") and drain the software pipeline (loop "postlude").

For the next step, the pipeline startup and draining instructions of the inner loop are merged with the outer loop. The strongly connected part of the software pipelined inner loop is represented as an atomic unit in the outer loop. The same procedure is applied to the outer loop. The procedure is then again applied to the next level outer loop, etc. until the whole program is compacted and software pipelined.

## 4. Move-op in Enhanced Percolation Scheduling

In this section, we will describe a crucial component of enhanced percolation scheduling, the modified move-op transformation.

In Figure 2, we describe by an example how we have modified the move-op transformation of percolation scheduling, so that anti-dependences are eliminated. This transformation is the key to the optimal schedules produced by enhanced percolation scheduling

```
before move-op of x+y->z from L2 to left branch of L1:

               L1                      L2
               |                       |
               |                       |
               |                       |
            if cc1                  if cc2
       w<0->cc2/\                     /\
        x'->x /  \             A(z)->y / \ x+y->z
             /    \           x+y->v  /   \
           L2      L3               L4      L5

after move-op:
               L1                      L2'
               |                       |
               |                       |
               |                       |
            if cc1                  if cc2
       w<0->cc2/\                     /\
        x'->x /  \             A(z)->y / \ z'->z
     x'+y->z'/    \             z'->v /   \
          L2'      L3               L4      L5

(original L2 is deleted if it no longer has predecessors)
```

Figure 2: Elimination of anti-dependence in move-op

on acyclic code.

To move an operation x+y→z (where x, y, z are registers) from an instruction n to the end of a branch *tp* of a predecessor instruction m, first the following check is made: if the source registers x,y are set via a non-copy operation in branch *tp*, then the move cannot be done and move-op fails (copy operations are register transfers such as x'→x, other operations are non-copy operations). Otherwise, the move can be made. A copy of n is made and is renamed as n', and branch *tp* of m is made to go to n' instead of n. If n no longer has any predecessors, it is deleted.

The operation x+y→z is modified as follows when moving it from n' into branch *tp* of m: given that there are copy operations x1→y1,... xk→yk in branch *tp* of m, the source registers of the operation are modified by substituting xi for yi (to get the new value of yi, it suffices to look into xi in m). The destination register z of x+y→z is also modified by changing it to a brand new register z' before making the move, to guarantee that those operations, which need the old value of z in n' and after n', still get the old value of z. All operations that have the same sources and opcode as x+y→z in n', namely x+y→z, x+y→z1, ..., are changed to z'→z, z'→z1, .... As a special case, as in ordinary compaction techniques, if the original destination register z of x+y→z is not used by any operation other than x+y→... in n', and for each branch of n', z is either set in the branch or is dead at the target of the branch, then z can be used instead of the brand new register z' (this obviates the need for the z'→z copy operation). Many of such copies will subsequently be eliminated through register coloring with coalescing ([3]) or dead code elimination.

The example in Figure 2 shows how the operation x+y→z is moved from instruction L2 to the left branch of L1. Since there is a copy operation x'→x in this branch of L1, the operation x+y→... has been modified as x'+y→.... Also, the destination register z of x+y→z has been renamed as z', since the old value of z cannot be destroyed (the old value is needed by another operation, namely, A(z)→y).

Other enhancements have also been made to move-op in our implementation for reducing pathlengths, such as compile-time disambiguation of memory references for moving loads above stores, and a "combining" feature ([6]), which removes flow dependences between seemingly dependent ops; e.g. y>10→cc1 can move to a predecessor containing x+2→y, as x>8→cc1.

## 5. Enhanced Pipeline Scheduling Algorithm

We will now describe enhanced pipeline scheduling. The version described here applies to a single inner loop for simplicity. The algorithm takes a global list (the instruction list in the previous section) of VLIW instructions "prog", which is initially sorted in depth-first order ([1]), as input, and modifies "prog" so it becomes parallelized.

```
procedure enhanc

for each instruc
let fence= {entı
while fence is ı
   for each inst
      fill(n)


   let newfence=





    Remove all ı
    Append all ı
    For all n iı
    fence= newf
end while
end enhanced_pi
```

As far as correc
constrained compa
such as one based
compaction algorit

```
procedure  fill

/* This procedu
operations and
resource constr
prog. When movi
instruction n'
in prog from n'
moved up to n o

end fill
```

Enhanced pipe
paction algorithm
serving compactio
scheduling ([11])).

```
procedure enhanced_pipeline_sched()

for each instruction n in prog do filled(n)=false
let fence= {entry instruction of prog}
while fence is not empty do:
   for each instruction n in fence do
        fill(n) /* move to n operations in prog
                that are below n, subject to resource constraints */

   let newfence= {s | (exists n in fence)
                        [s is in prog and
                         s is not in fence and
                         not filled(s) and
                         s is a successor of n and
                         s occurs after n in prog]}

    Remove all n in fence from prog
    Append all n in fence to the end of prog
    For all n in fence, set filled(n)=true
    fence= newfence
end while
end enhanced_pipeline_sched
```

As far as correct operation of enhanced pipeline scheduling is concerned, any resource-constrained compaction algorithm for acyclic programs would do in place of fill(n) here, such as one based on [6]. We will not give the implementation details of our acyclic code compaction algorithm since this is beyond the scope of this paper.

```
procedure  fill(n)

/* This procedure modifies the global list prog. Moves all
operations and tests which can be moved to n (subject to
resource constraints), from instructions below n in the list
prog. When moving operations and tests from an already filled
instruction n' to the fence instruction n on a given upward path p
in prog from n' to n, either all operations and tests in n' are,
moved up to n on  that path p, or n' is left intact.  */

end fill
```

Enhanced pipeline scheduling is clearly correct or semantics preserving, if the compaction algorithm used by "fill" is semantics preserving (an example of a semantics preserving compaction algorithm would be one that uses core transformations of percolation scheduling ([11])). To see informally why enhanced pipeline scheduling should terminate for an inner loop, notice that once an operation is placed in a fence instruction and the instruction is marked filled, that operation will always end up in instructions that are marked filled at the end of each iteration of the enhanced pipeline scheduling loop. Since the new fence at the end of each iteration has to come from instructions that are not marked filled, there will eventually be no more operations left to form such unfilled instructions, and the fence will be empty. Rigorous proof of this claim will of course require a detailed description of basic compaction algorithm "fill," which is omitted here.

## 6. A Parallelization Example

We will now give an example of the application of enhanced pipeline scheduling. Consider the following loop:

```
for(i=0;i<n;i++)
   {if (f(x)<k) x=h(x); else x=g(x);}
```

where f, h and g are single cycle operations.

In the following examples, we will use the notations "op1, op2, ..., goto label", or "op1, op2, ..., if cc1 (instr1) else (instr2)" to represent the instruction trees of our VLIW machine; we hope the correspondence between this notation and the instruction-trees will be obvious to the reader. The input for the enhanced pipelining algorithm (the initial value of prog) is a set of sequential tree-instructions for the loop, where each instruction contains at most one conditional branch or one operation:

```
loop: i<n->cc0, goto L2
L2:   if not cc0 (goto exit) else (goto L3)
L3:   f(x)->t, goto L4
L4:   t<k->cc1, goto L5
L5:   if not cc1 (goto L7) else (goto L6)
L6:   h(x)->x, goto L8
L7:   g(x)->x, goto L8
L8:   i+1->i, goto loop
exit:  cc0,cc1,t,i are dead in here;
       x is live in here.
```

Prog after filling the first fence = {loop}:

```
loop: i<n->cc0, f(x)->t, g(x)->x', h(x)->x'', i+1->i, goto L2'
L2':  if not cc0 (goto exit) else (x''->x, goto L4')
L4':  t<k->cc1, goto L5'
L5':  if not cc1 (x'->x, goto loop) else (goto loop)
```

Now the current fence {loop} is removed from prog and appended to the end of prog. The next fence is {L2'}, which, when filled, results in the following state in prog:

```
L2': if not cc0
     else (x''-
          t<k-
          i<n-
          f(x'
          g(x'
          h(x'
          f(x'
          g(x'
          h(x'
          i+1-
          goto
L5'': if not cc
     else (t'->
loop: i<n->cc0,
```

Then L2' is ta
result in prog (aft

```
/* let xj= valu
/* it. 0 comput
loop: i<n->cc0,

L2':  if not cc
      else (x''
           t<k-
           i<n-
           f(x'
           g(x'
           h(x'
           f(x'
           g(x'
           h(x'
           i+1-
           goto

/* invariant as
  (j=0 the fir
  x=h(x(j))
  x'=g(x(j))
  t=f(g(x(j)))
  x'''=g(g(x(j
  x'' =h(g(x(j
L2': if not cc0 (goto exit)
     else (x''->x,
          t<k->cc1,
          i<n->cc0,
          f(x')->t,
          g(x')->x'''
          h(x')->x''
          f(x'')->t'
          g(x'')->x''''
          h(x'')->x'''''
          i+1->i
          goto L5'')
L5'': if not cc1 (x'->x,x'''->x', goto L2')
     else (t'->t, x''''->x', x'''''->x'', goto L2')
loop: i<n->cc0, f(x)->t, g(x)->x', h(x)->x'',i+1->i, goto L2'
```

Then L2' is taken to the bottom of prog and the new fence {L5"} is filled. The final result in prog (after taking L5" to the end of prog) is given below, with comments:

```
/* let xj= value of x before iteration j */
/* it. 0 computes f(x0),g(x0), h(x0) */
loop: i<n->cc0, f(x)->t, g(x)->x', h(x)->x'', i+1->i, goto L2'

L2':  if not cc0 (goto exit) /* it. 0 exits with x=x0 */
      else (x''->x,
           t<k->cc1, /* it. 0 computes f(x0)<k  */
           i<n->cc0,
           f(x')->t,  /* ver. A of it. 1  */
           g(x')->x'''/* assumes x1= x' = g(x0) */
           h(x')->x''
           f(x'')->t' /* ver. B of it. 1 */
           g(x'')->x'''' /* assumes x1=x''=h(x0) */
           h(x'')->x'''''
           i+1->i
           goto L5'')

/* invariant assertions before execution of L5'' for j=0,1,...
   (j=0 the first time)             t'  =f(h(x(j)))
   x=h(x(j))                        x''''  = g(h(x(j)))
   x'=g(x(j))                       x''''' = h(h(x(j)))
   t=f(g(x(j)))                     i      = i(j+2)
   x'''=g(g(x(j)))                  cc1 =  (f(x(j))<k)
   x'' =h(g(x(j)))                  cc0  =  (i(j+1)<n)  */
```

```
L5'': if not cc1 (x'->x,  /* this branch taken if x1=x'=g(x0) */
                  x'''->x',
                  if not cc0 (goto exit) /* it. 1 exits with x=x1*/
                  else (x''->x,
                        t<k->cc1, /* it 1. computes f(x1)<k */
                        i<n->cc0,
                        f(x''')->t, /* ver. A of it. 2 assumes */
                        g(x''')->x'''/* x2=x'''=g(x1) */
                        h(x''')->x''
                        f(x'')->t'   /* ver. B of it. 2 assumes */
                        g(x'')->x'''' /* x2=x''=h(x1) */
                        h(x'')->x'''''
                        i+1->i
                        goto L5''))
      else (x''''->x', /* this path taken if x1=x=h(x0)*/
            if not cc0 (goto exit) /* it. 1 exits with x=x1 */
            else (x'''''->x,
                  t'<k->cc1,  /* it. 1 computes f(x1)<k */
                  i<n->cc0,
                  f(x'''')->t, /* ver. A of it. 2 assumes*/
                  g(x'''')->x'''/*x2=x''''=g(x1) */
                  h(x'''')->x''
                  f(x''''')->t' /* ver. B of it. 2 assumes */
                  g(x''''')->x'''' /* x2=x'''''=h(x1) */
                  h(x''''')->x'''''
                  i+1->i
                  goto L5''))
```

The resource requirements for the largest instruction L5" are: 4 way branching, 16 arithmetic/compare units, and 5 copy units.

The enhanced pipeline scheduling technique parallelizes this loop optimally (one iteration/cycle), in an interesting manner. Let x0, x1, ... be the value of x at the beginning of iteration i = 0, 1, ... of the given loop.

In the first cycle (loop) of the code produced by enhanced pipeline scheduling, iteration 0 computes f(x0), and both of h(x0) and g(x0), to make optimal preparation for both of the paths that might be taken by iteration 0.

In the second cycle (L2'), iteration 0 starts evaluating f(x0) < k, while two versions of iteration 1 are started, each of which computes f(x1), h(x1) and g(x1). One version of iteration 1 uses the value of h(x0) as x1, and the other version uses g(x0) as x1. Here we do not know which version of iteration 1 will be the correct one, since f(x0) < k is not evaluated yet, so we proceed on both paths.

Finally, in the third cycle (L5"), the value of f(x0) < k becomes available for iteration 0, and the decision is made as to which value, h(x0) or g(x0), will be assigned to x in iteration 0. The version of iteration 1, which made the incorrect assumption about the

path taken ma
iteration 1 sta
of which evalu
version uses g(
steady state.
That is, fo
ecution of L5".
started versio
starts computi
pute f(x(j+2)),
another versio
therefore allow
The salient
values are assi
On at least on
is shorter than
path taken by
root finding by
The resour
by A.K. Uht ([
([12]), about t
techniques. Fr
of the depende
not input depe
such loops opti
The enhan
Although in ge
of such loops, i
by enhanced p
resources.

## 7. Compar

Aiken and Nic
pipelining of lo
number of tim
starting from t
is another instr
the meaning of
perhaps that t
are then redire
This comp
should be a lir

path taken made by iteration 0, is abandoned. At the same time the surviving version of iteration 1 starts to evaluate f(x1) < k, and two versions of iteration 2 are started, each of which evaluates f(x2), h(x2), and g(x2). One version uses h(x1) as x2, and the other version uses g(x1) as x2. The code will then branch back to instruction L5", which is a steady state.

That is, for j = 0, 1, 2, ..., the following simultaneous events happen in each execution of L5". Iteration j makes its choice of path to take, one of the two previously started versions of iteration j+1 is abandoned, the surviving version of iteration j+1 starts computing f(x(j+1)) < k, and two versions of iteration j+2 are started that compute f(x(j+2)), h(x(j+2)), and g(x(j+2)): one version using h(x(j+1)) as x(j+2) and another version using g(x(j+1)) as x(j+2). The enhanced pipeline scheduling technique therefore allows one iteration/cycle (optimal) execution of this loop.

The salient properties of this loop are: 1) iteration j has multiple paths where different values are assigned to a variable x, which is always used by the next iteration j+1. 2) On at least one path in iteration j, the dependence chain to compute the new value of x is shorter than the dependence chain to compute the condition code that determines the path taken by iteration j. Loops for some practical problems such as binary search, or root finding by bisection method fall into this category.

The resource requirements for the optimal parallelization of such loops were analyzed by A.K. Uht ([14]), in an attempt to disprove a previous claim by Riseman and Foster ([12]), about the excessiveness of the resource requirements for such eager evaluation techniques. From polynomial to exponential resources are required, in terms of the length of the dependence chain to compute the condition code. The resource requirements are not input dependent. But the problem of actually finding the algorithm to parallelize such loops optimally was left open by Uht.

The enhanced pipeline scheduling algorithm provides one answer to this problem. Although in general the resource requirements may be exponential for optimal execution of such loops, in many serial loops of systems/commercial code, the schedules generated by enhanced pipeline technique can be small enough to fit within a VLIW with large resources.

## 7. Comparison to Previous Work

Aiken and Nicolau's perfect pipelining technique ([2]), is a different approach to software pipelining of loops with tests. In this approach, the loop is first unrolled for a sufficient number of times and compaction is made. Then on each path of the compacted code starting from the entry instruction, a search is made for an instruction n such that there is another instruction n', not necessarily on the same path, which is equivalent to n. Here the meaning of equivalence is: starting in n gives the same result as starting in n', except perhaps that the iteration trip counter i has to be incremented. All edges that go to n are then redirected to n'.

This compaction algorithm, however, must be well-behaved in the sense that there should be a limit on the number of cycles (we call *gaps*) between the execution of operations of the same iterations. Having no gaps between operations of an iteration seems to allow reasonable convergence rates in perfect pipelining (this is the *simple rule* of Aiken and Nicolau). If large gaps are allowed, the number of unrollings required to guarantee convergence of the algorithm on all paths must be increased.

Enhanced pipeline scheduling is an outgrowth of pipeline scheduling ([4]), and allows gaps between the operations of the same iteration without sacrificing the convergence time. Gaps between operations of the same iteration can be crucial for optimal performance. Namely, there are loops where iteration n+1 has to start as soon as its first operation can start, and then potentially wait. This is the only way we can achieve good performance if the following optimistic assumption holds: iteration n and n+1 will both take paths such that iteration n will provide data in time for subsequent operations of iteration n+1, and iteration n+1 will continue without pausing. But if the optimistic assumption fails and iteration n+1 or n takes a path where the data needed for iteration n+1 is not yet available from iteration n, subsequent operations of iteration n+1 may have to pause and wait until data is available (they may wait till the end of iteration n). Thus allowing large pauses (or gaps) does increase performance in loops with conditional jumps.

M. Lam ([9]) has also designed an algorithm for pipelining loops with tests, which is based on an extension of software pipelining of IFless loops. This technique pads out the shorter path of an if-then-else with no-ops, and subsequently treats the aggregate if-then-else as a single operation that is moved as a unit, thus allowing the use of the basic algorithm for software pipelining of IFless loops.

In general, however, two paths of an if-then-else may imply two different dependence cycles and two different iteration issue rates, and treating the if-then-else as a single unit entails accepting the worst of the two different iteration issue rates.

Enhanced pipeline scheduling, like pipeline scheduling ([4]), does not make such an assumption and generates schedules with variable, data-dependent iteration issue rates.

The optimal schedules produced by enhanced pipeline scheduling on many practical examples lead to the interesting question whether enhanced pipeline scheduling will achieve optimal performance on arbitrary loops.

Achieving time-optimal performance on arbitrary loops with tests, however, even when assuming unlimited resources and limiting ourselves to one iteration per cycle, is impossible for ***any*** software pipelining algorithm, as it was proved in [13]. This proof proceeds by demonstrating a loop whose optimal execution requires an amount of resources which is input-dependent; in particular, the number of paths on which speculative execution of an operation must be performed increases with the input. Thus, optimal execution cannot be achieved by any finite VLIW program.

Despite its suboptimality in general, however, our enhanced pipeline scheduling gives good results in practice as shown in the next section.

## 8. Some Ber

We have run seve
at the IBM Toky
The intermediate
the frontend, wh
and inlined subr
by our VLIW pa
scheduling techn
register transfer
constraints used
and unlimited ov

In Figure 3,
machine which c
or store in a sin
VLIW machine
some of the Stan
the simulation t
do the following:
for a matching e
array).

We note her
into our compil
unrolling featur
hancement usin

## 9. Conclusi

We have present
tively extract fin

| Program Name | Sequential | Parallel | |
|---|---|---|---|
| | Count | Count | Ratio |
| Sieve (size=15, it=1) | 419 | 75 | 5.6 |
| Sibubble (size=10) | 894 | 115 | 7.8 |
| Siperm (size=3) | 747 | 99 | 7.5 |
| Siquick (size=10) | 1025 | 209 | 4.9 |
| Sitree (size=15) | 2474 | 436 | 5.7 |
| Mergec (size=20) | 403 | 46 | 8.8 |
| Inserc (size=10) | 1145 | 197 | 5.8 |
| Pointerc (size=20) | 169 | 26 | 6.5 |
| Minmaxc (size=171) | 1047 | 96 | 10.9 |

Figure 3: Benchmark results

## 8. Some Benchmark Results

We have run several examples through our VLIW parallelizing compiler now operational at the IBM Tokyo Research Laboratory and at the IBM T.J. Watson Research Center. The intermediate code produced by the PL.8 compiler from C programs were fed to the frontend, which translated the intermediate code to VLIW sequential RISC format and inlined subroutines. Then this sequential RISC code was automatically parallelized by our VLIW parallelizing compiler that implements the enhanced pipeline-percolation scheduling technique. Finally the parallel code was assembled and simulated on our register transfer level simulator for our VLIW architecture (written in C). The resource constraints used here were 16 arithmetic operations, 16 loads or stores, 16 way branching, and unlimited overhead copy operations (generated by the compiler during compaction).

In Figure 3, we give the number of RISC instructions executed in the abstract RISC machine which can do one arithmetic operation, or one conditional branch, or one load or store in a single instruction, and the number of VLIW instructions executed in the VLIW machine after the RISC code has been compacted. The programs used include some of the Stanford integer benchmarks (with array bounds reduced in order to decrease the simulation time), and some other sequential-natured C programs (that respectively do the following: merge two sorted arrays into a third sorted array, insertion sort, search for a matching element in a list of linked records, find the minimum and maximum of an array).

We note here that unrolling feature (unrolling innermost loops) can be incorporated into our compiler to get a further performance improvement. For the details of the unrolling feature, other new techniques such as *combining*, and further performance enhancement using these techniques, readers should refer to [10].

## 9. Conclusions

We have presented the enhanced pipeline-percolation scheduling technique that can effectively extract fine-grain parallelism from general nested loops with unpredictable branches.

We have also described a VLIW architecture that has the architectural features to extract parallelism from highly sequential-natured code, and that supports the code generated by the enhanced pipeline-percolation scheduling technique. The approach seems to be robust, and thus it seems to be a significant step toward building compilers and hardware for parallelizing arbitrary sequential applications.

## 10. Acknowledgements

We would like to thank the other members of the VLIW group in IBM Thomas J. Watson Research Center, in particular Franco Gasperoni, and Uwe Schwiegelshohn, for helpful discussions on the compilation techniques and architecture described here. We also appreciate Tsutomu Kamimura and Susan Yeh of IBM Tokyo Research Laboratory for reading a preliminary draft of this paper. Parts of the parallelizing compiler currently being used were initially coded by Mauricio Breternitz.

## References

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

[2] Aiken, A. and Nicolau, A. [1988]. Perfect Pipelining: A New Loop Parallelization Technique. In *European Symposium on Programming*, pp. 221-235, Springer-Verlag, Lecture Notes in Computer Science No. 300.

[3] Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., and Markstein, P.W. [1981]. Register Allocation via Coloring. *Computer Languages* 6, pp. 47-57.

[4] Ebcioğlu, K. [1987]. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, pp. 69-79, ACM Press.

[5] Ebcioğlu, K. [1988]. Some Design Ideas for a VLIW Architecture for Sequential Natured Software. In *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing)*, M. Cosnard et al. (eds.), pp. 1-21, North Holland.

[6] Ebcioğlu, K. and Nicolau, A. [1989]. A Global Resource Constrained Parallelization Technique. To appear in *Proceedings of 1989 International Conference on Supercomputing*, Crete, Greece.

[7] Fisher, J.A. [1983]. Very Long Instruction Word Architectures and the ELI-512. *Proceedings of the 10th Annual Symposium on Computer Architecture*, pp. 140-150.

[8] Fisher, J.A. [1984]. The VLIW Machine: A Multiprocessor for Compiling Scientific Code. *IEEE Computer* 17(7), pp. 45-53.

[9] Lam, M. [1
Machines. I
*guage Desi*

[10] Nakatani, T
VLIW Arc
*Microprogr*

[11] Nicolau, A.
*85-678*, De

[12] Riseman, E
Conditiona

[13] Schwiegelsh
allelization.
*gramming*

[14] Uht, A. K
*Proceeding*
pp. 230-23

[15] Warren, S.
MacKay, A
*11974*, IBM

[9] Lam, M. [1988]. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN 1988 Conference of Programming Language Design and Implementation*, pp. 318-328, ACM Press.

[10] Nakatani, T. and Ebcioğlu K. [1989]. "Combining" as a Compilation Technique for VLIW Architectures. To appear in *Proceedings of the 22nd Annual Workshop on Microprogramming and Microarchitecture*, ACM Press.

[11] Nicolau, A. [1985]. Percolation Scheduling: A Parallel Compilation Technique. *TR 85-678*, Department of Computer Science, Cornell University.

[12] Riseman, E.M. and Foster, C.C. [1972]. The Inhibition of Potential Parallelism by Conditional Jumps. *IEEE Transactions on Computers*, C-21(12), pp. 1405-1411.

[13] Schwiegelshohn, U., Gasperoni, F., and Ebcioğlu, K. [1989]. On Optimal Loop Parallelization. To appear in *Proceedings of the 22nd Annual Workshop on Microprogramming and Microarchitecture*, ACM Press.

[14] Uht, A. K. [1988]. Requirements for Optimal Execution of Loops with Tests. In *Proceedings of 1988 International Conference on Supercomputing*, St. Malo, France, pp. 230-237, ACM Press.

[15] Warren, S.H., Auslander, M.A., Chaitin, G.J., Chibib, A.C., Hopkins, M.E., and MacKay, A.L. [1986]. Final Code Generation in the PL.8 Compiler. *Report No. RC 11974*, IBM T. J. Watson Research Center.