# Global Instruction Scheduling for Superscalar Machines

*David Bernstein*
*Michael Rodeh*

IBM Israel Scientific Center
Technion City
Haifa 32 000
ISRAEL

## Abstract

To improve the utilization of machine resources in superscalar processors, the instructions have to be carefully scheduled by the compiler. As internal parallelism and pipelining increases, it becomes evident that scheduling should be done beyond the basic block level. A scheme for global (intra-loop) scheduling is proposed, which uses the control and data dependence information summarized in a Program Dependence Graph, to move instructions well beyond basic block boundaries. This novel scheduling framework is based on the parametric description of the machine architecture, which spans a range of superscalar and VLIW machines, and exploits speculative execution of instructions to further enhance the performance of the general code. We have implemented our algorithms in the IBM XL family of compilers and have evaluated them on the IBM RISC System/6000 machines.



Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation. Toronto, Ontario, Canada, June 26-28, 1991.

## 1. Introduction

Starting in the late seventies, a new approach for building high speed processors emerged which emphasizes streamlining of program instructions; subsequently this direction in computer architecture was called RISC [**P85**]. It turned out that in order to take advantage of pipelining so as to improve performance, instructions have to be rearranged, usually at the intermediate language or assembly code level. The burden of such transformations, called *instruction scheduling*, has been placed on optimizing compilers.

Previously, scheduling algorithms at the instruction level were suggested for processors with several functional units [**BJR89**], pipelined machines [**BG89**, **BRG89**, **HG83**, **GM86**, **W90**] and Very Large Instruction Word (VLIW) machines [**E85**]. While for machines with $n$ functional units the idea is to be able to execute as many as $n$ instructions each cycle, for pipelined machines the goal is to issue a new instruction every cycle, effectively eliminating the so-called NOPs (No OPerations). However, for both types of machines, the common feature required from the compiler is to discover in the code instructions that are *data independent*, allowing the generation of code that better utilizes the machine resources.

It was a common view that such data independent instructions can be found within basic blocks, and there is no need to move instructions beyond basic

block boundaries. Virtually, all of the previous work on the implementation of instruction scheduling for pipelined machines concentrated on scheduling within basic blocks [**HG83**, **GM86**, **W90**]. Even for basic RISC architectures such restricted type of scheduling may result in code with many NOPs for certain Unix[1] -type programs that include many small basic blocks terminated by unpredictable branches. On the other hand, for scientific programs the problem is not so severe, since there, basic blocks tend to be larger.

Recently, a new type of architecture is evolving that extends RISC by the ability to issue more than one instruction per cycle [**GO89**]. This type of high speed processors, called *superscalar* or *superpipelined* architecture, poses more serious challenges to compilers, since instruction scheduling at the basic block level is in many cases not sufficient to allow generation of code that utilizes machine resources to a desired extent [**JW89**].

One recent effort to pursue instruction scheduling for superscalar machines was reported in [**GR90**], where *code replication techniques* for scheduling beyond the scope of basic blocks were investigated, resulting in fair improvements of running time of the compiled code. Also, one can view a superscalar processor as a VLIW machine with a small number of resources. There are two main approaches for compiling code for the VLIW machines that were reported in the literature: the *trace scheduling* [**F81**, **E85**] and the *enhanced percolation scheduling* [**EN89**].

In this paper, we present a technique for *global instruction scheduling* which permits the movement of instructions well beyond basic blocks boundaries within the scope of the enclosed loop. The method employs a novel data structure, called the Program Dependence Graph (PDG), that was recently proposed by Ferrante et. al [**FOW87**] to be used in compilers to expose parallelism for the purposes of vectorization and generation of code for multiprocessors. We suggest combining the PDG with the parametric description of a family of superscalar machines, thereby providing a powerful framework for global instruction scheduling by optimizing compilers.

While trace scheduling assumes the existence of a main trace in the program (which is likely in scientific computations, but may not be true in symbolic or Unix-type programs), global scheduling (as well as enhanced percolation scheduling) does not depend on such assumption. However, global scheduling is capable of taking advantage of the branch probabilities, whenever available (e.g. computed by profiling). As for the enhanced percolation scheduling, our opinion is that it is more targeted towards a machine with a large number of computational units, like VLIW machines.

Using the information available in a PDG, we distinguish between *useful* and *speculative* execution of instructions. Also, we identify the cases where instructions have to be *duplicated* in order to be scheduled. Since we are currently interested in machines with a small number of functional units (like the RISC System/6000 machines), we established a conservative approach to instruction scheduling. First we try to exploit the machine resources with useful instructions, next we consider speculative instructions, whose effect on performance depends on the probability of branches to be taken, and scheduling with duplication, which might increase the code size incurring additional

---

[1] Unix is a trademark of AT&T Bell Labs

costs in terms of instruction cache misses. Also, we do not overlap the execution of instructions that belong to different iterations of the loop. This more aggressive type of instruction scheduling, which is often called *software pipelining* [**L88**], is left for future work.

For speculative instructions, previously-it was suggested that they have to be supported by the machine architecture [**E88**, **SLH90**]. Since architectural support for speculative execution carries a significant run-time overhead, we are evaluating techniques for replacing such support with compile-time analysis of the code, still retaining most of the performance effect promised by speculative execution.

We have implemented our scheme in the context of the IBM XL family of compilers for the IBM RISC System/6000 (RS/6K for short) computers. The preliminary performance results for our scheduling prototype were based on a set of SPEC benchmarks [**S89**].

The rest of the paper is organized as follows. In Section 2 we describe our generic machine model and show how it is applicable to the RS/6K machines. Then, in Section 3 we bring a small program that will serve as a running example. In Section 4 we discuss the usefulness of the PDG, while in Section 5 several levels of scheduling, including speculative execution, are presented. Finally, in Section 6 we bring some performance results and conclude in Section 7.

## 2. Parametric machine description

Our model of a superscalar machine is based on the description of a typical RISC processor whose only instructions that reference memory are load and store instructions, while all the computations are done in registers. We view a superscalar machine as a collection of functional units of $m$ types, where the machine has $n_1, n_2, ..., n_m$ units of each type. Each instruction in the code can be potentially executed by any of the units of a specified type.

For the instruction scheduling purposes, we assume that there is an unbounded number of symbolic registers in the machine. Subsequently, during the register allocation phase of the compiler, the symbolic registers are mapped onto the real machine registers, using one of the standard (*coloring*) algorithms. Throughout this paper we will not deal with register allocation at all. For the discussion on the relationships between instruction scheduling and register allocation see [**BEH89**].

A program instruction requires an integral number of machine cycles to be executed by one of the functional units of its type. Also, there are pipelined constraints imposed on the execution of instructions which are modelled by the integer delays assigned to the data dependence edges of the computational graph.

Let $I_1$ and $I_2$ be two instructions such that the edge $(I_1,I_2)$ is a data dependence edge. Let $t$ ($t \geq 1$) be the execution time of $I_1$ and $d$ ($d \geq 0$) be the delay assigned to $(I_1,I_2)$. For performance purposes, if $I_1$ is scheduled to start at time $k$, then $I_2$ should be scheduled to start no earlier than $k + t + d$. Notice, however, that if $I_2$ is scheduled (by the compiler) to start earlier than mentioned above, this would not affect the correctness of the program, since we assume that the machine implements hardware interlocks to guarantee the delays at run time. More information about the notion of delays due to pipelined constraints can be found in [**BG89**, **BRG89**].

### 2.1 The RS/6K model

Here we show how our generic model of a superscalar machine is configured to fit the RS/6K machine. The RS/6K processor is modelled as follows:

- $m = 3$, there are three types of functional units: fixed point, floating point and branch types.
- $n_1 = 1$, $n_2 = 1$, $n_3 = 1$, there is a single fixed point unit, a single floating point unit and a single branch unit.
- Most of the instructions are executed in one cycle, however, there are also multi-cycle instructions, like multiplication, division, etc.
- There are four main types of delays:
  - a delay of one cycle between a load instruction and the instruction that uses its result register (*delayed load*);
  - a delay of three cycles between a fixed point compare instruction and the branch instruction that uses the result of that compare[2] ;
  - a delay of one cycle between a floating point instruction and the instruction that uses its result;
  - a delay of five cycles between a floating point compare instruction and the branch instruction that uses the result of that compare.

  There are a few additional delays in the machine whose effect is secondary.

In this paper we concentrate on fixed point computations only. Therefore, only the first and the second types of the above mentioned delays will be considered.

## 3. A program example

Next, we present a small program (written in C) that computes the minimum and the maximum of an array. This program is shown in Figure 1 and will serve us as a running example.

In this program, concentrating on the loop which is marked in Figure 1, we notice that two elements of the array $a$ are fetched every iteration of the loop. Next, these elements of $a$ are compared one to another ($if(u > v)$) , and subsequently they are compared to the `max` and `min` variables, updating the maximum and the minimum, if needed. The RS/6K pseudo-code for the loop, that corresponds to the real code created by the IBM XL-C compiler[3] , is presented in Figure 2.

For convenience, we number the instructions in the code of Figure 2 (I1-I20) and annotate them with the corresponding statements of the program of Figure 1. Also, we mark the ten basic blocks (BL1-BL10) of which the code of Figure 2 comprises for the purposes of future discussion.

For simplicity of notation, the registers mentioned in the code are real. However, as was mentioned in Section 2, we prefer to invoke the global scheduling algorithm before the register allocation is done (at this stage there is an unbounded number of registers in the code), even though conceptually there is no problem to activate the instruction scheduling after the register allocation is completed.

---

[2] More precisely, usually the three cycle delay between a fixed point compare and the respective branch instruction is encountered only when the branch is taken. However, here for simplicity we assume that such delay exists whether the branch is taken or not.

[3] The only feature of the machine that was disabled in this example is that of keeping the iteration variable of the loop in a special *counter register*. Keeping the iteration variable in this register allows it to be decremented and tested for zero in a single instruction, effectively reducing the overhead for loop control instructions.

```
/* find the largest and the smallest number
                          in a given array */
minmax(a,n) {
int i,u,v,min,max,n,a[SIZE];
min=a[0]; max=min; i=1;
/****************** LOOP STARTS *************/
while (i < n) {
    u=a[i]; v=a[i+1];
    if (u>v) {
          if (u>max) max=u;
          if (v<min) min=v;
    }
    else  {
          if (v>max) max=v;
          if (u<min) min=u;
    }
    i= i+2;
}
/****************** LOOP ENDS ***************/
printf("min=%d max=%d\n",min,max);
}
```

Figure 1. A program computing the minimum and the maximum of an array

Every instruction in the code of Figure 2, except for branches, requires one cycle in the fixed point unit, while the branches take one cycle in the branch unit. There is a one cycle delay between instruction I2 and I3, due to the delayed load feature of the RS/6K. Notice the special form of a load with update instruction in I2: in addition to assigning to r0 the value of the memory location at address (r31) + 8, it also increments r31 by 8 (post-increment). Also, there is a three cycle delay between each compare instruction and the corresponding branch instruction. Taking into consideration that the fixed point unit and the branch unit run in parallel, we estimate that the code executes in 20, 21 or 22 cycles, depending on if 0, 1 or 2 updates of max and min variables (LR instructions) are done, respectively.

```
max is kept in r30
min is kept in r28
i is kept in r29
n is kept in r27
address of a[i] is kept in r31
 ...  more instructions here ...
*************** LOOP STARTS ******************
    CL.0:
(I1)     L    r12=a(r31,4)     load u
(I2)     LU   r0,r31=a(r31,8) load v and
                                 increment index
(I3)     C    cr7=r12,r0       u > v
(I4)     BF   CL.4,cr7,0x2/gt
---------------------------------- END BL1
(I5)     C    cr6=r12,r30      u > max
(I6)     BF   CL.6,cr6,0x2/gt
---------------------------------- END BL2
(I7)     LR   r30=r12          max = u
---------------------------------- END BL3
    CL.6:
(I8)     C    cr7=r0,r28       v < min
(I9)     BF   CL.9,cr7,0x1/lt
---------------------------------- END BL4
(I10)    LR   r28=r0           min = v
(I11)    B    CL.9
---------------------------------- END BL5
    CL.4:
(I12)    C    cr6=r0,r30       v > max
(I13)    BF   CL.11,cr6,0x2/gt
---------------------------------- END BL6
(I14)    LR   r30=r0           max = v
---------------------------------- END BL7
    CL.11:
(I15)    C    cr7=r12,r28      u < min
(I16)    BF   CL.9,cr7,0x1/lt
---------------------------------- END BL8
(I17)    LR   r28=r12          min = u
---------------------------------- END BL9
    CL.9:
(I18)    AI   r29=r29,2        i = i+2
(I19)    C    cr4=r29,r27      i < n
(I20)    BT   CL.0,cr4,0x1/lt
---------------------------------- END BL10
*************** LOOP ENDS ********************
 ...  more instructions here ...
```

Figure 2. The RS/6K pseudo-code for the program of Figure 1

## 4. The Program Dependence Graph

The program dependence graph is a convenient way to summarize both the *control dependences* and *data dependences* among the code instructions. While the concept of data dependences, that carries the basic idea of one instruction computing a data value and another instruction using this value, was employed in compilers a long time ago, the notion of control dependences was introduced quite recently [**FOW87**]. In what follows we discuss the notions of control and data dependences separately.

### 4.1. Control dependences

We describe the idea of control dependence using the program example of Figure 1. In Figure 3 the control flow graph of the loop of Figure 2 is described, where each node corresponds to a single basic block in the loop. The numbers inside the circles denote the indices of the ten basic blocks BL1-BL10. We augment the graph of Figure 3 with unique ENTRY and EXIT nodes for convenience. Throughout this discussion we assume a single entry node in the control flow graph, i.e., there is a single node (in our case BL1) which is connected to ENTRY. However several exit nodes that have the edges leading to EXIT may exist. In our case BL10 is a (single) exit node. For the strongly connected regions (that represent loops in this context), the assumption of a control flow graph having a single entry corresponds to the assumption that the control flow graph is reducible.

The meaning of an edge from a node $A$ to a node $B$ in a control flow graph is that the control of the program may flow from the basic block $A$ to the basic block $B$. (Usually, edges are annotated with the conditions that control the flow of the program from one basic block to another.) From the graph of Figure 3 however, it is not apparent which basic block will be executed under which condition. The control subgraph of the PDG (CSPDG) of the loop of Figure 2 is shown in Figure 4. As in Figure 3, each node of the graph corresponds to a basic

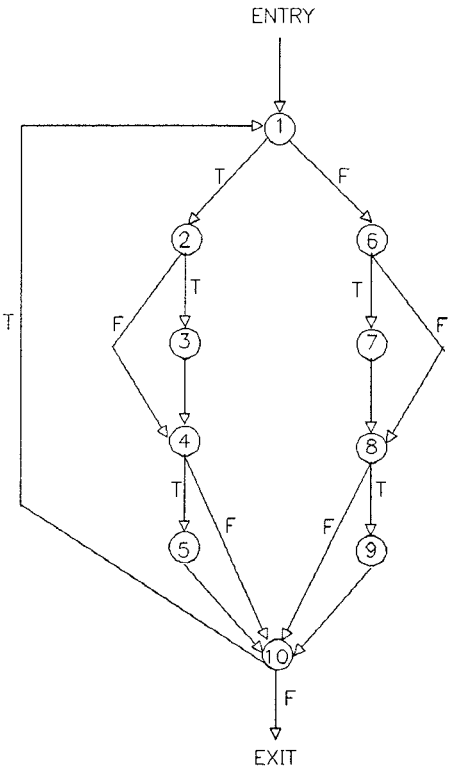



Figure 3. The control flow graph of the loop of Figure 2

block of the program. Here, a solid edge from a node $A$ to a node $B$ has the following meaning:

1. there is a condition *COND* in the end of $A$ that is evaluated to either *TRUE* or *FALSE*, and
2. if *COND* is evaluated to *TRUE*, $B$ will definitely be executed, otherwise $B$ will not be executed.

The control dependence edges are annotated with the corresponding conditions as for the control flow graph. In Figure 4 solid edges designate control dependence edges, while dashed edges will be discussed below. For example, in Figure 4 the edges emanating from BL1 indicate that BL2 and BL4 will be executed if the condition at the end of BL1 will be evaluated to TRUE, while BL6 and BL8 will be executed while the same condition is FALSE.

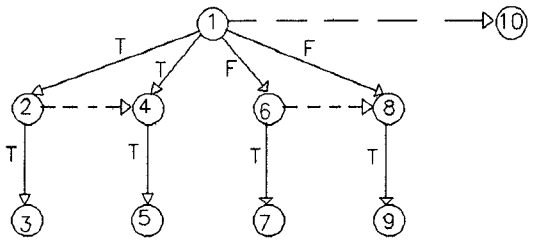



Figure 4. The forward control subgraph of the PDG of the loop of Figure 2

As was mentioned in the introduction, currently we schedule instructions within a single iteration of a loop. So, for the purposes of this type of instruction scheduling, we follow [**CHH89**] and build the *forward control dependence graph* only, i.e. we do not compute the control dependences that result from or propagate through the back edges in the control flow graph. The CSPDG of Figure 4 is a forward control dependence graph. In the following we discuss forward control dependence graphs only. Notice that forward control dependence graphs are acyclic.

The usefulness of the control subgraph of PDG stems from the fact that basic blocks that have the same set of control dependences (like BL1 and BL10, or BL2 and BL4, or BL6 and BL8 in Figure 4) can be executed in parallel up to the existing data dependences. For our purposes, the instructions of such basic blocks can be scheduled together.

Now let us introduce several definitions that are required to understand our scheduling framework. Let $A$ and $B$ be two nodes of a control flow graph such that $B$ is *reachable* from $A$, i.e., there is a path in the control flow graph from $A$ to $B$.

**Definition 1.** *$A$ dominates $B$* if and only if $A$ appears on every path from ENTRY to $B$.

**Definition 2.** *$B$ postdominates $A$* if and only if $B$ appears on every path from $A$ to EXIT.

**Definition 3.** $A$ and $B$ are *equivalent* if and only if $A$ dominates $B$ and $B$ postdominates $A$.

**Definition 4.** We say that moving an instruction from $B$ to $A$ is *useful* if and only if $A$ and $B$ are equivalent.

**Definition 5.** We say that moving an instruction from $B$ to $A$ is *speculative* if $B$ does not postdominate $A$.

**Definition 6.** We say that moving an instruction from $B$ to $A$ requires *duplication* if $A$ does not dominate $B$.

It turns out that CSPDGs are helpful while doing useful scheduling. To find equivalent nodes, we search a CSPDG for nodes that are *identically control dependent*, i.e. they depend of the same set of nodes under the same conditions. For example, in Figure 4, BL1 and BL10 are equivalent, since they do not depend on any node. Also, BL2 and BL4 are equivalent, since both of them depend on BL1 under the TRUE condition. In Figure 4 we mark the equivalent nodes with dashed edges, the direction of these edges provides the dominance relation between the nodes. For example, for equivalent nodes BL1 and BL10, we conclude that BL1 dominates BL10.

CSPDG is useful also for speculative scheduling. It provides "the degree of speculativeness" for moving instructions from one block to another. When scheduling a speculative instruction, we always "gamble" on the outcome of one or more branches; only when we guess the direction of these branches correctly, the moved instruction becomes profitable. CSPDG provides for every pair of nodes the

number of branches we gamble on (in case of speculative scheduling). For example, when moving instructions from BL8 to BL1, we gamble on the outcome of a single branch, since when moving from BL8 to BL1 in Figure 4, we cross a single edge. (This is not obvious from the control flow graph of Figure 3.) Similarly, moving from BL5 to BL1 gambles on the outcome of two branches, since we cross two edges of Figure 4.

**Definition 7.** We say that moving instructions from $B$ to $A$ is *n-branch speculative* if there exists a path in CSPDG from $A$ to $B$ of length $n$.

Notice that useful scheduling is 0-branch speculative.

### 4.2. Data dependences

While control dependences are computed at a basic block level, data dependencies are computed on an instruction by instruction basis. We compute both intrablock and interblock data dependencies. A data dependence may be caused by the usage of registers or by accessing memory locations.

Let $a$ and $b$ be two instructions in the code. A data dependence edge from $a$ to $b$ is inserted into PDG in one of the following cases:

- A register defined in $a$ is used in $b$ (*flow dependence*);
- A register used in $a$ is defined in $b$ (*anti-dependence*);
- A register defined in $a$ is defined in $b$ (*output dependence*);
- Both $a$ and $b$ are instructions that touch memory (loads, stores, calls to subroutines) and it is not proven that they address different locations (*memory disambiguation*).

Only the data dependence edges leading from a definition of a register to its use carry a (potentially non-zero) delay, which is a characteristic of the underlying machine, as was mentioned in Section 2. The rest of the data dependence edges carry zero delays. To minimize the number of anti and output data dependences, which may unnecessarily constrain the scheduling process, the XL compiler does certain renaming of registers, which is similar to the effect of the static single assignment form **[CFRWZ]**.

To compute all the data dependences in a basic block, essentially every pair of instructions there has to be considered. However, to reduce the compilation time, we take advantage of the following observation. Let $a$, $b$ and $c$ be three instructions in the code. Then, if we discover that there is a data dependence edge from $a$ to $b$ and from $b$ to $c$, there is no need to compute the edge from $a$ to $c$. To use this observation, the basic block instructions are traversed in an order such that when we come to determine the dependency between $a$ and $c$, we have already considered the pairs $(a,b)$ and $(b,c)$, for every possible $b$ in a basic block. (Actually, we compute the transitive closure for the data dependence relation in a basic block.)

Next for each pair $A$ and $B$ of basic blocks such that $B$ is reachable from $A$ in the control flow graph, the intrablock data dependences are computed. The observation in the previous paragraph helps to reduce the number of pairs of instructions that are considered during the computation of the intrablock data dependences as well.

Let us demonstrate the computation of data dependences for BL1; we will reference the instructions by their numbers from Figure 2. There is an anti-dependence from (I1) to (I2), since (I1) uses r31 and (I2) defines a new value for r31. There is a flow data dependence from both (I1) and (I2) to (I3), since (I3) uses r12 and r0 defined in (I1) and (I2), respectively. The edge ((I2),(I3)) carries a one cycle delay, since (I2) is a load instruction (delayed

load), while ((I1),(I3)) is not computed since it is transitive. There is a flow data dependence edge from (I3) to (I4), since (I3) sets cr7 which is used in (I4). This edge has a three cycle delay, since (I3) is a compare instruction and (I4) is the corresponding branch instruction. Finally, both of ((I1),(I4)) and ((I2),(I4)) are transitive edges.

It is important to notice that, since both the control and data dependences we compute are acyclic, the resultant PDG is acyclic as well. This facilitates convenient scheduling of instructions which is discussed next.

## 5. The scheduling framework

The global scheduling framework consists of the top-level process, which tries to schedule instructions cycle by cycle, and of a set of heuristics which decide what instruction will be scheduled next, in case there is a choice. While the top-level process is suitable for a range of machines discussed here, it is suggested that the set of heuristics and their relative ordering should be tuned for a specific machine at hand. We present the top-level process in Section 5.1, while the heuristics are discussed in Section 5.2.

### 5.1. The top-level process

We schedule instructions in the program on a region by region basis. In our terminology a *region* represents either a strongly connected component that corresponds to a loop (which has at least one back edge) or a body of a subroutine without the enclosed loops (which has no back edges at all). Since currently we do not overlap the execution of different iterations of a loop, there is no difference in the process of scheduling the body of a loop and the body of a subroutine.

Innermost regions are scheduled first. There are a few principles that govern our scheduling process:

- Instructions are never moved out or into a region.
- All the instructions are moved in the upward direction, i.e, they are moved against the direction of the control flow edges.
- The original order of branches in the program is preserved.

Also, there are several limitations that characterize the current status of our implementation for global scheduling. This includes:

- No duplication of code is allowed (see Definition 6 in Section 4.1).
- Only 1-branch speculative instructions are supported (see Definition 7 in Section 4.1).
- No new basic blocks are created in the control flow graph during the scheduling process.

These limitations will be removed in future work.

We schedule instructions in a region by processing its basic blocks one at a time. The basic blocks are visited in the topological order, i.e., if there is a path in the control flow graph from $A$ to $B$, $A$ is processed before $B$.

Let $A$ be the basic block to be scheduled next, and let $EQUIV(A)$ be the set of blocks that are equivalent to $A$ and are dominated by $A$ (see Definition 3). We maintain a set $C(A)$ of *candidate blocks* for $A$, i.e., a set of basic blocks which can contribute instructions to $A$. Currently there are two levels of scheduling:

1. Useful instructions only: $C(A) = EQUIV(A)$;
2. 1-branch speculative: $C(A)$ includes the following blocks:
   a. the blocks of $EQUIV(A)$;
   b. All the immediate successors of $A$ in CSPDG;
   c. All the immediate successors of blocks in $EQUIV(A)$ in CSPDG.

Once we initialize the set of candidate blocks, we compute the set of *candidate instructions* for $A$. An instruction $I$ is a *candidate* for scheduling in block $A$ if it belongs to one of the following categories:

- $I$ belonged to $A$ in the first place.
- $I$ belongs to one of the blocks in $C(A)$ and:
  1. $I$ belongs to one of the blocks in $EQUIV(A)$ and it may be moved beyond its basic block boundaries. (There are instructions that are never moved beyond basic block boundaries, like calls to subroutines.)
  2. $I$ does not belong to one of the blocks in $EQUIV(A)$ and it is allowed to schedule it speculatively. (There are instructions that are never scheduled speculatively, like store to memory instructions.)

During the scheduling process we maintain a list of *ready instructions*, i.e., candidate instructions whose data dependences are fulfilled. Every cycle we pick from the ready list as many instructions to be scheduled next as required by the machine architecture, by consulting the parametric machine description. If there are too many ready instructions, we choose the "best" ones based on priority criteria. Once an instruction is picked up to be scheduled, it is moved to the proper place in the code, and its data dependences to the following instructions are marked as fulfilled, potentially enabling new instructions to become ready. Once all the instructions of $A$ are scheduled, we move to the next basic block. The net result is that the instructions in $A$ are reordered and there might be instructions external to $A$ that are physically moved into $A$.

It turns out that the global scheduler does not always create the best schedule for each individual basic block. It is mainly due to the two following reasons:

- The parametric machine description of Section 2 does not cover all the secondary features of the machine;
- The global decisions are not necessarily optimal in a local context.

To solve this problem, the basic block scheduler is applied to every single basic block of a program after the global scheduling is completed. The basic block scheduler has a more detailed model of the machine which allows more precise decisions for reordering the instructions within the basic blocks.

### 5.2. Scheduling heuristics

The heart of the scheduling scheme is a set of heuristics that provide the relative priority of an instruction to be scheduled next. There are two integer-valued functions that are computed locally (within a basic block) for every instruction in the code, these functions are used to set the priority of instructions in the program.

Let $I$ be an instruction in a block $B$. The first function $D(I)$, called *delay heuristic*, provides a measure of how many delay slots may occur on a path from $I$ to the end of $B$. Initially, $D(I)$ is set to 0 for every $I$ in $B$. Assume that $J_1, J_2, \ldots$ are the immediate data dependence successors of $I$ in $B$, and let the delays on those edges be $d(I,J_1), d(I,J_2), \ldots$. Then, by visiting $I$ after visiting its data dependence successors, $D(I)$ is computed as follows:

$$D(I) = \max((D(J_1) + d(I,J_1)),(D(J_2) + d(I,J_2)), \ldots)$$

The second function $CP(I)$, called *critical path heuristic*, provides a measure of how long it will take to complete the execution of instructions that depend on $I$ in $B$, including $I$ itself, and assuming an unbounded number of computational units. Let $E(I)$ be the execution time of $I$. First, $CP(I)$ is initialized to $E(I)$ for every $I$ in $B$. Then, again by

visiting $I$ after visiting its data dependence successors, $CP(I)$ is computed as follows:

$$CP(I) = \max((CP(J_1) + d(I,J_1)),$$

$$(CP(J_2) + d(I,J_2)), \ldots) + E(I)$$

During the decision process, we schedule useful instructions before speculative ones. For the same *class* of instructions (useful or speculative) we pick an instruction with has the biggest delay heuristic ( $D$). For the instructions of the same class and delay we pick one that has a biggest critical path heuristic ($CP$). Finally, we try to preserve the original order of instructions.

To make it formally, let $A$ be a block that is currently scheduled, and let $I$ and $J$ be two instructions that (should be executed by a functional unit of the same type and) are ready at the same time in the scheduling process, and one of them has to be scheduled next. Also, let $U(A) = A \bigcup EQUIV(A)$, and let $B(I)$ and $B(J)$ be the basic blocks to which $I$ and $J$ belong. Then, the decision is made in the following order:

1. If $B(I) \in U(A)$ and $B(J) \notin U(A)$, then pick $I$;
2. If $B(J) \in U(A)$ and $B(I) \notin U(A)$, then pick $J$;
3. If $D(I) > D(J)$, then pick $I$;
4. If $D(J) > D(I)$, then pick $J$;
5. If $CP(I) > CP(J)$, then pick $I$;
6. If $CP(J) > CP(I)$, then pick $J$;
7. Pick an instruction that occurred in the code first.

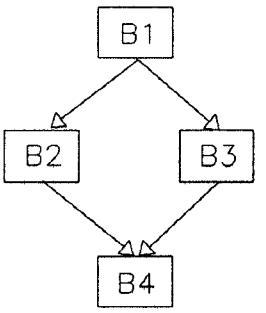
Notice that the current ordering of the heuristic functions is tuned towards a machine with a small number of resources. This is the reason for always preferring to schedule a useful instruction before a speculative one, even though a speculative instruction may cause longer delay. In any case, experimentation and tuning are needed for better results.

## 5.3. Speculative scheduling

In the global scheduling framework, while doing non-speculative scheduling, to preserve the correctness of the program it is sufficient to respect the data dependences as they were defined in Section 4.2. It turns out that for speculative scheduling this is not true, and a new type of information has to be maintained. Examine the following excerpt of a C program:

```
 ...
if (cond) x=5;
else x=3;
printf("x=%d", x);
 ...
```

The control flow graph of this piece of code looks as follows:




Instruction x=5 belongs to B2, while x=3 belongs to B3. Each of them can be (speculatively) moved into B1, but it is apparent that both of them are not allowed to move there, since a wrong value may be printed in B4. Data dependences do not prevent the movement of these instructions into B1.

To solve this problem, we maintain the information about the (symbolic) registers that are *live on exit* from a basic block. If an instruction that is being considered to be moved speculatively to a block $B$ computes a new value for a register that is live on exit from $B$, such speculative movement is disallowed. Notice that this type of information has to be updated dynamically, i.e., after each speculative motion this information has to be updated. Thus, let us say, x=5 is first moved to B1. Then, $x$ (or actually a symbolic register that

```
 ...  more instructions here ...
********** LOOP STARTS ************
    CL.0:
(I1)     L        r12=a(r31,4)
(I2)     LU       r0,r31=a(r31,8)
(I18)    AI       r29=r29,2
(I3)     C        cr7=r12,r0
(I19)    C        cr4=r29,r27
(I4)     BF       CL.4,cr7,0x2/gt
(I5)     C        cr6=r12,r30
(I8)     C        cr7=r0,r28
(I6)     BF       CL.6,cr6,0x2/gt
(I7)     LR       r30=r12
    CL.6:
(I9)     BF       CL.9,cr7,0x1/lt
(I10)    LR       r28=r0
(I11)    B        CL.9
    CL.4:
(I12)    C        cr6=r0,r30
(I15)    C        cr7=r12,r28
(I13)    BF       CL.11,cr6,0x2/gt
(I14)    LR       r30=r0
    CL.11:
(I16)    BF       CL.9,cr7,0x1/lt
(I17)    LR       r28=r12
    CL.9:
(I20)    BT       CL.0,cr4,0x1/lt
********** LOOP ENDS **************
 ...  more instructions here ...
```

Figure 5. The results of applying the useful scheduling to the program of Figure 2

```
 ...  more instructions here ...
********** LOOP STARTS ************
    CL.0:
(I1)     L        r12=a(r31,4)
(I2)     LU       r0,r31=a(r31,8)
(I18)    AI       r29=r29,2
(I3)     C        cr7=r12,r0
(I19)    C        cr4=r29,r27
(I5)     C        cr6=r12,r30
(I12)    C        cr5=r0,r30
(I4)     BF       CL.4,cr7,0x2/gt
(I8)     C        cr7=r0,r28
(I6)     BF       CL.6,cr6,0x2/gt
(I7)     LR       r30=r12
    CL.6:
(I9)     BF       CL.9,cr7,0x1/lt
(I10)    LR       r28=r0
(I11)    B        CL.9
    CL.4:
(I15)    C        cr7=r12,r28
(I13)    BF       CL.11,cr5,0x2/gt
(I14)    LR       r30=r0
    CL.11:
(I16)    BF       CL.9,cr7,0x1/lt
(I17)    LR       r28=r12
    CL.9:
(I20)    BT       CL.0,cr4,0x1/lt
**********  LOOP ENDS **************
 ...  more instructions here ...
```

Figure 6. The results of applying the useful and speculative scheduling to the program of Figure 2

corresponds to $x$ ) becomes live on exit from B1, and the movement of x=3 to B1 will be prevented. More detailed description of the speculative scheduling and its relationship to the PDG-based global scheduling is out of the scope of this paper.

### 5.4. Scheduling examples

Let us demonstrate the effect of useful and speculative scheduling on the example of Figure 2. The result of scheduling useful instructions only to this program is presented in Figure 5. During the scheduling of BL1, the only instructions that were considered to be moved there were those of BL10, since only $BL10 \in EQUIV(BL1)$. The result is that two instructions of BL10 (I18 and I19) were moved into BL1, filling in the delay slots of the instructions there. Similarly, I8 was moved from BL4 to BL2, and I15 was moved from BL8 to BL6. The resultant program in Figure 5 takes 12-13 cycles per iteration, while the original program of Figure 2 was executing in 20-22 cycles per iteration.

Figure 6 shows the result of applying both the useful and the (1-branch) speculative scheduling to the same program. In addition to the motions that were described above, two additional instructions (I5 and I12) were moved speculatively to BL1, to fill in the three cycle delay between I3 and I4. Interestingly enough, since I5 and I12 belong to basic blocks that are never executed together in any

single execution of the program, only one of these two instructions will carry a useful result. All in all, the program in Figure 6 takes 11-12 cycles per iteration, a one cycle improvement over the program in Figure 5.

## 6. Performance results

A preliminary evaluation of the global scheduling scheme was done on the IBM RS/6K machine whose abstract model is presented in Section 2.1. For experimentation purposes, the global scheduling has been embedded into the IBM XL family of compilers. These compilers support several high-level languages, like C, Fortran, Pascal, etc.; however, we concentrate only on the C programs.

The evaluation was done on the four C programs in the SPEC benchmark suite [**S89**]. In the following discussion LI denotes the Lisp Interpreter benchmark, GCC stands for the GNU C Compiler, while EQNTOTT and ESPRESSO are two programs that are related to minimization and manipulation of Boolean functions and equations.

The basis for all the following comparisons (denoted by BASE in the sequel) is the performance results of the same IBM XL C compiler in which the global scheduling was disabled. Please notice that the base compiler includes two types of instruction scheduling on its own (aside of all the possible machine independent and peephole optimizations) as follows:

- a sophisticated basic block scheduler similar to that of [**W90**], and
- a set of code replication techniques that solve certain loop-closing delay problems [**GR90**].

So, in some sense certain improvements due to the global scheduling overlap those of the scheduling techniques that were already part of the base compiler.

Next we describe how the global scheduling scheme was configured so as to exploit the trade-off of the compile-time overhead and the run-time improvement to a maximum extent. The following design decisions characterize the current status of the global scheduling prototype:

- Only two inner levels of regions are scheduled. So, we distinguish between inner regions (i.e., regions that do not include other regions) and outer regions (i.e. regions that include only inner regions).
- Only "small" reducible regions are scheduled. "Small" regions are those that have at most 64 basic blocks and 256 instructions.
- In a preparation step, before the global scheduling is applied, the inner regions that represent loops with up to 4 basic blocks are unrolled once (i.e., after unrolling they include two iterations of a loop instead of one).
- After the global scheduling is applied to the inner regions, such regions that represent loops with up to 4 basic blocks are rotated, by copying their first basic block after the end of the loop. By applying the global scheduling the second time to the rotated inner loops, we achieve the partial effect of the software pipelining, i.e., some of the instructions of the next iteration of the loop are executed within the body of the previous iteration.
- The general flow of the global scheduling is as follows:
  1. certain inner loops are unrolled;
  2. the global scheduling is applied the first time to the inner regions only;
  3. certain inner loops are rotated;
  4. the global scheduling is applied the second time to the rotated inner loops and the outer regions.

The compile-time overhead of the above described scheme is shown in Figure 7. The column marked

BASE gives the compilation times of the programs in seconds as measured on the IBM RS/6K machine, model 530 whose cycle time is 40ns. The column marked CTO (Compile-Time Overhead) provides the increase in the compilation times in percents. This increase in the compilation time includes the time required to perform all of the above mentioned steps (including loop unrolling, loop rotation, etc.).

| PROGRAM | BASE | CTO |
|---|---|---|
| LI | 206 | 13% |
| EQNTOTT | 78 | 17% |
| ESPRESSO | 465 | 12% |
| GCC | 2457 | 13% |

Figure 7. Compile-time overheads for the global scheduling

There are two levels of scheduling that we distinguish at the moment, namely useful only and useful and speculative scheduling. The run-time improvement (RTI) for both types of scheduling is presented in Figure 8 in percents relative to the running time of the code compiled with the base compiler which is shown in seconds. The accuracy of the measurements is about 0.5%-1%.

| PROGRAM | BASE | RTI | |
|---|---|---|---|
| | | USEFUL | SPECULATIVE |
| LI | 312 | 2.0% | 6.9% |
| EQNTOTT | 45 | 7.1% | 7.3% |
| ESPRESSO | 106 | -0.5% | 0% |
| GCC | 76 | -1.5% | 0% |

Figure 8. Run-time improvements for the global scheduling

We notice in Figure 8 that for EQNTOTT most of the improvement comes from the useful scheduling only, while for LI, the speculative scheduling is dominant. On the other hand, for both ESPRESSO and GCC, no improvement is observed. (Actually, there is a slight degradation in performance for both benchmarks, when the global scheduling is restricted to useful scheduling only.)

To summarize our short experience with the global scheduling, we notice that the achieved improvement in run-time is modest due to the fact that the base compiler has already been optimized towards the existing architecture. We may expect even bigger payoffs in machines with a larger number of computational units. As for the compile-time overhead, we consider it as reasonable, especially since no major steps were taken to reduce it except of the control over the size of the regions that are being scheduled.

## 7. Summary

The proposed scheme allows the global scheduling of instructions by an optimizing compiler for better utilization of machine resources for a range of superscalar processors. It is based on a data structure proposed for parallel/parallelizing compilers (PDG), a parametric machine description and a flexible scheduling framework that employs a set of useful heuristics. The results of evaluating the scheduling scheme on the IBM RS/6K machine are quite encouraging. We are going to extend our work by supporting more aggressive speculative scheduling, and scheduling with duplication of code.

**Acknowledgements.** We would like to thank Kemal Ebcioglu, Hugo Krawczyk and Ron Y. Pinter for many helpful discussions, and Irit Boldo and Vladimir Rainish for their help in the implementation.

## References


[BG89] Bernstein, D., and Gertner, I., "Scheduling expressions on a pipelined processor with a maximal delay of one cycle", *ACM Transactions on Prog. Lang. and Systems*, Vol. 11, Num. 1 (Jan. 1989), 57-66.

[BRG89] Bernstein, D., Rodeh, M., and Gertner, I., "Approximation algorithms for scheduling arithmetic expressions on pipelined machines", *Journal of Algorithms*, 10 (Mar. 1989), 120-139.

[BJR89] Bernstein, D., Jaffe, J.M., and Rodeh, M., "Scheduling arithmetic and load operations in parallel with no spilling", *SIAM Journal of Computing*, (Dec. 1989), 1098-1127.

[BEH89] Bradlee, D.G., Eggers, S.J., and Henry, R.R., "Integrating register allocation and instruction scheduling for RISCs", to appear in *Proc. of the Fourth ASPLOS Conference*, (April 1991).

[CHH89] Cytron, R., Hind, M., and Wilson, H., "Automatic generation of DAG parallelism", *Proc. of the SIGPLAN Annual Symposium*, (June 1989), 54-68.

[CFRWZ] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, F.K., "An efficient method for computing static single assignment form", *Proc. of the Annual ACM Symposium on Principles of Programming Languages*, (Jan. 1989), 25-35.

[E88] Ebcioglu, K., "Some design ideas for a VLIW architecture for sequential-natured software", *Proc. of the IFIP Conference on Parallel Processing*, (April 1988), Italy.

[EN89] Ebcioglu, K., and Nakatani, T., "A new compilation technique for parallelizing regions with unpredictable branches on a VLIW architecture", *Proc. of the Workshop on Languages and Compilers for Parallel Computing*, (August 1989), Urbana.

[E85] Ellis, J.R., "Bulldog: A compiler for VLIW architectures", Ph.D. thesis, Yale U/DCS/RR-364, Yale University, Feb. 1985.

[FOW87] Ferrante, J., Ottenstein, K.J., and Warren, J.D., "The program dependence graph and its use in optimization", *ACM Transactions on Prog. Lang. and Systems*, Vol. 9, Num. 3 (July 1987), 319-349.

[F81] Fisher, J., "Trace scheduling: A technique for global microcode compaction", *IEEE Trans. on Computers*, C-30, No. 7 (July 1981), 478-490.

[GM86] Gibbons, P.B. and Muchnick, S.S., "Efficient instruction scheduling for a pipelined architecture", *Proc. of the SIGPLAN Annual Symposium*, (June 1986), 11-16.

[GR90] Golumbic, M.C. and Rainish, V., "Instruction scheduling beyond basic blocks", *IBM J. Res. Dev.*,(Jan. 1990), 93-98.

[GO89] Groves, R.D., and Oehler, R., "An IBM second generation RISC processor architecture", *Proc. of the IEEE Conference on Computer Design*, (October 1989), 134-137.

[HG83] Hennessy, J.L. and Gross, T., "Postpass code optimization of pipeline constraints", *ACM Trans. on Programming Languages and Systems 5* (July 1983), 422-448.

[JW89] Jouppi, N.P., and Wall, D.W., "Available instruction-level parallelism for superscalar and superpipelined machines", *Proc. of the Third ASPLOS Conference*, (April 1989), 272-282.

[L88] Lam M, "Software Pipelining: An effective scheduling technique for VLIW machines", *Proc. of the SIGPLAN Annual Symposium*, (June 1988), 318-328.

[P85] Patterson, D.A., "Reduced instruction set computers", *Comm. of ACM*, (Jan. 1985), 8-21.

[SLH90] Smith, M.D, Lam M.S., and Horowitz M.A., "Boosting beyond static scheduling in a superscalar processor", *Proc. of the Computer Architecture Conference*, (May 1990), 344-354.

[S89] "SPEC Newsletter", Systems Performance Evaluation Cooperative, Vol. 1, Issue 1, (Sep. 1989).

[W90] Warren, H., "Instruction scheduling for the IBM RISC System/6K processor", *IBM J. Res. Dev.*, (Jan. 1990), 85-92.