An Incremental Basic Block Instruction Scheduler

R. Venugopal
email:venug@csa.iisc.ernet.in

Y. N. Srikant
email:srikant@csa.iisc.ernet.in

Computer Science and Automation
Indian Institute of Science
Bangalore - 560012
India

Abstract

Incremental methods for the earlier phases of compilation such as lexical analysis, parsing, semantic analysis and machine-independent optimization have been well studied. In this paper, we describe an approach towards incremental machine dependent optimization, in particular, basic block instruction scheduling. First we describe a simple algorithm for the incremental construction of the dependency directed acyclic graph (DDAG). We then target basic bloock scheduling based on the longest path length criterion. In this context, we apply an existing incremental shortest path algorithm for computing the lengths of the longest paths incrementally. The techniques described have been implemented in the form of an incremental basic block instruction scheduler for the IBM RISC System/6000 processor. Compile-time measurements on a few sample programs indicate that this approach results in significant reduction in the time taken for instruction scheduling when compared with that taken by the non-incremental approach.

Keywords: Integrated programming environments, incremental compilation, instruction scheduling, compiler optimization.

1 Introduction

Incremental compilers play a crucial role in integrated programming environments [Tei81,Sch84]. In these environments, the programmer edits the program text and invokes the compiler from within the environment. A small change may be made to the program and when the compiler is invoked, the programmer expects fast turn-around times in compilation. In other words, the programmer expects that the compilation time corresponding to the small change is also small and that the system will return to the edit mode as quickly as possible. In such a situation, the compiler cannot afford the luxury of recompiling the whole program. It has to identify the affected portion of the code and recompile only that portion of the code using the data structures built up during an earlier pass or during batch compilation (we shall use the terms *non-incremental* and *batch* interchangeably). Such compilers are called incremental compilers and they have been the subject of study for some time now.

A lot of work has been done in incrementalizing the earlier phases of compilation. Incremental lexical analysis and incremental parsing have received attention in [Ghe79,Jal82,Bee91]. [Rep82, Rep83,Rep86] deals with incremental attribute evaluation and semantic analysis. Some work has been done in incremental machine independent optimization [Bur87a,Ryd88,Mar89]. However, incremental approaches for the back-end of a compiler have been very few and far between. There have been some attempts at incremental register allocation [Biv90] and incremental code generation [For85, Tic86] but as far as we know, nothing has been done towards incrementalizing machine dependent optimizations.

The problem that we look at here is specifically basic block instruction scheduling. A weighted DDAG representing the dependency constraints of the instructions in the basic block is constructed in an earlier pass from which a code schedule has been obtained. The programmer now makes a change in the source program. This change gets transmitted by the front-end of the compiler into changes in the DDAG. These changes could be in the form of insertion of nodes, deletion of nodes or modification of nodes (ie. a change in the instruction to which the node corresponds). It is now the function of the incremental scheduler to get the new schedule using as much information computed during the earlier pass as possible.

The reason for there not having been much work in this direction thus far could be the lack of a global framework which captures all machine dependent optimizations. Such a framework exists in the domain of its machine independent counterpart [Ryd88,Mar89]. Hence any attempt towards incremental machine dependent optimization could suffer from the drawback of being limited in scope or being applicable to a certain specific processor.

In this work, we have tried to mitigate this difficulty by identifying some common element which appears in such optimizations. Since we focus on basic block or local instruction scheduling, our aim was to identify something common to the different basic block instruction scheduling algorithms. The technique we have identified is *list scheduling* using the length of the longest paths in the DDAG or the *critical path* method. In this method, the nodes in the DDAG are scheduled in non-increasing order of the lengths of their longest paths to the final node. This approach is an important component in several instruction scheduling algorithms [Gib86,Lam88,Pal90,War90] . Hence we have adapted an existing incremental shortest path algorithm [Ram93] for computing longest paths incrementally with the view that all these instruction scheduling algorithms would benefit from this approach.

We have built an incremental basic block scheduler for the IBM RS6000 processor (however, we emphasize that our work is not processor specific). We measured the time taken to perform basic block scheduling by this incremental scheduler and compared it with the time taken by the corresponding batch scheduler also implemented by us. The incremental scheduler shows significant gains in scheduling time indicating that this is a worthwhile approach particularly in the context of integrated programming environments. This is especially true in the light of the data given in [Ber92]. This work shows that a significant percentage of time is spent on such optimizations by the compiler. If the earlier phases of the compiler have been incrementalized, then it is reasonable to claim that their full benefits can be reaped only if subsequent phases of compilation are also incrementalized. The work reported here is a step in this direction.

The rest of the paper is organized as follows. Section 2 provides the justification for incremental compilation. Section 3 explains why incremental basic block scheduling could prove beneficial. Section 4 defines the notion of bounded incremental computation which is a useful concept for comparing two incremental algorithms. Section 5 gives an overview of the construction of the scheduler. Section 6 describes a simple algorithm to incrementally construct the DDAG corresponding to a given basic block. In section 7 is given the details of the batch computation of the lengths of the longest paths of the nodes of the DDAG and also the various approaches to computing the same incrementally. In section 8, we describe how the schedule itself can be listed using as much of the existing information as possible. In section 9 we give our implementation results. Section 10 summarizes the work and indicates future work. For the sake of completeness, we list Ramalingam's algorithm [Ram93] in the appendix. This is the algorithm that we have applied for incrementally computing the longest path lengths.

2 Why Incremental Compilation?

It is true that machines are becoming faster thus decreasing compilation times. However, this does not decrease the importance of incremental compilation. We submit the following points in support of this view:

- As mentioned in section 1, in the context of integrated programming environments [Tei81, Sch84,Rep84], incremental compilation is very important. When the programmer works within such an environment, he expects fast turn-around times in compilation. Then even a small amount of time gained proves important. It is our view that with increasingly complex software projects, such environments themselves could prove important in increasing the productivity of the programmer.

- One important application of incremental compilation is in debugging with compiled code. When a breakpoint is set in the debugger and the programmer changes the value corresponding to a variable he would like to see the effect of the change made. Then incremental compilation is a good way to get the code compiled with the change so that the results are obtained quickly.

- Incremental compilation, especially optimization, is useful in the debugging of optimized code. Under normal circumstances, when the optimization flag is set while compiling, the debugging flag has to be turned off and vice versa. The reason is that optimization changes the order of statements in the intermediate representation (maybe assembly code) and thus the correspondence with source statements is lost. One side-effect of incremental compilation is that it could help in the debugging of optimized code in the sense that the data structures used for incremental compilation could be tailored to maintain the correspondence between source language statements and the intermediate representation. [Pol86] deals with this aspect of incremental compilation in some detail.

3 The Need for Incremental Basic Block Scheduling

It is our view that incremental basic block scheduling has great significance in the overall objective of high-speed compilation. We submit the following points in support of this view:

- It is our experience that basic blocks are not all that short. This is particularly true of code generated for RISC processors. Such processors have very simple instructions and more complicated ones have to be implemented as a collection of simple instructions. For instance, a simple address calculation for single dimensional arrays may involve the generation of quite a few number of instructions. Calculating the address A[i] involves the multiplication of the index and the addition to the contents of a base register. This itself could result in the generation of a few instructions. For instance, even the IBM RS6000 (which cannot be termed a RISC in the pure sense of the term possessing as it has nearly 170 instructions, many of them quite complex) generates fairly long basic blocks for even simple programs. In the programs that we consider for testing our implementation (the table given in figure 14 in section 9) basic blocks of an average length of 20 instructions are not uncommon. A few of them are even longer. Programs exist for which basic blocks of length 50 instructions are more the norm rather than an exception. Consider an expression such as $a = b + c$, which is mong the simplest that one can think of. An examination of the assembly code generated for the IBM RS6000 reveals that the evaluation of this expression itself involves the generation of instructions to calculate the addresses of a, b and c, instructions to perform the actual addition and instructions to write back the value into the computed address of a. On the other hand, suppose the change made by the programmer is to this expression, converting it to $a = b + c + 1$, then this involves the insertion of just a single instruction (an add increment instruction). Thus the change made when compared with the number of instructions generated for the evaluation of the original expression is small. An incremental scheduler can take advantage of such situations. The case for such a scheduler improves when one considers that more than one such expression could reasonably be expected to be present in the basic block. Also note that the kind of change we considered in this example (the addition of one to the expression) is the kind that one could reasonably expect a programmer to make.

- Let us assume, for the sake of argument, that basic blocks are small in size. It is possible however that the changes made by the programmer get reflected as changes to more than one basic block. Assume that a change has taken place in the form of one instruction in basic block B_1 of length 20 and another, again in the form of one instruction, in a second basic block B_2 of length 20. Then a non-incremental scheduler would do work corresponding to 40 instructions whereas an incremental scheduler (especially if the algorithms incorporated are bounded) would do only as much work as is necessary, viz. work corresponding to just two instructions. Thus while estimating the amount of work that the incremental scheduler does viz-a-viz the non incremental scheduler it is not just the length of a single basic block that comes into the picture but the sum of the lengths of the all the affected basic blocks.

- Scientific code is well-known for producing long basic blocks. So at least for this class of programs, the incremental scheduler should prove beneficial.

- As briefly mentioned in section 1, in [Ber92] is given evidence that basic block scheduling does not take an insignificant amount of time. Among other things, the authors compare the time taken for basic block scheduling as per the algorithm given in [War90] (the longest path length criterion is an important component of this algorithm), branch scheduling as per the algorithm given in [Gol90] and global scheduling as per the algorithm given in [Ber91] on the IBM RS6000 processor. The results are on SPEC benchmarks. Summarizing the results, basic block scheduling takes about 5 per cent of total compile time where all the three optimizations taken together take up 19 per cent of compile time. Thus basic block scheduling takes more than a quarter of the time spent on all the machine dependent optimizations. A further examination of the results in [Ber92] indicates that there are programs for which the time taken for basic block scheduling is one third the time spent on all the optimizations and there is at least one program for which the time taken is as much as half. Even assuming that the time taken for these optimizations, when viewed as absolute numbers, is not high, in the context of integrated programming environments, it makes sense to look at an incremental approach to these problems especially when the other phases of compilation are incrementalized. We stress that our work be viewed in the context of a full fledged incremental compiler where each component is incremental. It is our belief that the time spent on performing machine dependent optimizations in a non-incremental way would be significant when compared with the time taken for doing the earlier phases incrementally. Incremental approaches towards machine independent optimizations have been well studied. This work may be viewed as a logical extension of the same to the machine dependent domain which has been hitherto unexplored.

- Many compilers routinely perform loop unrolling to increase the amount of instruction level parallelism available. Longer basic blocks are obtained as a result of loop unrolling. Constructing traces is another technique used to increase the available instruction level parallelism. Traces may essentially be viewed as long basic blocks. The approach described in our work is applicable to the basic blocks obtained by these methods.

4 Bounded Incremental Computation

When an incremental algorithm is designed for a problem, one needs to evaluate its performance in terms of some complexity measure. In many cases, the traditional measures prove inadequate for the purpose. For instance, using worst case complexity measures might indicate that the incremental algorithm does not perform any better than the corresponding batch algorithm [Ram93]. Hence a different approach is adopted for evaluating incremental algorithms. This approach initially appeared in [Rep84] and later also in [Alp90]. The recent work of Ramalingam [Ram93] deals at length with this approach towards incremental computation.

Let $CHANGE = \Delta_{input} + \Delta_{output}$, where Δ_{input} measures the change in input and Δ_{output} measures the change in output. Then an incremental algorithm is said to be bounded if its complexity can be expressed as $f(CHANGE)$. It is polynomially bounded if f is a polynomial. Similarly, it is exponentially bounded if f is an exponential function. A problem is said to be bounded if it has a bounded incremental algorithm, otherwise it is unbounded. It can be seen that boundedness intuitively characterizes an incremental algorithm by the amount of work that necessarily has to be done for performing the update.

This discussion is relevant to us in the sense that one of the desirable but not mandatory goals of our work is to design polynomially bounded incremental algorithms for our problems. In many cases, this might not be possible as there are inherently unbounded problems [Ram93]. But boundedness is a useful criterion which may be kept in mind when these algorithms are being designed.

5 Overview of the Scheduler

The front-end of the compiler is assumed to produce assembly code for the target processor. In our implementation, the target processor is the IBM RS6000. The assembly code can potentially use an infinite number of registers. In other words, the assumption is that register allocation has not been done. The question of whether instruction scheduling should be performed before or after register allocation is an interesting and crucial one, but it is one which does not directly concern us here. Instead, we sidetrack this issue and assume that instruction scheduling is done before register allocation. This is quite often the case in practice [Gib86,War90]. Possibly, there could be a second pass of instruction scheduling after register allocation. The assembly code is then divided into basic blocks (see [Aho86] for the algorithm to do so). For each block its DDAG is constructed. In the next section, we shall give a detailed description of what we mean by a DDAG. The construction of the DDAG is straightforward in the batch mode. Each pair of instructions in the basic block is examined for dependencies and the DDAG is built. In the incremental mode, we have designed a simple algorithm which uses the DDAG built during an earlier pass (which could be a batch pass) and updates it to reflect the changes that the programmer has made. This algorithm is unbounded in the sense of the definition given in the previous section but quite efficient in practice. The input to the scheduler is this DDAG together with the list of changes that have been made to it. The nodes are scheduled in the order of non-increasing lengths of the longest path of each node to the final node. In the batch mode, the longest path information can be computed bottom-up as the DDAG is being constructed. But, in the incremental mode, this is not possible and this information has to be computed in a separate pass. For this we adapt Ramalingam's algorithm [Ram93]. Finally the scheduled code is emitted. Figure 1 gives the overall scheme of things of the incremental scheduler.

The scheduler schedules each node (basic block) of the flow graph individually since it is a basic block or local scheduler. It is possible that the changes that the programmer makes to the source code are such that they get translated to structural changes in the flow graph. In other words, there could be structural changes affecting the connectivity of the flow graph. However, such changes do not concern us here. Only changes *within* a node of the flow graph are considered.

Like in the case of all online or incremental algorithms, the choice of appropriate data structures

begin

incrementally construct the DDAG
incrementally compute the longest path lengths of each node in the DDAG to the final node
incrementally update the code schedule to get the modified schedule

end

Figure 1: Outline of the incremental scheduler

is crucial to the running times of these algorithms. Hence as we describe each step of the computation process, we shall also highlight the data structures used in our implementation.

6 Construction of the DDAG

For each node or basic block in the flow graph, its corresponding dependency directed acyclic graph (DDAG) is constructed. The DDAG captures the dependency relations between the instructions inside the basic block. It is a weighted directed acyclic graph with each node corresponding to an instruction inside the basic block. An edge exists from node u to node v in the DDAG if the instruction corresponding to node v (i_v) depends on the instruction corresponding to node u (i_u). Instruction i_v depends on i_u if:

- i_u writes a value into a resource which is read by i_v (*flow dependency*).
- i_v writes into a resource which is read by i_u (*anti dependency*).
- Both i_u and i_v write into the same resource (*output dependency*).

The term *resource* has been left deliberately vague in the description above. In general, it can stand for registers, memory locations or even control register (CR) fields. For instance, in the IBM RS6000 processor it represents all three. From here on, when we use the term resource, we shall mean by it one of these three objects - register, CR-field or memory location. Also, we shall interchangeably use node n of the DDAG and the assembly language instruction i_n that n stands for.

Each edge (u, v) of the DDAG is also given a weight w ($w \geq 0$) which represents the latency between i_u and i_v. This depends on the processor architecture and it means that i_v can execute only w units of time after i_u.

The construction of the DDAG in the batch mode is easy. Each pair of instructions in the DDAG is examined to see if any of the three dependency relations (flow, anti or output) holds good between them. Thus $n(n-1)/2$ pairs have to be examined where n is the number of instructions in the basic block. This is a time consuming process. *In fact, it is reported in [War90] that the building of the*

DDAG takes up the major portion of the time of the scheduler. Thus, this approach is not to be recommended when the DDAG has to be updated to reflect the changes made by the programmer. Hence we look for an incremental approach towards constructing the DDAG.

It is easy to see that incremental construction of the DDAG is essentially an incremental data-flow analysis problem. This has received a lot of attention in recent years. See [Bur87a,Ryd88,Mar89]. All these approaches tackle the general problem ie. to construct solutions to data flow problems for the whole flow graph which may include loops also. Thus these are incremental global data flow analysis problems. The method generally adopted in these approaches is to set up equations computing the sets of values required for the solution and to incrementally update the solution to these equations. Since we deal only with one basic block at a time, we do not require this general approach. Instead, we shall describe a simple algorithm which adjusts the nodes and the edges in the DDAG (which has been constructed in the earlier pass and which is now available for the current one) to reflect the changes that have been effected by the programmer.

We now begin the description of the algorithm which incrementally updates the DDAG after changes have been made to the source program by the programmer. This algorithm is similar to the DAG construction algorithm given in [Kri90]. The author in [Kri90], however, does not consider the incremental DDAG construction problem. But the non-incremental construction of the DDAG can be viewed as the incremental construction starting from an empty basic block and progressively adding instructions one at a time and at the same time updating the partially constructed DDAG one node at a time. But we view the problem as the updation of the DDAG after changes have been made to an already constructed basic block. Thus, while the algorithm in [Kri90] can be viewed as handling DDAG construction after the insertion of an instruction, we consider both insertion and deletion of instructions. Also, we analyze our algorithm in terms of the changes that have occurred to the input and the output.

The input to our algorithm is the DDAG *ddag* which has been constructed during an earlier pass, pointers to the nodes in the DDAG which have been newly inserted, *inserted_node_ptrs* and pointers to the nodes in the DDAG which have been deleted, *deleted_node_ptrs*. This lists are assumed to be provided by the front-end of the compiler, desirably an incremental front-end. Considerable work has been done in this area. See [Jal82,Bee91,Rep84,Rep86].

Insertion of a node in the DDAG corresponds to the change whose net effect is the insertion of an assembly language instruction inside the current basic block. Similarly, deletion of a node in the DDAG corresponds to the change whose net effect is the deletion of an assembly language instruction inside the current basic block. Note that a change whose net effect is to modify an assembly language instruction can be modeled as the deletion of the instruction before modification followed by the insertion of the modified instruction. Insertion and deletion of edges between two existing nodes or the modification of the latency of edge (u, v) where u and v already exist do not occur in the cases that we consider. Hence our algorithm need handle only the two cases: deletion of nodes and insertion of nodes in the DDAG. In addition to these inputs, the algorithm takes as input *defn_lists* which is an array of lists indexed by register number, CR field number or memory location address and *uses_lists* which again is an array of lists indexed in a similar way. *defn_lists* has a list for each register or CR field or memory location and is a list of those instructions in the basic block which define the corresponding register, CR field or memory location. For example, *defn_lists[r]* is the list of those instructions in the basic block that define r. Each element in the list is a node which contains a pointer to the corresponding node in the DDAG. Similar to this is *uses_lists* except for the fact that the instructions included *use* the value in the corresponding register, CR field or memory location. For example, *uses_lists[r]* is the list of those instructions in the basic block that use register r. Again each node in the list contains a pointer to the corresponding node in the DDAG. The algorithm produces as output the modified DDAG and the updated lists, *defn_lists* and *uses_lists*.

The algorithm is listed in figure 2.

The algorithm in figure 2 starts off by considering the deleted nodes. For each deleted node it gets the value defined by the instruction corresponding to that node. This is done in line number 5. The value defined could be the value inside a register, a CR field in the IBM RS6000 or a memory location. If a value is defined, it is used to index into the array *defn_lists* to get the list which stores pointers to all those DDAG nodes whose corresponding instructions define the same resource. The following fact may be kept in mind here: since an infinite number of registers has been assumed, for registers, this list will consist of a single node. This is because each new definition goes into a new register. But for CR fields, this is not the case. If such a DDAG node is obtained, then it is used to adjust the flow dependency edges in the procedure *del_flow_dependency()* and the output dependency edges in the procedure *del_output_dependency()*. A description of these procedures will be given shortly. In a similar fashion, the value defined by the deleted instruction is used to index into *uses_lists* to get the list of uses of the same resource. This is accomplished in line number 14. Again, because of our assumption of a infinite number of registers, such a list would consist of a single node in the case of registers but not in the case of CR-fields or memory locations. Using the list of uses and their corresponding DDAG nodes, the anti dependency edges are adjusted in the procedure *del_anti_dependency()* which will be described shortly and which is invoked in line number 16. Now, the node corresponding to the current deleted instruction is deleted from its list in the array *defn_lists* in line number 17 and the nodes corresponding to the uses of the current deleted instruction are deleted from their lists in the array *uses_lists* in line number 19. The node itself is

Input: ddag, defn_lists, uses_lists, inserted_node_ptrs, deleted_node_ptrs.
Output: modified ddag, updated defn_lists, updated uses_lists.
variables

```
        n, p_d, f : ↑dag_node; v : integer
        p_u : array of ↑dag_node; current_defn_ptr : ↑defns
begin
        while more deleted nodes remain do
        begin
                n := current_deleted_node(deleted_node_ptrs)
                v := value_defined(n)
                if v exists then
                begin
                        p_d := previous_defn_node(defn_lists[v], n, current_defn_ptr)
                        if p_d exists then
                        begin
                                del_flow_dependency(n, p_d)
                                del_output_dependency(n, p_d)
                        end
                        p_u := previous_uses_nodes(uses_lists[v], n, current_defn_ptr)
                        if p_u is not empty then
                                del_anti_dependency(n, p_u)
                        delete_from_defn_list(defn_lists[v], current_defn_ptr)
                end /* if */
                delete_from_uses_list(uses_lists, n)
                delete_node(n)
        end /* while */
        insert_defn_lists(inserted_node_ptrs, defn_lists)
        insert_uses_lists(inserted_node_ptrs, uses_lists)
        while more inserted nodes remain do
        begin
                n := current_inserted_node(inserted_node_ptrs)
                v := value_defined(n)
                if v exists then
                begin
                        p_d := previous_defn_node(defn_lists[v], n, current_defn_ptr)
                        if p_d exists then
                        begin
                                ins_flow_dependency(n, p_d)
                                ins_output_dependency(n, p_d)
                        end
                        p_u := previous_uses_nodes(uses_lists[v], n, current_defn_ptr)
                        if p_u is not empty then
                                ins_anti_dependency(n, p_u)
                end /* if */
                add_edges_to_ins_instn(defn_lists, n)
                if ((v does not exist) or (p_d does not exist)) then
                begin
                        Let f be the final node
                        add_edge(n, f)
                end
        end /* while */
end
```

Figure 2: Algorithm to construct DDAG incrementally

deleted from the DDAG structure in line number 20. In our implementation, this is accomplished very simply. The DDAG is represented as a hash table and deleting a node merely involves resetting an *occupied* field corresponding to the node to zero. This automatically deletes all the edges coming into this node since the convention adopted is that for an edge to be a valid (existing) one, the node to which it points should have the *occupied* field set to one. All edges emanating from the node are also deleted since they emanate from a node whose *occupied* field has been reset to zero. This concludes the steps taken by the algorithm to handle a deleted node.

For each inserted node, the algorithm follows the broad outline of the steps taken for a deleted node. Of course, the details of the adjustment of the edges of the DDAG do differ. In line 22, the newly inserted nodes have their corresponding nodes in *defn_lists* inserted, In line 23, a similar process is carried out with respect to *uses_lists*. The algorithm follows a similar route to the one taken while handling deleted nodes except for the procedures *ins_flow_dependency()* in line 33, *ins_output_dependency()* in line 34 and *ins_anti_dependency()* in line 38. Let i_u be the instruction corresponding to the current inserted node u. Then *add_edges_to_ins_instn()* in line 40 adds flow dependency edges which are incoming to node u, one for each use in instruction i_u. Then, in line 44, an edge is inserted from the current node to the final node of the DDAG under the assumption that any node of the DDAG has to be scheduled before the final one.

We now proceed to detail the various procedures called by the procedure in figure 2. In figure 3 is given the procedure *previous_defn_node()*.

The procedure in figure 3 takes as its first argument a list of definitions - one of the lists in the array of lists, *defn_lists*. The second argument n is a pointer to a node of the DDAG. The third argument *current_defn_ptr* stores in it a pointer to that node in the list of definitions which corresponds to the DDAG node defining the same resource as n and the last such node to do so before n in the code schedule. This pointer is assigned by the procedure in figure 3 and in this sense is one of the outputs of the procedure obtained as a sort of side-effect. This pointer is subsequently required by the algorithm in figure 2. The procedure in figure 3 is self-explanatory and so we desist from elaborating on it here. It is merely a sequential scanning of the list given by the first argument until the appropriate node is found.

Procedure *del_flow_dependency()* is given in figure 4.

Procedure *del_flow_dependency()* adjusts the flow dependent edges in the DDAG with respect to a deleted node n. p_d is that node of the DDAG corresponding to the last instruction before n which defines the same resource as n does (remember our convention that we do not distinguish between a node n and instruction i_n which is represented by n in the DDAG). In line 2 each flow dependent edge emanating from n is taken up. Assume that (n,x) is one such edge. When n gets deleted, it is

```
dag_node ↑previous_defn_node(defn_lists_v : ↑defns, n : ↑dag_node, current_defn_ptr : ↑defns)
variables
            small_node_ptr, t : ↑defns
            small_label : label_type
(1) begin
(2)         small_node_ptr := NULL
(3)         small_label := 0
            /* or some other minimum value depending on the type of small_label */
(4)         t := defn_lists_v
(5)         while (t ≠ NULL) do
(6)         begin
(7)                 if ((less(t→label, n→label)) and (greater(t→label, small_label))) then
(8)                 begin
(9)                         small_label := t→label
(10)                        small_node_ptr := t
(11)                end
(12)                if (t→label = n→label) then
(13)                        current_defn_ptr := t
(14)                t := t→next
(15)        end /* while */
(16)        return small_node_ptr→ddag_ptr
(17) end
```

Figure 3: Procedure previous_defn_node()

clear that this edge must now emanate from p_d since that is now the node corresponding to the last instruction before x defining the same resource as n. Therefore, in line 4, this new edge emanating from p_d is added and its latency is marked in line 5. At the same time, edge (n,x) is deleted from the DDAG.

The next procedure is *del_output_dependency()*. This is given in figure 5. This procedure is very similar to *del_flow_dependency()* and so we do not elaborate on it.

Now we come to procedure *previous_uses_nodes()* in figure 6 which is similar in action to procedure *previous_defn_node()* given in figure 3.

In figure 7 is given procedure *del_anti_dependeny()*. This procedure is very similar to procedure *del_flow_dependency()* given in figure 4 and procedure *del_output_dependency()* given in figure 5.

Procedure *delete_from_defn_list()* is just straightforward deletion of a node pointed to by *current_defn_ptr* (the second argument passed) from the linked list *defn_lists_v* (the first argument passed).

Let i_n be the instruction corresponding to node n. Then procedure *delete_from_uses_list()* in figure 8 deletes the nodes corresponding to uses in i_n from the lists in *uses_lists*.

Procedure *delete_list()* consists of just sequential searching for the node given by the second argument in the list given by the first argument. Procedure *delete_node()* involves the resetting of a

```
del_flow_dependency(n, p_d : ↑dag_node)
variables
              x : ↑dag_node
(1) begin
(2)           for each flow dependent edge (n,x) in DDAG do
(3)           begin
(4)                     add (p_d,x) to DDAG
(5)                     modify_latency(p_d,x)
(6)                     delete (n,x) from DDAG
(7)           end
(8) end
```

Figure 4: Procedure del_flow_dependency()

```
del_output_dependency(n, p_d : ↑dag_node)
variables
              x : ↑dag_node
(1) begin
(2)           for each output dependent edge (n,x) in DDAG do
(3)           begin
(4)                     add (p_d,x) to DDAG
(5)                     modify_latency(p_d,x)
(6)                     delete (n,x) from DDAG
(7)           end
(8) end
```

Figure 5: Procedure del_output_dependency()

field called *occupied* to zero in the structure corresponding to node n which is the argument of the procedure. Procedure *insert_defn_lists()* is just the straightforward appending of the nodes pointed to by the elements of the first argument (which is an array of pointers) to the various lists given by the second argument. Procedure *insert_uses_lists()* is similar to *insert_defn_lists()*, only that the array of lists to which the appending is done is given by *uses_lists* and the appending takes place for each source operand in the instruction corresponding to the DDAG node.

We now come to procedure *ins_flow_dependency()* in figure 9.

This procedure is similar to procedure *del_flow_dependency()*. Here again p_d is the node corresponding to the instruction which is the last definition before n of the same resource defined by n. Hence it is clear that since n is a newly inserted node, all the flow dependent edges emanating from p_d and going beyond n (x > n) ie. terminating at a node which is after n in the code sequence should now emanate from n. This is accomplished in line 4. The latency is modified in line 5. The edges emanating from p_d are deleted in line 6. An edge is inseted from n to f in line 10 because, as

```
previous_uses_nodes(uses_lists_v : ↑uses, n : ↑dag_node, current_defn_ptr : ↑defns)
variables
            u : ↑uses
            p_u : array of ↑dag_node
begin
            p_u := NULL
            u := uses_lists_v
            while (u ≠ NULL) do
            begin
                        if ((greater(u→label, current_defn_ptr→label)) and (less(u→label, n→label))) then
                                    append(p_u, u→ddag_ptr)
                        u := u→next
            end /* while */
            return p_u
end
```

Figure 6: Procedure previous_uses_nodes()

```
del_anti_dependency(n, p_u : array of ↑dag_node)
variables
            u, x : ↑dag_node
begin
            for each u ∈ p_u do
                        for each output dependent edge (n,x) in DDAG do
                        begin
                                    add (u,x) to DDAG
                                    modify_latency(u,x)
                                    delete (n,x) from DDAG
                        end
end
```

Figure 7: Procedure del_anti_dependency()

mentioned before, every node in the DDAG is assumed to precede the final node in the schedule.

Procedure *ins_output_dependency()* in figure 10 is very similar to that in figure 9 and is self-explanatory.

Procedure *ins_anti_dependency()* in figure 11 is a little more complicated. It takes as its first argument the newly inserted node n. Let i_n be the instruction corresponding to node n. The second argument p_u is the list of uses of the same resource defined by i_n and such that these uses appear in instructions before i_n in the code sequence. Let u be one such node which is an element of p_u. Then for each anti dependent edge emanating from u and terminating in a node x such that i_x comes after i_n in the code sequence, an edge (n,x) is added to the DDAG. The latency of this edge is marked and the edge emanating from u is deleted. Anti dependent edges can emanate from n also. The steps to

```
delete_from_uses_list(uses_lists : array of ↑uses, n : ↑ dag_node)
variables
        u : array of integer
        v : integer
begin
        u := values_used(n)
        for each value v ∈ u do
                delete_list(uses_lists[v],n)
end
```

Figure 8: Procedure delete_from_uses_list()

```
ins_flow_dependency(n, p_d : ↑dag_node)
variables
        x, f : ↑dag_node
begin
        for each flow dependent edge (p_d,x) in DDAG such that (x > n) do
        begin
                add (n,x) to DDAG
                modify_latency(n,x)
                delete (p_d,x) from DDAG
        end
        Let f be the final node of the DDAG
        if (x ≠ f) then
                add_edge(n,f)
end
```

Figure 9: Procedure ins_flow_dependency()

handle this appear in line numbers 10 to 15 and are self explanatory.

Complexity of the incremental DDAG construction algorithm: We assume that the DDAG is maintained in the form of an adjacency list structure. For each node, the nodes adjacent to it are maintained in the form of an unsorted list of nodes. Note that the DDAG is essentially a set of nodes and the operations that one requires to perform on this set are insertion, deletion, finding of a node etc. and these operations are to be performed in a bounded fashion (see section 4). This problem is very similar to the problem of dynamic insertion, deletion etc on a set and sophisticated algorithms with good worst-case as well as amortized complexity are known [Sas76,Yel90]. But we do not use these approaches in our implementation mainly because the simple data structures and techniques that we use seem to yield good results in practice. We also point out that in the basic blocks we consider, it is possible to set a bound on the degree of a node in the DDAG and hence the length of the adjacency list is bounded by this constant. Earlier in this section, we mentioned that the DDAG is maintained as a hash table. There is no contradiction here. The header nodes of the

```
ins_output_dependency(n, p_d : ↑dag_node)
variables
        x : ↑dag_node
begin
        for each output dependent edge (p_d,x) in DDAG such that (x > n) do
        begin
                add (n,x) to DDAG
                modify_latency(n,x)
                delete (p_d,x) from DDAG
        end
end
```

Figure 10: Procedure ins_output_dependency()

adjacency lists are maintained as a hash table. The hashing function used is a function of the opcode of the instruction for which the node stands. This permits efficient insertion, deletion and finding of a node. Implementation results show that collisions do not occur often and if they do occur, they are handled by a simple linear hashing scheme where we insert the colliding node into the next empty bucket in the hash table.

The data structures used for *defn_lists* and *uses_lists* are maintained as unsorted linked lists of nodes. Again, the operations that have to be performed on these lists are insertion, deletion and finding of a node and these operations have to be performed incrementally. This problem is quite similar to the dynamic list updating problem for which a number of efficient algorithms are known [Riv76,Sle85]. Again, we do not use these techniques. The reason is that since we assume that an infinite number of registers are available, the lengths of these lists for registers is at most one. For lists indexed by CR fields or memory locations, the length could be greater but again, in practice, we find that their lengths are fairly small so that our simple approach does yield good results in practice. What does happen, however, on account of the discussion above, is that our algorithm is not bounded in the sense of the definition given in section 4.

It is easy to see that the algorithm in figure 2 has a worst-case complexity of O(n$|Modified|$) where *Modified* is the set of inserted and deleted nodes and n is the number of instructions in the basic block, or equivalently, the number of nodes in the DDAG. More sophisticated data structures can bring the complexity down to O($\log n$ $|Modified|$) (for instance, using a heap or binary tree instead of lists for *defn_lists* and *uses_lists*).

As an aside we mention that, like registers, CR fields can also be considered infinite in number. This would result in reducing the length of the lists used in the incremental DDAG construction for CR fields to one. Of course, then the allocation of these CR fields should be done later just as

```
ins_anti_dependency(n, p_u : array of ↑dag_node)
variables
        x : ↑dag_node
        a : array of integer
        v : integer
        p_d, curr_defn_node : ↑defns
begin
        for each u ∈ p_u do
                for each anti dependent edge (u,x) in DDAG such that (x > n) do
                begin
                        add (u,n) to DDAG
                        modify_latency(u,n)
                        delete (u,x) from DDAG
                end
        a := value_used(n)
        for each v ∈ a do
        begin
                p_d := previous_defn_node(defn_lists_v, n, current_defn_node)
                for output dependent edge (p_d,x) in DDAG do
                        add (n,x) to DDAG
        end
end
```

Figure 11: Procedure ins_anti_dependency()

registers are allocated. We did not consider this possibility as the usual procedure is to just allocate registers and in any case the lengths of these lists are not large.

7 Computation of Lengths of Longest Paths

As we mentioned in section 1, the scheduling technique that we focus upon is list scheduling using the length of the longest path from a node to the final node as the criterion. The DDAG is constructed first, either in the batch mode or in the incremental mode as described in the previous section. The DDAG has latencies (≥ 0) marked on the edges. At each step a list of ready nodes is constructed. Initially, the list consists of those nodes which do not have any predecessors in the DDAG. The node whose length of its longest path to the final node is the greatest among all the ready nodes is scheduled. This makes available for scheduling more nodes which are put in the ready list. The above process is repeated. The length of the longest path is obtained by summing the latencies of the edges along a path from the node to the final node. Each node appearing on the path also contributes a unit to this length since it is assumed that a node takes a unit of time for execution. The length of the longest path of the node is the maximum length among all these paths.

In practice, most instruction scheduling algorithms do not rely on just this one criterion of the lengths of the longest paths. the longest path length criterion may be used just as a secondary heuristic. Other criteria are used to break ties between nodes like scheduling that node with greater number of children, scheduling the one which reduces register-pressure etc. The question might be justifiably asked: why concentrate on the longest path length heuristic? Let us state the reasons with some emphasis here:

- Instruction scheduling algorithms differ widely in the nature of input they consider, the machine architecture that they optimize and in the heuristics that they use. Thus if we had concentrated on all the heuristics that are used in a particular algorithm, our incremental approach would have been specific to that algorithm. We thought it better to make our approach broad-based so that more than one algorithm could benefit by our approach.

- As far as we could discern, the longest path length criterion is one of the few heuristics which appears in more than one instruction scheduling algorithm particularly of the list scheduling variety. Thus the algorithms given in [Gib86,Kri90,War90] all use this heuristic as *one of the* scheduling criteria. By concentrating on this heuristic we provide an incremental solution to an important problem that would appear if one were designing incremental versions of all these algorithms.

- Computing the longest path length information incrementally is by itself an interesting algorithmic problem. It is important to note that one way of doing this is to just consider this as an application of the *incremental circuit value annotation* problem [Alp90,Ram93]. But it has been proved in [Alp90] that this problem has an exponential lower bound at least with respect to the locally persistent model of computation. The other approach is the one that we have taken in this paper. We note that incremental longest path length computation can be achieved as solution in the fixed-point computation of a collection of swsf equations (an swsf equation will be defined later in this section). As a result we are provided with a bounded incremental algorithm of polynomial complexity. The algorithm for the incremental swsf problem is from [Ram93], but we feel that the observation that it can be used for our purposes and in the setting of incremental basic block scheduling an important and interesting one.

We shall now examine Warren's algorithm [War90] and see if the other heuristics used in this algorithm can be incrementalized. This is a list scheduling algorithm specifically targeted for the IBM RS6000, the processor we have used in our implementation. It turns out that the information that needs to be computed incrementally for the other heuristics can be done so trivially. For instance, one of the heuristics schedules that node which has the minimum *liveness weight*. This information just depends on the type of instruction being scheduled. Thus for a newly inserted instruction, all one needs to know is the type of the inserted instruction and the information can be updated. Deletion causes no problems. Similarly, the information associated with the other heuristics can be updated. Thus, at least, with respect to Warren's algorithm the longest path length problem appears to be the most interesting and difficult one.

In the batch mode, one does not require an additional pass to compute the longest path lengths. We assume that the DDAG in the batch mode is constructed bottom-up starting from the final node. Then the length of the longest path of a node n is given by:

$$lp(n) = \max_{s \in successor(n)} lp(s) + d(n,s) + 1 \tag{1}$$

Here $lp(n)$ is the length of the longest path from n to the final node and d(n,s) is the latency marked on edge (n,s).

However, in the incremental mode, the DDAG is updated incrementally. Nodes are inserted and deleted and so the lengths of only some of the nodes are affected. The problem then is to compute the lengths of just these nodes or at least to compute the lengths in a polynomially bounded way.

In the context of the computation of the lengths of the longest path, a modified node is one which meets one or more of the following criteria:

- An edge emanating from it has been newly inserted.
- An edge emanating from it has been deleted.
- The node is a newly inserted one.
- The latency associated with an edge emanating from it is changed.

The last condition above does not really arise in the cases that we consider, since it is difficult to think of a change that can modify the latency associated with an already existing edge. In any case, it can be considered to be a combination of the first two conditions as follows: the edge with the old latency is deleted and this is followed by the insertion of the edge with the new latency.

To incrementally compute the lengths of the longest paths, a naive algorithm immediately suggests itself. It is listed in figure 12.

The algorithm in figure 12 is a simple one and is easy to understand. The catch is in line 4. In what order does one remove nodes from *modified*? Consider the reverse topological order ie. first consider the node which has no modified node as its successor; then consider a node which has no modified node as its successor or if it has, it is a node for which an evaluation of the longest length has already been done and so on. If the modified nodes are considered in this order in line 4, then in one pass over them the new and correct lengths are computed. [Alp90] gives an algorithm for

```
naive_longest_paths(modified : array of ↑dag_node)
variables
            s, n : ↑dag_node
begin
            while modified is not empty do
            begin
                        remove node n from modified
                        lp(n) := max_{s∈successor(n)} lp(s) + d(n, s) + 1
                        add the predecessors of n to modified
            end
end
```

Figure 12: Procedure naive_longest_paths()

incrementally computing the topological order in time $O(\gamma^2 \log \gamma)$ where γ is the size of the change in this order or the number of nodes for which this order is affected. But the point is that γ can, in general, be totally unrelated to the number of nodes for which the lengths of the longest paths has changed. Thus, it is clear that using this topological ordering algorithm can result in an unbounded incremental algorithm for computing the lengths of the longest paths.

Consider the alternative. We evaluate the modified nodes in some arbitrary order. This leads to some of the nodes acquiring spurious values the effect of which could, in the worst case. percolate to all the nodes of the DDAG. Thus, this would again result in an unbounded algorithm.

In our implementation, we have chosen to adapt the algorithm in [Ram93] to our needs, because doing so gives us a bounded algorithm for our problem. The algorithm in [Ram93] is actually one for incrementally computing the fixed-point of a collection of SWSF equations (these equations will be defined below) as a special case of which one can incrementally compute the shortest paths to a single sink in the graph. This is the reason for our referring to this method as an incremental shortest path algorithm in the previous sections. Our use of this algorithm will be to calculate incrementally the longest path lengths of the nodes of the DDAG.

We now define the following term. The definition has been taken from [Ram93].

Let (D,$\sqcap$) be a semi-lattice. A function g : $D^k \longrightarrow$ D is a *strict weakly superior function (swsf)* if it is monotone non-decreasing in each variable and if for every i $\in$ [1,k]

$$g(x_1, \ldots, x_i, \ldots, x_k) \leq x_i \Rightarrow g(x_1, \ldots, x_i, \ldots, x_k) = g(x_1, \ldots, \infty, \ldots, x_k) \quad (2)$$

The significance of the above definition with respect to our problem is as follows. Let lp(n) stand for the length of the longest path from a node to the final node of the DDAG. Then

$$lp(n) \quad = \quad 1, \quad \text{if n is the final node} \quad (3)$$

$$= \quad max_{s \in successor(n)} lp(n) + d(n,s) + 1, \quad \text{for n any other node of the DDAG} \tag{4}$$

Consider a collection Q of equations in k unknowns $x_1, \ldots, x_k$ of the form

$$x_i = g_i(x_1, \ldots, x_k) \tag{5}$$

This equation is said to be an *swsf equation* if g_i is an swsf. The swsf fixed-point problem is to compute the unique fixed-point of a collection of swsf equations.

From the definition of an swsf as given in equation 2 it can be seen that the right hand side of equation of 4 is an swsf. This is because the value of this function is always greater than the value of each one of its arguments. Hence the left side of equation 2 is always false thus making the entire implication in that equation always true. Thus the equation defined by 4 constitutes an swsf equation and hence the incremental computation of the longest path length information can be expressed as the incremental computation of the fixed-point of a collection of swsf equations.

In [Ram93], Ramalingam gives an algorithm for the computation of the fixed-point of a collection of swsf equations in an incremental fashion. He views the equations in the form of a dependence graph (V, E) where $V = \{x_i | 1 \leq i \leq k\}$ and $E = \{x_j \longrightarrow x_i |\ x_j$ occurs in the rhs expression of the equation for $x_i\}$. The algorithm is bounded and it can handle multiple heterogeneous changes to the dependence graph ie. more than one change of different kinds like deletion of a node, insertion of a node, deletion of an edge, insertion of an edge etc. The handling of multiple heterogeneous changes is very important as it prevents the effects of spurious changes in the values at the nodes of the DDAG from being transmitted to potentially all the nodes of the DDAG. Also the kind of changes that can occur to the DDAG in our case is certainly of the multiple heterogeneous kind. Thus the algorithm is well-suited to our application and we have already shown that the computation of the lengths of the longest paths can be expressed as an swsf equation. For the sake of completeness, we include a description of Ramalingam's algorithm in the appendix. This description is almost verbatim taken from [Ram93].

The complexity of Ramalingam's algorithm is O$(|\delta|(log|\delta| + M_\delta))$. Here δ measures the size of the modified set. Strictly speaking, it is the size of the modified set and also the vertices one edge (or one hop) away from these vertices. But for graphs whose degree can be bounded by a constant, this does not disturb the boundedness of the algorithm and these are all the graphs one encounters in practice. m_δ is the total time required to compute the function associated with the vertices.

In our implementation of the algorithm, we have used a simple balanced binary tree structure for the heap (see the appendix). More sophisticated data structures may be used (for instance Ramalingam talks of a relaxed heap implementation) but we find that the one we have used gives good results in practice in addition to being simple.

We shall conclude this section with the following remark. In our work, there is no assumption that the latency has to be greater than zero. Assuming zero latency does not in any way affect the techniques presented in this paper. However, there is a situation where assuming zero latencies can create a problem in the incremental updation of the longest path information. With zero latencies and without the assumption of unit execution time for each instruction, the incremental longest path length computation equations (equations 3 and 4 in section 5) would no longer be swsf equations. However, we are saved by the assumption that each instruction takes unit time for execution. It is not clear whether the longest path length information can be computed incrementally in a bounded and efficient manner if we were to drop this assumption.

8 Listing the Code Schedule

The code schedule generated during the earlier pass is available. Now some nodes from the DDAG have been deleted and some inserted (for reasons explained in the previous sections, these are the two cases that arise in practice). Using the algorithm of the previous section, the new lengths of the nodes have been computed. Using these lengths and list scheduling the new code schedule has to be produced. It is possible to do this by carrying out a pass over the nodes. But, since we have the old schedule and the instructions that have to be deleted and those that have to be inserted, it is worthwhile to explore whether the old schedule itself can be updated to get the new one.

Deletion of instructions causes no problems. One can go ahead and just delete them. It is not difficult to see that this would still be a valid schedule obtained by list scheduling using the lengths of the longest paths of the nodes.

Insertion of instructions is slightly more involved. One has to insert it in the proper slot in the sorted sequence of instructions. Sequential searching for this slot would result in an unbounded algorithm for this problem.

To get around this, we adopt the data structure shown in figure 13. Each node in the DDAG has a pointer to its corresponding instruction in an array. The array is indexed from 1 to the maximum value of the longest path length that any node can possibly have. Our studies show that this value for a given basic block is not large. Hence this array is moderately sized. Nodes which have their longest path lengths equal to l are slotted in the l^{th} location of the array. All those nodes which have longest path lengths equal to l are slotted to the l^{th} location of the array and are chained together (although this is not explicitly shown in the figure). The occupied slots themselves are all linked together by doubly-linked pointers to form a chain of instructions so that an instruction following a given instruction or an instruction preceding a given instruction can be found immediately.

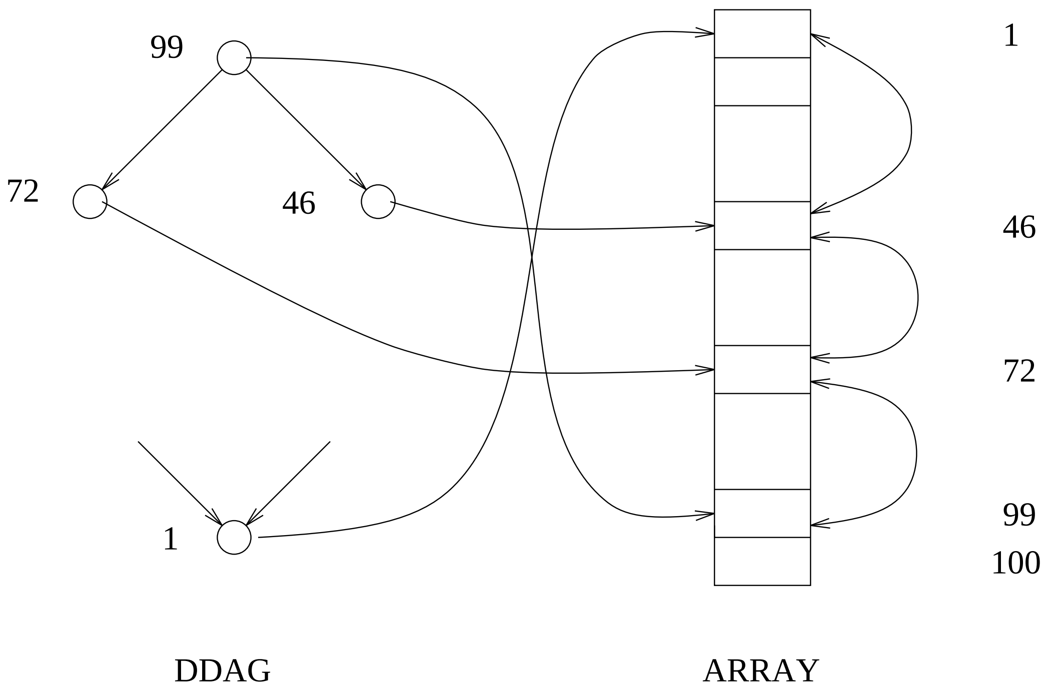

Figure 13: Data structure for listing the code schedule

Now deleting an instruction and inserting an instruction in an already occupied slot are straightforward. When an instruction is inserted in a currently vacant slot, then it is required to search back in the array to find the instruction immediately preceding the currently inserted one so that the pointer chain can be readjusted. Similarly, it is required to search forward to find the instruction immediately succeeding the currently inserted one. But the number of slots that one would have to traverse in either case is at most the maximum latency that the machine architecture possesses which can be reasonably assumed to be a constant. Thus this strategy results in a bounded algorithm for listing the code schedule.

9 Implementation Results

We have implemented the instruction scheduler that works both in the batch mode and the incremental mode. The implementation was done on an IBM340 workstation and the scheduler was targeted to the IBM RS6000 processor. In previous sections, we have indicated the data structures that we used in the implementation. The incremental scheduler uses Ramalingam's algorithm for recomputing the longest path lengths. The results of the implementation are summarized in figure 14. The batch scheduler was first run on four programs listed in the first column of figure 14 (which together comprise 45 basic blocks) one basic block at a time and the time taken for producing the schedule was measured (T_b). The time taken to construct the data structures like *defn_lists* and *uses_lists*

Program	Number of basic blocks	T_b	Percentage change	T_i	Fraction of time (F)
Cholesky decomposition	10	310.59	7.6	28.52	9.1
Shell sort	16	336.45	15.1	52.82	15.6
Polynomial evaluation	14	606.50	2.6	20.17	3.3
Special LU decomposition	5	70.13	2.0	2.24	3.1

Figure 14: Implementation results

which are used by the incremental scheduler was not charged to the batch scheduler. Then random changes were made to these programs and the incremental scheduler was run on these programs one basic block at a time. The time taken for the incremental scheduler to produce the modified schedule was measured (T_i). In each case, as can be seen from figure 14, the percentage of the fraction of scheduling time (F) taken by the incremental scheduler with respect to the batch scheduler (as given in the last column) was found to be small. It is also interesting to note that, in most cases, the time taken for recomputation of the schedule by the incremental scheduler is roughly proportional to the amount of change that was effected on the basic block. This would mean that, in some sense, the incremental scheduler is doing only as much work as required. Note that in figure 14, the entry for the percentage of change in the fourth column is the number of instructions in the basic block which are changed divided by the total number of instructions in the original basic block and expressed as a percentage. Here, modified instructions include newly inserted instructions, deleted instructions and modified instructions. While computing the percentage of change, modified instructions are counted as two because the incremental scheduler considers it as a deleted instruction followed by the insertion of the modified instruction. Also note that the time taken by the instruction scheduler (both batch and incremental) is the sum of the DDAG construction time, the longest path length computation time and the code listing time.

The reason we used *random* changes is that we could have done little else. There has been hardly any empirical study on the nature of changes that programmers make to their programs. A programmer is free to make whatever changes he feels like to the source program. So the kind of changes we experimented with were the ones that we thought would mimic those that a programmer would make. For example, in the source program we would arbitrarily choose a statement $a = b + c$ and change it to $a = b + c + 1$ and see the incremental scheduler on the resulting basic block. This is just one example of the type of change made. We also experimented with changes which affect the basic blocks to a greater degree and also with some that affect them to a lesser degree.

The algorithm we have incrementalized here is very similar to Warren's algorithm [War90]. However, we have only concentrated on the longest path length criterion. We are yet to implement the other criteria used by Warren to break ties like register-pressure, greater number of children of the nodes in the DDAG etc.

10 Conclusions and Future Work

In this paper, we have described what we think is the first attempt towards an incremental approach towards machine dependent optimization. We described a simple algorithm for readjusting the DDAG which is suited to our purpose. Then we adapted existing work for incremental list scheduling based on the longest path length criterion. The ideas described here were implemented in the form of an incremental instruction scheduler for the IBM RS6000 processor. Implementation results show significant gains in scheduling time over the batch scheduler also implemented by us.

The next step would be to design an incremental approach to scheduling beyond basic blocks. This could be followed by incrementalizing code selection and register allocation. The successful solution to these problems would bring code generation also into the ambit of incremental compilation.

References

[Aho86] Aho, A.V., Sethi, R. and Ullman, J.D., *Compilers : Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, (1986).

[Alp90] Alpern, B., Hoover, R., Rosen, B.K., Sweeney, P.F. and Zadeck, K., "Incremental evaluation of computational circuits" in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, (San Fransisco, Cal., January 22-24, 1990), pp 32-42, Society for Industrial and Applied Mathematics, Philadelphia, PA,(1990).

[Bee91] Beetem, J.J. and Beetem, A.F., "Incremental scanning and parsing with Galaxy", *IEEE Trans. on Software Engineering* **17**(7) pp 641-651 (July 1991).

[Ber91] Bernstein, D. and Rodeh, M., "Global instruction scheduling for superscalar machines", *Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp 241-255, (1991).

[Ber92] Bernstein, D., Cohen, D., Lavon, Y. and Rainish, V., "Performance evaluation of instruction scheduling on the IBM RISC System/6000" in *Proceedings of the IEEE MICRO 25 Conf.* pp 226-235 (1992).

[Biv90] Bivens, M.P. and Soffa, M.L., "Incremental register reallocation", *Software Practice and Experience* **20**(10) pp 1015-1047 (October 1990).

[Bur87a] Burke, M. and Ryder, B.G., "Incremental iterative data flow analysis algorithms", Res. Rep. RC 13170, IBM T.J. Watson Research Center, Yorktown Heights, NY, (October 1987).

[For85] Ford, R. and Sawamiphakdi, D., "A greedy concurrent approach to incremental code generation", in *Conf. Record of the Twelfth ACM Symposium on Principles of Programming Languages*, (New Orleans, LA, January 14-16, 1985), ACM, New York, NY, (1985).

[Ghe79] Ghezzi, C. and Mandrioli, D, "Incremental parsing", *ACM Trans. Program. Lang. Syst.* **1**(1) pp 58-70 (July 1979).

[Gib86] Gibbons, P.B. and Muchnick, S.S., "Efficient instruction scheduling for a pipelined architecture" in *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, pp 11-16 (June 1986).

[Gol90] Golumbic, M.C. and Rainish, V., "Instruction scheduling beyond basic blocks", *IBM Journal of Research and Development* **34**(1), pp 93-98, (1990).

[Jal82] Jalili, F. and Gallier, J., "Building friendly parsers", in *Conf. Record of the Ninth ACM Symposium on Principles of Programming Languages* (Albuquerque, NM, January 25-27, 1982), pp 196-206, ACM, New York, NY, 1982.

[Kri90] Krishnamurthy, S.M., *Static Scheduling of Multi-Cycle Operations for a RISC Processor*, MS Dissertation, Clemson University, May, (1990).

[Lam88] Lam, M., *A systolic array optimizing compiler*, PhD dissertation, Carnegie Mellon University, (1987).

[Mar89] Marlowe, T.J., *Data flow analysis and incremental iteration*, PhD dissertation and Tech. Rep. DCS-TR-255, Rutgers University, New Brunswick, NJ, (October 1989).

[Pal90] Palem, K.V. and Simons, B.B., "Scheduling time-critical instructions on RISC machines" in *Proceedings of the ACM Symposium on Principles of Programming Languages* (1990).

[Pol86] Pollock, L.L., *An Approach to Incremental Compilation of Optimized Code*, PhD dissertation, University of Pittsburgh, Pittsburgh, PA, (1986).

[Ram93] Ramalingam,G., *Bounded incremental computation*, PhD dissertation and Tech. Rep. 1172, University of Wisconsin Madison,(1993).

[Rep82] Reps, T., "Optimal-time incremental semantic analysis for syntax-directed editors", in *Conf. record of the Ninth ACM Symposium on Principles of Programming Languages* (Alburquerque, NM, January 25-27, 1982), pp 169-176 ACM, New York, NY, (1982).

[Rep83] Reps, T., Teitelbaum, T. and Demers, A., "Incremental context-dependent analysis for language based editors", *ACM Trans. Program. Lang. Syst.* **5**(3) pp 449-477 (July 1983).

[Rep84] Reps, T., *Generating language-based environments*, The M.I.T. Press, Cambridge, MA, (1984).

[Rep86] Reps, T., Marceau C. and Teitelbaum, T., "Remote attribute updating for language-based editors", in *Conf. Record of the Thirteenth ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL, January 13-15, 1986), pp 1-13, ACM, New York, NY, (1986).

[Riv76] Rivest, R., "Self-organizing seqenetial search heuristics", *Comm. of the ACM* **19**(2) pp 63-67 (February 1976).

[Ryd88] Ryder, B.G. and Paull, M.C., "Incremental data flow analysis algorithms", *ACM Trans. Prog. Lang. Syst.* **10**(1) pp 1-50 (January 1988).

[Sas76] Sassa, M. and Goto, E., "A hashing method for fast set operations", *Inf. Process. Lett.* **5**(2) pp 31-34 (June 1976).

[Sch84] Schwartz, M., Delisle, N. and Begwani, V., "Incremental compilation in Magpie", in *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, (Montreal, Can., June 20-22, 1984), ACM SIGPLAN Notices, **19**(6) pp 122-131 (June 1984)

[Sle85] Sleator. D.D. and Tarjan, R.E., "Amortized efficiency of list update and paging rules" *Comm. of the ACM* **28**(2) pp 202-208 (February 1985).

[Tic86] Tichy, W.F., "Smart Recompilation", *ACM Trans. Prog. Lang. Syst.* **8**(3) pp 637-654 (July 1986).

[Tei81] Teitelbaum, T. and Reps, T., "The Cornell Program Synthesizer : A syntax-diected programming environement", *Communications of the ACM* **24**(9) pp 563-573 (September 1981).

[War90] Warren H.S. Jr., "Instruction scheduling for the IBM RISC System/6000 processor", *IBM Journal of Research and Development* **34**(1) pp 85-92 (January 1990).

[Yel90] Yellin, D., "Representing sets with constant time equality testing", in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, (San Fransisco, Cal., January 22-24, 1990), pp 64-73, Society for Industrial and Applied Mathematics, Philadelphia, PA,(1990).

A Appendix

```
dynamic_swsf(G : dependence graph, v : modified vertices)
variables
        u, v, w : vertices in dependence graph; heap : heap of vertices
begin
        heap := NULL
        for u ∈ U do
        begin
                rhs[u] := g_u(d[x_1], ..., d[x_k])
                if (rhs[u] ≠ d[u]) then
                        insertintoheap(heap, u, min(rhs[u],d[u]))
        end /* for */
        while (heap ≠ NULL) do
        begin
                u := extractanddeletemin(heap)
                if (rhs[u] < d[u]) then
                begin
                        d[u] := rhs[u]
                        for v ∈ succ(u) do
                        begin
                                rhs[v] := g_v(d[x_1], ..., d[x_k])
                                if (rhs[v] ≠ d[v]) then
                                        adjustheap(heap, v, min(rhs[v], d[v])
                                else if v ∈ heap then
                                        remove v from heap
                        end /* for */
                end /* if */
                else begin
                        d[u] := ∞
                        for v ∈ (succ(u) ∪ {u}) do
                        begin
                                rhs[v] := g_v(d[x_1], ..., d[x_k])
                                if (rhs[v] ≠ d[v]) then
                                        adjustheap(heap, v, min(rhs[v],d[v]))
                                else if v ∈ heap then
                                        remove v from heap
                        end /* for */
                end /* else */
        end
end
```

Figure 15: Ramalingam's algorithm for incremental computation of the fixed-point of a collection of swsf equations

Given in figure 15 is a description of Ramalingam's algorithm for the incremental computation of the fixed-point of a collection of swsf equations. This description is from [Ram93]. We shall omit the details of the proofs of boundedness and the complexity of the algorithm. These can be found in [Ram93] which also includes a detailed explanation of the theory behind the algorithm.

$d[x_i]$ is the output value of node x_i of the dependence graph of the collection of swsf equations before modification. $rhs[x_i]$ is the output value of node x_i ie. $g_i(d[x_i], \ldots, d[x_k])$ under the given assignment of the ouput values of the nodes at a particular instant. Procedure *insertintoheap(heap,i,k)* inserts an item i into the heap with a key k. *extractanddeletemin(heap)* gets the element with the minimum key and deletes it from the heap while *adjustheap(heap,i,k)* inserts an item i into the heap with key k if i is not in the heap and changes the key of item i in the heap to k if i is in the heap.