The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC

Vladimir N. Makarov
Red Hat
vmakarov@redhat.com

Abstract

A new model to describe the pipeline characteristics of processors is proposed in the article. The model is based on the usage of regular expressions. The model is compared to the one used in GNU C compiler (GCC) for long time. The article also describes the pipeline hazard recognizer generated from the new model currently implemented in GCC and instruction scheduler which uses the pipeline hazard recognizer. The current implementation of the pipeline hazard recognizer is based on the usage of *deterministic* and *nondeterministic* finite state automata.

Examples of usage of the new model, the pipeline hazard recognizer, and the instruction scheduler based on it are given. Possible future directions of developing them to use them for different algorithms of instruction scheduling and software pipelining are discussed.

Introduction

To increase the productivity of computer systems the modern processors can execute several instruction simultaneously. It is achieved by using several functional units and/or pipelined functional units. Of course the instruction execution could start only if the input data are ready and enough processor functional units necessary for the instruction execution are available. If at least one of the two conditions is not satisfied, a processor stall might occur and the instruction execution might be delayed. The delay because the first condition is not satisfied is called data delay. The delay because of the second condition is called resource delay.

A special component in an optimized compiler, called the instruction scheduler, is responsible for decreasing the data and resource delays and (as a consequence) to increase the parallelism of instruction execution. It is achieved mainly by changing the original order of instructions, although more powerful code transformation (like instruction cloning, partial register renaming and forward substitution, and instruction mutation) could be used. An important component of the instruction scheduler responsible to find the resource delays is called the pipeline hazard recognizer.

There is big variety of processors even for one architecture. Therefore writing the pipeline hazard recognizer manually is not wise. This is especially true for portable compilers. Therefore many compilers have a model to describe pipeline characteristics of the target processors and usually a generator of pipeline hazard recognizers. The model language can be a subset of the compiler implementation language (like C used to describe the reservation tables) or a special language designed for this task.

GCC as a compiler ported to most platforms had such a model and generator for long time. This model has its drawbacks. It can not accurately describe many modern processors. As a consequence the generated code is worse than it could be with the same instruction scheduler. The more pipeline irregularity the processor has, the more is the impact of an instruction scheduling inaccuracy. Another drawback is the inconvenience of description. The model is oriented to describe which instructions a functional unit executes instead of the more natural model in which the reservation of the functional units by given instruction is described.

GCC pipeline hazard recognizer is a part of the instruction scheduler itself. It is driven by tables generated from the description. The tables are just a simple translation of the description. The more complex the pipeline description is, the slower the pipeline hazard recognizer is. The modern processor becomes more complex and the slow speed of the pipeline hazard recognizer becomes a problem.

To solve this drawback, the new model and implementation of the pipeline hazard recognizer have been proposed. The model is based on the usage of regular expressions describing all the reservations of functional units by instructions. The corresponding implementation of pipeline hazard recognizer is based on the usage of finite state automata.

Each state of the automaton encodes all current and planned reservations of functional units. If there is an arc from one state to another state marked by an instruction, then the instruction can be issued in a given state and there will be no conflicts on functional unit usage with the instructions issued earlier. The destination state encodes all current and planned functional unit reservations after issuing the instruction. Each state also has an arc marked by *cycle advancing*. The destination state in this case is the state after increasing the simulated processor cycle. Transitions by the arc finally result in freeing functional units.

So the instruction scheduler should only check the presence of the arcs marked by the instruction from given state to find a resource delay. After issuing the instruction the instruction scheduler should change the current state to the destination state. If no instruction can be issued, the instruction scheduler should change the current state to the destination state into which an arc marked by ‘cycle advancing’ enters and increase the simulated processor cycle.

This approach is not new. It has been described in [Bala, Proebsting]. What is a really new thing in the approach described in the article is usage of *alternatives* in the reservations. The alternatives can be treated *deterministically* and *nondeterministically*.

The deterministic treatment of the alternative is to try the first alternative reservation and, if there is a conflict on any functional unit reserved by previously issued instructions, try the next alternative. The nondeterministic one is to try all alternative reservations concurrently.

The first section of the article describes in more detail the description model and the corresponding pipeline hazard recognizer used in GCC for a long time. It also describes the drawbacks of such an approach. In the second section, the proposed model is described. The third section describes the generation of the pipeline hazard recognizer from the proposed model and its interface to the instruction scheduler. The fourth section contains examples of descriptions as deterministic and nondeterministic ones. The fifth section describes an algorithm, called the first cycle multipass instruction scheduling. The algorithm improves instruction scheduling by evaluation of more than one instruction schedule. Usage of the fast

pipeline hazard recognizer makes it practical. In the sixth section, the possible future directions of developing the proposed approach are discussed.

1 The old GCC processor pipeline description model

This section is based on the documentation of Gcc internals [Gcc]. Practically all processor parallelism for GCC is described with the aid of one type of constructions—`define_function_unit`—in a Gcc machine description file. Each usage of a functional unit by a class of instructions is specified with a `define_function_unit` expression (see Table 1).

`(define_function_unit NAME MULTIPLICITY SIMULTANEITY TEST READY-DELAY ISSUE-DELAY [CONFLICT-LIST])`	
`NAME` is a string giving the name of the functional unit.	`MULTIPLICITY` is an integer specifying the number of identical units in the processor. If more than one unit is specified, they will be scheduled independently.
`SIMULTANEITY` specifies the maximum number of instruction that can be executing in each instance of the functional unit simultaneously.	`TEST` is an attribute test that selects the instructions we are describing in this definition. Note that an instruction may use more than one functional unit.
`READY-DELAY` is an integer that specifies the number of cycles after which the result of the instruction can be used without introducing any stall.	`ISSUE-DELAY` is an integer that specifies the number of cycles after the instruction matching the `TEST` expression begins using this unit until a subsequent instruction can begin. A cost of `N` indicates an `N-1` cycle delay.
`CONFLICT-LIST` is an optional list giving instructions with which additional conflicts occur.	

Table 1: The old description model construction.

As an example, consider a classic RISC machine where the result of a load instruction is not available for two cycles (a single "delay" instruction is required) and where only one load instruction can be executed simultaneously. This would be specified as:

```
(define_function_unit "memory" 1 1
       (eq_attr "type" "load") 2 0)}
```

For the case of a floating point function unit that can pipeline either single or double precision, but not both, the following could be specified:

```
(define_function_unit "fp" 1 0
       (eq_attr "type" "sp_fp") 4 4
       [(eq_attr "type" "dp_fp")])
(define_function_unit "fp" 1 0
       (eq_attr "type" "dp_fp") 4 4
       [(eq_attr "type" "sp_fp")])
```

A special utility in Gcc generates different tables of bit vectors, macros, and some functions (mainly for dealing with conflict lists), which are used by the pipeline hazard recognizer embedded into the instruction scheduler.

The current GCC instruction level parallelism description model has serious drawbacks. The biggest one is that the description model is not powerful enough. Each functional unit is believed to be reserved at the start of instruction's execution. The model also does not permit alternatives in the reservations. This is a big constraint for accurate descriptions of modern processors. As a consequence of inaccurate descriptions, the machine dependent files of Gcc contain a lot of code to fix it. For example, the SPARC machine-dependent files contained about one thousand lines of C code.

Another important drawback of the model is the unnatural way of description when a developer should write a unit and condition which selects instructions using the unit. My experience shows that writing all units reservation for

an instruction (an instruction class) are more natural.

The pipeline hazard recognizer of resource delays has a slow implementation. The Gcc schedulers support structures which describe the unit reservations. The more complex the pipeline description, the slower the pipeline hazard recognizer. Such implementation would become even slower when we enable to reserve functional units not only at the instruction execution start. The slow implementation becomes critical for the modern processor (especially VLIW and EPIC).

2 The proposed processor pipeline description model and its implementation

As the old processor pipeline description, the proposed pipeline description should be placed in the machine description files of Gcc. There are several constructions to describe the processor. The order of all such constructions in the machine description file is not important. All constructions are Lisp like construction because the machine description file has Lisp like syntax. Please don't be confused—it is just an implementation form of the description model. The syntax of the major constructions is given on Table 2.

`(define_automaton AUTOMATON-NAME)`	`AUTOMATA-NAME` is a string giving the name of the automaton.
`(define_cpu_unit UNIT-NAMES AUTOMATON-NAME)`	`UNIT-NAMES` is a string giving the names of the functional units. `AUTOMATON-NAME` is a string giving the name of the automaton to which the unit is bound.
`(define_insn_reservation INSN-NAME DEFAULT-LATENCY CONDITION REGEXP)`	`DEFAULT-LATENCY` is a number giving the latency time of the instruction. `INSN-NAME` is a string giving an internal name of the instruction. It is good practice to use the instruction class names as described in the processor manual. `CONDITION` defines what RTL instructions are described by this construction. `REGEXP` is a string describing the reservation of the cpu's functional units by the instruction (the syntax is given in table 3).
`(define_reservation RESERVATION-NAME REGEXP)`	`RESERVATION-NAME` is a string giving the name of REGEXP.
`(exclusion_set UNIT-NAMES UNIT-NAMES)` `(presence_set UNIT-NAMES PATTERNS)` `(absence_set UNIT-NAMES PATTERNS)`	`UNIT-NAMES` is a string giving names of functional units. `PATTERNS` is a string giving patterns of functional units separated by a comma. Currently a pattern is one unit or units separated by white-spaces.

Table 2: The major constructions of the proposed description model.

To describe a processor, first we should define an automaton with the construction `define_automaton`. We can have more than one automaton in a machine description file. All the automata should have unique names. The automaton name is used in the construction `define_cpu_unit`.

It is good practice to use separate automaton to describe a processor of a given architecture. For example, the machine description file for SPARC architecture could have one automaton for UltraSparcII and another one for UltraSparcIII.

We could also use more than one automaton to describe a single processor. Sometimes the generated finite state automaton used by the pipeline hazard recognizer is large. If we use more than one automaton and bind functional units to the automata, the summary size of the automata is usually less than the size of the single automaton.

Each functional unit used in the description of instruction reservations should be described by the construction `define_cpu_unit`.

The construction `define_insn_reservation` is the major construction to describe pipeline characteristics of an instruction. The reservations are described by regular expressions according to the syntax on Table 3.

`regexp = regexp "," oneof` `| oneof`	`,` is used for describing the start of the next cycle in the reservation.
`allof = allof "+" repeat` `| repeat`	`+` is used for describing a reservation described by the first regular expression and the second regular expression etc.
`oneof = oneof "|" allof` `| allof`	`|` is used for describing a reservation described by the first regular expression or the second regular expression etc.
`repeat = element "*" number` `| element`	`*` is used for convenience and simply means a sequence in which the regular expression is repeated `NUMBER` times with cycle advancing (see ‘`,`’).

`element = cpu_unit_name`
`| reservation_name`
`| result_name`
`| "nothing"`
`| "(" regexp ")"`
	`cpu_unit_name` denotes reservation of the named cpu functional unit. `nothing` denotes no unit reservations.

Table 3: Syntax of the regular expressions.

As an example, consider a superscalar RISC machine which can issue three instructions (two integer instructions and one floating point number instruction) on a cycle but can finish only two instructions. To describe this, we define the following functional units.

```
(define_cpu_unit "i0_pipeline, i1_pipeline")
(define_cpu_unit "f_pipeline,port0, port1")
```

All simple integer instructions can be executed in any integer pipeline and their result is ready in two cycles. The simple integer instructions are issued into the first pipeline unless it is reserved, otherwise they are issued into the second pipeline. Integer division and multiplication instructions can be executed only in the second integer pipeline and their results are ready correspondingly in 8 and 4 cycles. The integer division is not pipelined, i.e. the subsequent integer division instruction can not be issued until the current division instruction finished. Floating point instructions are fully pipelined and their results are ready in 3 cycles. To describe all of this we could specify

```
(define_cpu_unit "div")
(define_insn_reservation "simple" 2
   (eq_attr "cpu" "int")
   "(i0_pipeline|i1_pipeline), (port0|port1)")
(define_insn_reservation "mult" 4
   (eq_attr "cpu" "mult")
   "i1_pipeline, nothing*2, (port0|port1)")
(define_insn_reservation "div" 8
   (eq_attr "cpu" "div")
   "i1_pipeline, div*7, div + (port0|port1)")
(define_insn_reservation "float" 3
   (eq_attr "cpu" "float")
   "f_pipeline, nothing, (port0|port1))
```

In our example we see that the unit reservations for different instructions contain common parts. In such case, we can simplify the pipeline description by defining an abbreviation by the construction `define_reservation`. To simplify the description in our example we could use a reservation as follows

```
(define_reservation "finish" "port0|port1")
(define_insn_reservation "simple" 2
      (eq_attr "cpu" "int")
      "(i0_pipeline | i1_pipeline), finish")
```

Some processors (especially VLIW ones) have many constraints which are quite difficult to describe only by the constructions mentioned above. The three constructions `exclusion_set`, `presence_set`, and `absence_set` make description easy.

The first construction (`exclusion_set`) means that each functional unit in the first string can not be reserved simultaneously with a unit whose name is in the second string and vice versa. For example, the construction is useful for describing processors (e.g. some SPARC processors) with a fully pipelined floating point functional unit which can execute simultaneously only single precision floating point instructions or only double precision floating point instructions.

The second construction (`presence_set`) means that each functional unit in the first string can not be reserved unless at least one of the pattern in the second string has been reserved. This is an asymmetric relation. For example, it is useful to description that VLIW `slot1` is reserved after a reservation `slot0` or `slot1` is reserved only after a `slot0` and unit `b0` reservation. We could describe it by the following constructions:

```
(presence_set "slot1" "slot0")
(presence_set "slot1" "slot0 b0")
```

The third construction (`absence_set`) means that each functional unit in the first string can be reserved only if each pattern in the second string is not reserved. This is an asymmetric relation. For example, it is useful for description that VLIW `slot0` can not be reserved after a `slot1` or `slot2` reservation or that `slot2` can not be reserved if `slot0` and unit `b0` are reserved or `slot1` and unit `b1` are reserved. We could describe it by the following constructions:

```
(absence_set "slot2" "slot0, slot1")
(absence_set "slot2" "slot0 b0, slot1 b1")
```

All functional units mentioned in a set should belong to the same automaton.

There are other constructions to describe pipeline characteristics of processors. But for the sake of brevity they are not described in this article.

A special utility (the generator) generates the automaton based pipeline hazard recognizer in a separate file. The instruction scheduler communicates with it through a procedural interface. The major procedure gets an automata state and an instruction as parameters and returns information on whether the instruction can be issued or not. If it can be issued then the procedure changes the state to reflect the instruction issue.

Each state of the automaton encodes all current and planned reservations of functional units. If there is an arc to another state marked by an instruction, then the instruction can be issued in the given state and there will be no conflicts on functional unit usage with the instructions issued earlier. The destination state encodes all current and planned functional unit reservations after issuing the instruction. If the instruction parameter is null, it means that the simulated processor cycle should be advanced. Each state has an arc marked by `cycle advancing`. The destination state in this case is the state after incrementing the simulated processor cycle. Transitions by such arcs result in the freeing of all functional units.

The DFA pipeline hazard recognizer is believed to not be as flexible as the old Gcc recognizer. This is not true. It is easy to get information from the automata. For example, the generator also generates many other procedures like querying the reservation of functional units for a given automaton state, finding the minimal reservation delay needed to issue an instruction in a given state, checking that no one instruction can be issued in given state and so on.

The *nondeterministic* treatment of alternatives means trying all alternatives concurrently. Some of them may be rejected by reservations

in the subsequent instructions. Actually, the nondeterministic treatment of alternatives is enough to describe deterministic alternatives. For example, let us look at the following reservation with deterministic treatment of alternatives.

```
(define_reservation "deterministic"  "u1|u2")
```

It means that we reserve `u1` and, if it is not possible (because `u1` has been already reserved), we reserve `u2`. We can describe it with the following constructions

```
(define_reservation "nondeterministic"
       "u1|u2+u1_present")
(presence_set "u1_present" "u1")
```

Here we use a reservation with nondeterministic treatment of the alternative. What variant of alternative should we use? The processors are deterministic devices, so alternatives should usually be treated deterministicaly (this is the default treatment). Let us look at a dual instruction issue processor which has two integer units. One integer unit `IU1` can execute any integer instruction and another one (`IU2`) can execute any integer instruction except multiply. In the first example, the processor always issues instructions into `IU1` if it is free. The processor could be described by using deterministic alternatives as follows

```
(define_insn_reservation "int" 1
       (eq_attr "cpu" "int") "IU1 | IU2")
(define_insn_reservation "mult" 1
       (eq_attr "cpu" "mult") "IU1")
```

Actually the processor has a bad design because if an integer instruction is followed by multiply instruction the two instructions can not be issued simultaneously. The improved processor should always issue an integer instruction into `IU2` if it is not busy. We could describe this using deterministic alternatives as follows

```
(define_insn_reservation "int" 1
       (eq_attr "cpu" "int") "IU2 | IU1")
(define_insn_reservation "mult" 1
       (eq_attr "cpu" "mult") "IU1")
```

On the other hand we could use nondeterministic treatment in the example too. The result automaton would be the same. But nondeterministic treatment could better reflect the processor's behaviour if the processor had an instruction look ahead buffer to find the best assignment of functional units to instructions in the buffer. Another example of usage of the nondeterministic treatment of alternatives for Itanium and Itanium2 processors is described in the next section.

Generally speaking, the same processor can be described differently. I would distinguish two kind of descriptions. One is the *structural* description which describes (almost) all processors functional units mentioned in processor's documentation. Another one (*behavioural*) aims to describe only pipeline hazards (sometimes with the aid of non-existing functional units). The first one is usually more verbose and the resulting automata are bigger. The second one is simpler and the resulting automata are smaller. But it is better to follow the documentation (in other words to use a structural description) because it makes understanding the description easier for other people.

3 Generation of the pipeline hazard recognizer

Here is a brief description of the phases of the generator of pipeline hazard recognizers and the more interesting tasks solved by the generator. First, the generator of pipeline hazard recognizer translates the pipeline description into an internal representation.

Then it checks the correctness of the automaton pipeline description. The most nontrivial

task is to check the correctness of assignments of functional units occurring in a reservation to the automata. There is no such problem for reservations without alternatives [Bala]. Let us consider the following description:

```
(define_cpu_unit "div" "div")
(define_cpu_unit "decode" "rest")
(define_insn_reservation "div" 3
       (eq_attr "cpu" "div") "decode + div*3")
```

The corresponding automata are given on Figure 1. The figure also contains the single automaton as if all units were assigned to one automaton. They behave analogously to the single automaton with the two functional units `decode` and `div`. It means that transition marked by an instruction exists in the single automaton if and only if there are transitions marked by the instruction between the corresponding states of all two automata. Instead of changing only one state for a single automaton, the pipeline hazard recognizer changes the states of the two automata simultaneously. Although a number of the states is hidden in the pipeline hazard interface and there is only one state in the interface, in reality the interface state is represented by two states and pipeline hazard recognizer internally manipulates the states of the two automata.

Let us consider a more advanced dual instruction issue processor with a faster division unit.

```
(define_cpu_unit "decode1" "a1")
(define_cpu_unit "div,decode2" "a2")
(define_insn_reservation "div" 2
       (eq_attr "cpu" "div")
       "(decode1|decode2) + div*2")
```

For automata `a1` and `a2` we have correspondingly the following functional unit reservations for the instruction `div`

```
decode1|nothing
nothing|decode2 + div*2
```

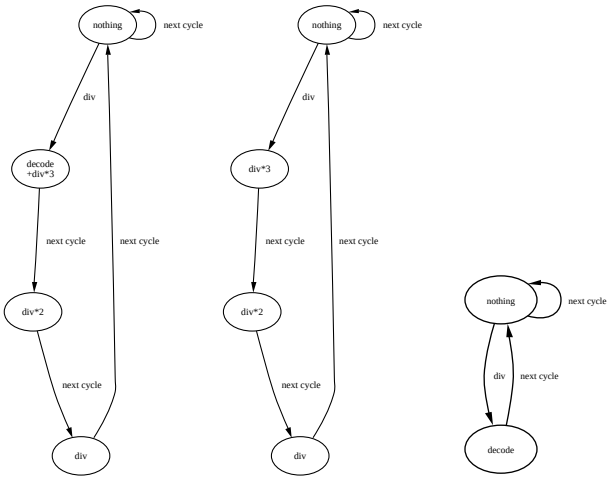

Figure 1: The single automaton and the two automata of the single issue processor.

Figure 2 contains the single automaton (as if all units were assigned to one automaton) and the corresponding two automata.

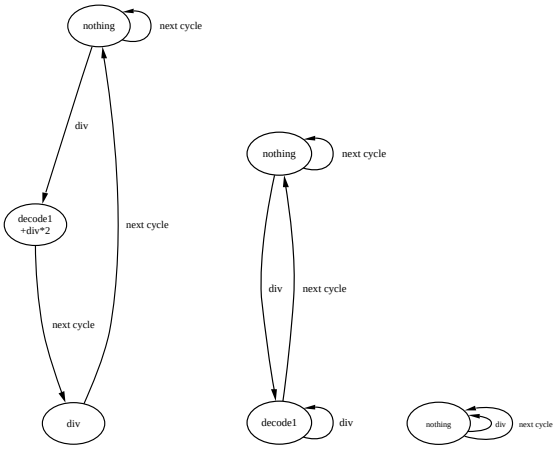

Figure 2: The single automaton and the two incorrect automata of the dual issue processor.

The two automata are not equivalent to the single automata. For example, we could issue any number of division instructions on one cycle according to the two automata. The simple solution of this problem could be the usage of the requirement to assign all functional

units occurring in the same reservation to the same automaton. It is a very severe constraint to assign functional units to automata which results in the impossibility of decreasing automata size in many cases even if we have reservations without alternatives. Instead of it, the current implementation uses a less severe requirement. If a functional unit reservation (`div` in our example) is present on a particular cycle of an alternative for an instruction reservation, then some unit from the same automaton must be present on the same cycle for the other alternatives of the instruction reservation. The requirement is not too complicated to be understood and it still helps to considerably decrease automata size in many cases. Let us consider the following distributions of the functional units (The corresponding automata are given on Figure 3):

```
(define_cpu_unit "decode1,decode2" "a1")
(define_cpu_unit "div" "a2")
(define_insn_reservation "div" 2
       (eq_attr "cpu" "div")
       "(decode1|decode2) + div*2")
```

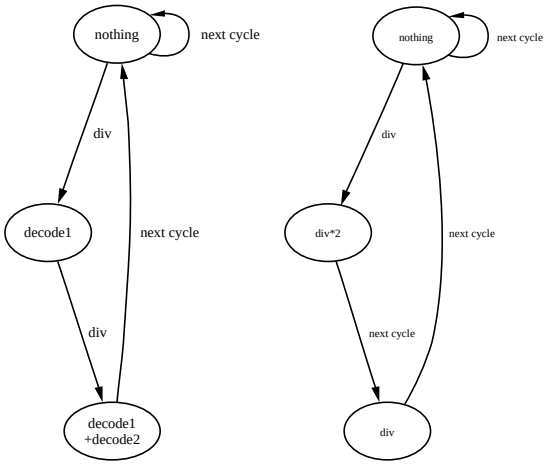

Figure 3: The two correct automata of the dual issue processor.

We see that the automata on figure 3 behave analogously to the single automaton.

After checking the description, the generator of the pipeline hazard recognizer creates the automata and, if the alternatives are treated nondeterministicaly, transforms nondeterministic finite state automata into deterministic ones.

After creating the automata, the generator does a minimization of the finite state automata by merging automaton states (I should mention that Gcc experience shows importance of some preliminary minimization during building the automata because even if the minimized is small the automata before the minimization could be huge). The minimization task is a bit complicated. If we have functional units in the description whose reservation may be queried for a given state. Let us consider a processor with different functional units for multiply and for the rest of the integer instructions

```
(define_insn_reservation "int" 1
       (eq_attr "cpu" "int") "decode + int")
(define_insn_reservation "mult" 1
       (eq_attr "cpu" "mult") "decode + mult")
```

The corresponding automata before and the after the minimization are given on Figure 4. If we want to know whether functional unit `mult` is reserved in the second state of the minimized automaton, we can not get this information from just the state. The simplest solution of the problem could be prohibiting the minimization for automata with queried units. Unfortunately such a solution is not reasonable because automaton minimization is an important optimization which permits to considerably decrease the size of the automata in many cases. Instead of the simplest approach we use minimization with modified state equivalence. The new state equivalence takes queried functional units in the corresponding reservations into account. This approach still permits to considerably decrease the automata size in many cases.

After the minimization, the generator forms tables, compresses them by different algorithms

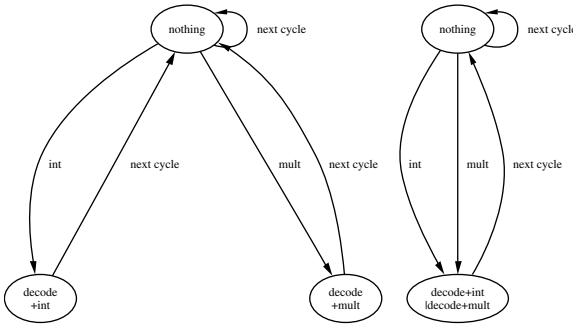

Figure 4: The automaton before and after the minimization.

(like a comb vector algorithm) and outputs them (and functions accessing them (including functions which are interface functions of the pipeline hazard recognizer)) into a C file for further compilation.

The biggest problem of the usage of the DFA approach is the size of the automata. How big can the automata be? For example, Gcc for Intel IA64 has four automata (two for Itanium and two Itanium2) with 24K states and 170K arcs. But this is an extreme case. Itanium and Itanium2 have extremely complicated pipeline characteristics. The IA64 automata are also used for VLIW packaging (bundling instructions). Therefore the IA64 automata have many queried units.

To solve the big automata size problem, it is better to split an automata into several ones and not to use queried units as it was mentioned in above. Now automaton splitting should be done manually by assigning functional units to the automata. Automatic splitting of an automaton into several automata with total size less than the size of the original automaton is a challenging research work.

4 Usage of the proposed model and the pipeline hazard recognizer

The first public usage of DFA based instruction scheduling was for UltraSparc. The previous implementation of the pipeline hazard recognizer contained about 1000 lines of machine-dependent C code for tuning the old pipeline hazard recognizer generated from a non-DFA pipeline description. The DFA description of Sparc which resulted in all this code has been gone and the instruction scheduling has been improved. Table 4 contains a comparison of SPECfp95 run a 500 Mhz UltraSparcIIe box. The average improvement of EEMBC [EEMBC] for a 233 Mhz UltraSparcII box was 5.5%.

Benchmarks	Ratio	Ratio
101.tomcatv	12.6	13.4
102.swim	22.4	22.7
103.su2cor	5.95	6.04
104.hydro2d	6.04	6.05
107.mgrid	7.73	8.65
110.applu	7.52	7.65
125.turb3d	13.7	13.8
141.apsi	10.9	11.0
145.fpppp	11.0	11.4
146.wave5	14.7	15.0
SPECfp95 (Geom. Mean)	10.4	10.7

Table 4: Sparc GCC with non-DFA and DFA pipeline hazard recognizers.

The usage of a DFA description for SH4 is an example of the importance of accurate pipeline descriptions for processors which have complicated pipeline constraints: such as SH4. Improvement of instruction scheduling with the DFA pipeline hazard recognizer for SLALOM benchmark [Slalom] on a 200Mhz SH4 box was about 12-13%.

A good example of usage of *nondeterministic* automata is the description of Itanium and Ita-

nium2 processors. The IA64 architecture is an extension of a typical VLIW architecture. Instructions can only be placed in specific slots (syllables in IA64 terminology) of a VLIW instruction (bundle in IA64 terminology). To place an instruction in the current bundle or the next bundle, sometimes one or two NOP instructions should be issued first. Gcc already had pipeline hazard recognizer for the Itanium processor. It was written manually on C because the old description model was not powerful enough. The code was big and complicated. It was tuned very well to achieve good instruction scheduling. The code tried to insert such NOP instructions.

The nondeterministic automaton permits to easily describe where to insert such NOP instructions. The DFA descriptions have been written for Itanium and Itanium2 processors. Each processor has been described by two automata. The first (nondeterministic) automaton described the instruction reservations with an optional issue of one or two NOP instructions before the instruction. So the pipeline hazard recognizer followed all possibilities of inserting NOP instructions. This automaton is used for the first and second instruction scheduling in Gcc. The second automaton is deterministic. It is used to bundle instructions on the final phase of Gcc. Bundling instructions is to insert NOPs and *template selectors*. Inserting NOPs was a dynamic programming algorithm which tests all alternatives in inserting NOPs before the instructions and choses the best ones. It uses the second automaton and information about new processor cycle start points prepared by the previous instruction scheduling. Templates are defined by querying the functional units of the second automaton.

Such implementation permitted to speed up all Gcc run (with -O2) up to 45% for Itanium. The code has been improved by 2% (see Table 5) for SPECInt2000 benchmark on a 733 Mhz Ita-

Benchmarks	Ratio	Ratio
164.gzip	176	177
175.vpr	192	203
176.gcc	236	235
181.mcf	142	144
186.crafty	248	243
197.parser	168	171
252.eon	149	147
253.perlbmk	201	207
254.gap	163	167
255.vortex	232	233
256.bzip2	182	188
300.twolf	247	265
Est.SPECint2000	191	195

Table 5: Itanium Gcc with non-DFA and DFA pipeline hazard recognizers.

Benchmarks	Ratio	Ratio
164.gzip	345	361
175.vpr	444	454
176.gcc	460	477
181.mcf	252	249
186.crafty	480	497
197.parser	366	368
252.eon	274	273
253.perlbmk	449	463
254.gap	326	331
255.vortex	509	512
256.bzip2	362	376
300.twolf	506	559
Est. SPECint2000	388	399

Table 6: Itanium Gcc with non-DFA pipeline hazard recognizers vs. Itanium2 Gcc with DFA pipeline hazard recognizer.

nium box.

Unfortunately, there is no implementation of Gcc for Itanium2 using a non-DFA pipeline hazard recognizer. Therefore we could only compare Itanium compiler using the non-DFA pipeline hazard recognizer with the Itanium2 compiler using the DFA-pipeline hazard recognizer. The compiler speed up is about 55% for such a comparison. The SPECInt2000 benchmark results of Gcc (with usage -O2) on a 900Mhz Itanium2 box are given in Table 6.

5 The first cycle multipass instruction scheduling

The usage of the fast DFA pipeline hazard recognizer permits to implement instruction scheduling algorithms trying several schedules and choosing the best one. The traditional instruction scheduling algorithms try only one instruction schedule. The schedule is chosen by a fixed set of heuristics. Usually the major heuristic is a heuristic based on the critical path length [Muchnick, Morgan]. This heuristic works fine for classical RISC processors. For super-scalar RISC or VLIW processors, a greedy algorithm [Muchnick] trying to issue the maximal number instructions on each processor cycle might work better.

The first cycle multi-pass instruction scheduling has been designed to integrate the best of the both approaches. The idea of the algorithm is to choose an instruction whose issue can result in the issue of a maximal number of instructions on the current simulated processor cycle. The highest priority instruction should be among these instructions. In other words, the algorithm guarantees that the instruction with the highest priority will be issued on the current cycle (although necessarily not the first in the cycle). On the other hand, it tries to maximize the number of issued instructions on the cycle. The second highest priority instruction might be not issued on the same cycle even if it could be issued with the highest priority instruction. If it happens, the second highest priority instruction will be issued on the next cycle.

```
function MaxIssues (ReadyArray, var ReadyTry,
                    State, var Index) : integer
begin
  if no one instruction can be issued in State
  then return 0; fi

  Best := 0;

  for i := 0 to length (ReadyArray) do
    if not ReadyTry [i] then
      Insn := i-th of ReadList;
      TempState := State;
      if Insn can be issued in TempState then
        change TempState as if Insn were issued;
        ReadyTry [i] = true;
        n := MaxIssues (ReadyArray, TempState,
                        TempIndex);
        if n > 0 || ReadyTry[0]
        then n := n + 1; fi;
        if Best < n then
          Best := n;
          Index := i;
        fi;
        ReadyTry [i] := false;
      fi
    fi
  end
  return Best;
end

function ChooseReady (ReadyArray, State) : Insn
begin
  ReadyTry := array of length (ReadyArray)
              initialized by false;
  if MaxIssues (ReadyArray, ReadyTry,
                State, i) == 0
  then return the first instruction in
       ReadyArray;
  else return i-th instruction in ReadyArray;
  fi
end
```

Figure 5: The first cycle multi-pass instruction scheduling algorithm.

To find the instruction to issue, the algorithm tries permutations of an array of ready instructions sorted by their priorities. The algorithm might try too many permutations. Therefore the speed of the pipeline hazard recognizer is critical. The number of all permutations is $n!$, where n is number of the ready instructions.

This number can be huge and some heuristics are used to limit the processed permutations. The recursive version of the algorithm (without the heuristics) is given in Figure 5.

The algorithm is written on a Pascal/Modula like language. The function `ChooseReady` gets the array of the ready instructions sorted by their priorities and a DFA state reflecting the current and future functional unit reservations and returns a ready instruction which should be issued. The function calls another function `MaxIssues` to find the best instruction. The recursive function `MaxIssues` gets the ready instruction array, the information about already issued ready instructions as a boolean array, and the current DFA state reflecting issuing the ready instructions. The function returns the maximal number of instructions which can be issued in the given conditions and the index of the instruction which should be issued first to achieve this number. The function checks only the instruction sequences which contain the first ready instruction.

How much can the algorithm improve the code? The improvement can be significant especially for VLIW processors. For example, the test twolf from SpecInt2000 has been improved by 12% for an Intel Itanium2 machine. The overall SpecInt2000 has been improved by 2%. It should be mentioned that a modified algorithm, limiting number of the permutations being checked, was used. The modification was necessary to make the algorithm fast (as a small fraction of all the instruction scheduler work time) so as to be practical for use in industrial compilers.

The algorithm tries all the possibilities to improve the schedule in the scope of one processor cycle. It can be generalized to improve code in scope of a basic block. So the algorithm can be considered as an intermediate step in the algorithm making an optimal or close to optimal instruction schedule.

6 Future directions

The pipeline hazard recognizer based on the proposed model of description and its DFA implementation could be developed in the following ways:

- The same approach in the implementation of the old pipeline hazard recognizer could be used for an implementation of the proposed model. It means a slower but more compact pipeline hazard recognizer. Such an implementation could be useful for debugging and for complicated cases when the automata are too big.
- Some optimization algorithms need to define a DFA state before issuing an instruction having a DFA state after issuing the instruction. It is necessary for trace scheduling [Fisher]. It could be useful for VLIW slot assignment (instruction bundling) too when we have a final DFA state at the end of the basic block and we move backward querying functional unit reservations in order to place instructions into VLIW slots. This kind of algorithm requires reversed automata generation [Bala].
- Some algorithms need a union of DFA states. The union of two DFA states is a DFA state which reflects the union of functional unit reservations (in other words, a simultaneous reservation of functional units) from the both DFA states. It is necessary when we need to know the worst case in a joint point of control flow graph. Perfect software pipelining [Allan] and some interblock instruction scheduling are such kind of algorithms.

The union of states is also necessary for the most widely used kind of software pipelining - modulo scheduling [Allan]. To implement modulo scheduling we need to make a union of the state after instruction issue and the states gotten from the it by advancing the simulated processor cycle by `II * n`, where `II` is the initiation interval and `n` is 1, 2, 3 and so on. `N` could be constrained by a value which results in the state designating no functional unit reservations.
Actually, we could implement the union of states as simply a set of the states. But it results in a slower implementation. On the other hand, adding states to an automaton which are the union of all automaton states might result in the generation of a huge automaton. So this approach requires additional research.

The first cycle multipass instruction scheduling tries all possibilities to improve the schedule in the scope of one processor cycle. It can be generalized to improve code in scope of a whole basic block. It means an optimal or close to optimal instruction scheduling. Optimal instruction scheduling is a *NP*-hard task. But we could decrease the number of all considered instruction schedules by heuristics and by using dynamic programming to reuse the results of optimal instruction scheduling of a subsequence of the instructions. The DFA pipeline hazard recognizer would be important part of the optimal instruction scheduling implementation because of its speed.

Regular expressions in the current implementation describes automata forming *direct acyclic graphs* (DAGs). It is not an adequate model to accurately describe *out-of-order speculative execution* processors. Usually they have register renaming buffers, retire queues and so on. *Generic regular expressions* or *context free grammars* could be an accurate description model for such processors. The single question is "is it worth implementing?" From my point of view, such an accurate description will not give significant improvement of instruction scheduling for the processors. But it could be a good research work.

7 Acknowledgments

I would like to thank Robert Morgan and Norman Rubin. Communication with them gave me my original interest in automaton based pipeline hazard recognizers.

I am grateful to my current and former colleagues at RedHat for their interest in and support of this project. Among them are Richard Henderson who had the biggest impact on the design of the description model, Jeff Law who provided resources for this work through different contracts, Jason Eckhardt for the exchange of interesting ideas in this field, David Miller who wrote the UltraSparc description and had proven the advantages of the approach.

I should name many contributors to Gcc who have affected this work. The full list could be very long. So I only name Jim Wilson at RedHat, Jan Hubicka at SUSE, David Edelhson at IBM, Geoffrey Keating at Apple, Naveen Sharma at HCL Technologies, Dan Nicolaescu at University of California, Irvine. This is the power of the open source community!

Last but not least, I would like to thank my son, Serguei, for the help in editing the article.

References

[Bala] V. Bala and N. Rubin, *Efficient Instruction Scheduling Using Finite State Automata*, International Journal of Parallel Programming (1995).

[Proebsting] T. Proebsting and C. Fraser, *Detecting pipeline structural hazards quickly*, Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1994) p. 280–286.

[Gcc] *A Gcc Manual*, Published by the Free Software Foundation, 59 Temple Place – Suite 330, Boston, MA 02111–1307 USA.

[EEMBC] EEMBC, `http://www.eembc.org`

[Slalom] Slalom, `http://www.scl.ameslab.gov/Publications/SLALOM/FirstScalable.html`

[Allan] V. Allan and others, *Software pipelining*, Computing Survey, Sept. (1995).

[Muchnick] Steven S. Muchnick, *Advanced compiler design implementation*, Academic Press (1995), ISBN 1–55860–320–4.

[Morgan] Robert Morgan, *Building an Optimizing Compiler*, Digital Press, ISBN 1–55558–179–X.

[Fisher] J. A. Fisher, *Trace scheduling: A technique for global microcode compaction*, IEEE Trans. Computing 30 (1981) p. 478–490.