View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Using clp(FD) to Support Instruction Schedulers for Multi-issue Pipelined Architectures

Annie Despland, Monique Mazaud

▶ **To cite this version:**

Annie Despland, Monique Mazaud. Using clp(FD) to Support Instruction Schedulers for Multi-issue Pipelined Architectures. [Research Report] RR-3078, INRIA. 1996. inria-00073614

HAL Id: inria-00073614

https://hal.inria.fr/inria-00073614

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Using `clp(FD)` *to support instruction schedulers for multi-issue pipelined architectures.*

Annie Despland, Monique Mazaud

N° 3078

Decembre 1996

THÈME 2

ISSN 0249-6399

Using `clp(FD)` to support instruction schedulers for multi-issue pipelined architectures.

Annie Despland, Monique Mazaud

Thème 2 — Génie logiciel
et calcul symbolique
Projet dirrocq

Rapport de recherche n°3078 — Decembre 1996 — 24 pages

Abstract: In the conventional models of pipelined architectures, pipeline conflicts are generally avoided through techniques like reservation tables. They are intended to describe the run time of instructions for which the delays between the execution stage and the fetch one is a constant depending only on the instruction.

In fact actual superscalar processors don't comply this model since such delays are context dependent. The proposed approach fully supports slackness in the run time flow of execution. We advocate a model that is based on the concurrency of tasks performed by pipelines rather than the concurrent usage of resources as most current approaches do.

The scheduling algorithm is based on the properties of this model. Since most of the constraints can be stated as linear equations or inequations, an implementation using CLP with finite domains is straightforward.

(Résumé : tsvp)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Prototypage de réordonnanceurs d'instructions pour machines superscalaires en `clp(FD)`.

Résumé : Dans les appproches classiques de modélisation de machines pipelinées, les conflits sont genéralement traités en utilisant des techniques de tables de réservation. Elles sont fondées sur l'hypothèse que le délai entre la date d'exécution et la date d'alimentation dans le pipe est une constante qui ne dépend que de l'instruction.

Ceci ne correspond pas au processus effectif d'exécution dans une machine superscalaire où les délais peuvent varier pour une même instruction suivant le contexte.

Nous proposons un modèle qui prend en compte une telle élasticité. Il permet de formaliser à la fois les dépendances entre instructions d'une même séquence et les contraintes liées à l'exécution effective de celles-ci dans le processeur. Il s'appuie sur une modélisation de la concurrence des tâches effectuées par les pipelines plutôt que sur celle de l'usage concurrent des ressources comme dans la plupart des approches usuelles.

À ce modèle nous associons un algorithme de réordonnancement qui s'appuie sur les propriétes définies avec le modèle d'exécution.

Cet algorithme peut s'implanter de facon très directe en CLP avec domaines finis car ces derniers constituent un cadre approprié pour traiter l'élasticité dans le pipeline.

1 Introduction

Compilers must perform many kinds of code optimizations to effectively exploit the power of modern processors, in particular at the back-end level. One of them is called instruction scheduling. Its aim is to exploit the fine grain parallelism capabilities based on the RISC approach. These machines provide for concurrent instruction execution; the simplest and most common way to achieve this is to use a pipeline.

Superscalar processors strive to issue *multiple* instructions every cycle. This can be achieved exploiting parallelism through pipelining and multiple execution units. When an instruction needs the result of a previous one, it has to wait and the pipeline stalls (*data hazards*). Stalls can also occur when to avoid the use of the same functional unit at the same time (*structural hazards*). Structural hazards mainly arise in the new superscalar processors because their functional units are not fully replicated.

Compilers and more precisely schedulers must have a precise model of the parallel features of the machine to schedule the code.

The first generation of pipelined architectures support a linear pipeline using a single path that all instructions follow through the pipe.

In this paper, we will concentrate on parallel pipelines that provide different paths depending on the instruction type. Usually, the first stages are fully replicated for instruction fetching and decoding, then there is a splitting into different parts depending on the type instruction being executed. This allows the processing of multiple instructions with more independence. Implementations of this type are referred to as superscalar: the PowerPC, DEC ALPHA machines use such a form of parallel structure, as do most second generation implementations.

High performance can be achieved implementing either great flexibility in processing order (i.e. PowerPC) or streamlined implementation structure (DEC ALPHA). The proposed approach fully supports both in order and out of order issue.

We consider pipelines that can always be divided into static and dynamic stages of execution whenever cache misses are not considered. The first stages generally consist of instruction fetch, swap, decode and issue logic. They are called static because instructions can remain valid in the same pipeline stage for multiple cycles while waiting for a resource or stalling for other reasons. In the sequel, we will call issue the last stage of the static part of the pipeline. Upon satifying issue requirements, instructions are allowed to continue through the dynamic stages of their allowed execution path toward completion.

Most fine grain schedulers rely on precise modeling of machine resources. Resource contention specification may be based on reservation tables [1][2] or contention

recognizing state machines derived from them [3][4]. Both approaches require an important effort of the compiler writer in the understanding of the microarchitecture and leads to large sizes of either target machine specifications or generated automatons.

In this paper, we provide an abstract model of the machine that relies on a representation of the tasks performed by the pipes rather than the resources used. This leads to a more precise computation of the actual behaviour of an instruction through the multi-pipe stages. Furthermore, our abstract model offers capabilities to support both in order and out of order issues; Section 2 is devoted to the abstract model of a superscalar processor. In Section 3 we describe our execution model for a multi-issue processor model and the scheduling algorithm based on this model. Section 4 presents the implementation in CLP. Our algorithm is illustrated by an example in Section 5. Concluding remarks are given in Section 6.

2 Pipeline flow in superscalar architectures

2.1 Conventional machine model

A pipeline divides instruction processing into stages of execution, allowing overlapped execution of instructions.

In the conventional model of execution, the stages always advance state at each cycle, and are unaffected by any stall in the pipeline. In such a model, a reservation table is a good representation of resource constraints that an instruction must satisfy to enter the pipe. These techniques are borrowed from VLIW scheduling techniques.

In such a model, an instruction is fetched at cycle T_i only if it can be executed without conflicts at cycle T_{i+a} where a is a constant that denotes the number of pipe stages between fetching and execution dates. Therefore, instructions can be fetched simultaneously and make a group only if they can be executed concurrently. This leads to empty groups or variable size groups.

In fact, actual processors don't comply this model since instructions may remain valid in some stages for multiple cycles waiting for a resource or stalling for other reasons. They always fetch up a constant number of instructions to form a group, this constant is called the *degree* of the superscalar architecture. Nevertheless instructions in a group are not always executed concurrently. If a conflict occurs, some of them wait in their static stages. When referring to the conventional model, one can say that some instructions are fetched in advance. Thus groups of variable size provide a good fundation for developing an execution model since the processor can cope with them afterwards.

This optimistic point of view is not always realistic since it tends to produce less optimized code. Assume that the instructions A and B belong to the group G_1 and the instructions C and D belong to the group G_2. Let us consider a microarchitecture of degree 3 (a group includes 3 instructions). Moreover, a delay of 1 cycle is required between the execution stages of A and C. If the group G_1 is completed by a *no-op* instruction and A and B don't need any waiting cycles, the group G_2 can be executed one cycle after the group G_1.

If the group G_1 is completed by the instruction C fetched in advance, it freezes the pipe since a delay must be preserved between A and C. Thus the best execution of the group G_2 can begin 2 cycles later after G_1 has begun executing. Moreover such a delay may propagate in the execution timings of the instructions that depend of C.

An execution that complies the optimistic schedule can be achieved inserting appropriate *no-ops*. They must be chosen to fill up the empty pathes among the set of execution pathes.

In this approach, the number of *no-ops* required to prevent the processor from fetching inadequate instructions becomes excessive causing unacceptable increase in code size and leading to instruction cache misses.

2.2 Abstract model of a superscalar processor

The machine model our algorithm uses capture the essential of widespread superscalar architectures. The abstract logical structure of the microarchitecture is partitioned into logical chunks. Once a chunk starts executing, it runs to completion with a constant execution time; on the contrary variable waiting times may occur between two chunks. Thus a sequence of dynamic stages can be considered as a unique chunk while our vision for multi-issue architecture can be represented by parallel pathes between chunks.

On most architectures, there exists a stage S in the pipeline from which if the latencies are satisfied between instructions, they flow through the pipe without any waiting time. It follows that the sequence of stages from S to the end may build one single dynamic chunk. Thus no loss of generality is incurred if on each path of execution, the only one dynamic chunk is the last one.

As we want to model the run time behaviour of instructions, a unique execution path has to be associated to each codeop of instruction. Thus there are as many dynamic chunks as there are different classes of instruction behaviour.

This is exemplified by add integer instructions that may be executed either in `E0` or `E1` in the 21164 microarchitecture while shifts are processed only by `E0` and

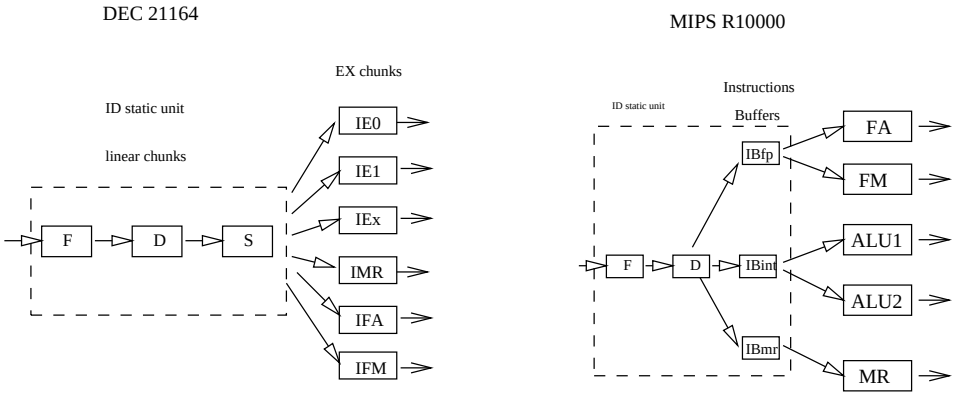

Figure 1: The abstract models of two architectures.

multiply only by `E1`. Therefore three execution chunks are necessary respectively named `E0`, `E1`, `Ex`. This splitting is similar to the one used in the processor handbook to disambiguate one execution path in multi-issue architecture.

The abstract logical structure of the 21164 is presented in Figure 1. The linear part of the pipeline decodes up to four instructions in parallel and checks that the required resources are available for each instruction. It issues only the instructions for which all required resources are available, afterwards the pipeline splits into as many execution chunks as abstract execution units.

A chip is divided into a linear part which may include several chunks. For purpose of legibility, we will denote ID static unit this linear part.

The ID static unit feeds the pipe while EX chunks are responsible for performing computations. The ID static unit feeds the pipe, execute all static pipeline stages,i.e those where bubbles may occur between instructions. It checks that the required resources are available, the instructions are issued.

This abstract model can be exemplified by two architectures of the DEC ALPHA family: the 21064 and the 21164. For both architectures the ID static unit includes 3 chunks, the difference between the two can be found in the dynamic part, the 21064 has only two EX chunks while the 21164 provides six ones.

Two important issues can be tackled by our model : the instruction fetching policy (dynamic for the MIPS R10000 or static for the DEC 21164) and the in order or out of order completion policy as well. Both of them can be expressed by

constraints on the input dates of some chunks in the ID unit for any instruction in a group.

In a MIPS R10000, instructions are fetched in order at the rate of up to four instructions. Once the instructions have gone through the static chunks, they are buffered in dedicated buffers according to their types as shown in Figure 1. They are not necessarily dispatched and removed from these buffers in the same order.

The comparison between the DEC family and the MIPS exemplifies another significant difference. The ID static unit is not always linear : according to their classes, the instructions are queued in different buffers before they are sent into dynamic chunks for execution. The delay between the output date of this buffer and the input date to the associated dynamic chunk for a given instruction is not a constant. It follows that an instruction buffer can be modelled by a static chunk.

Referring again to our abstract model, we need to express the constraints that instructions must satisfy to avoid data and structural hazards. This can be simply stated by delays between the input dates of the EX chunks of two instructions.

Since the fetching and grouping strategies can be entirely expressed in terms of the input dates of the ID unit chunks, once all the EX stages of the instruction sequence are scheduled, the holes in the groups can be filled using the waiting ability of the ID chunks instead of *no-ops* instructions. The motivation for fetching instructions that would freeze the pipeline is to avoid some insertion of *no-ops*.

3 A scheduling strategy using our machine model

3.1 Run time specification of a serial stream of instructions

Returning to our abstract model, we have now to specify the behaviour of a serial stream of instructions according to the run time point of view. For that purpose, variables that are input dates to the chunks for the path followed by an instruction during its execution are introduced. We need some more definitions to go further in the process of modeling the run time of instructions in the abstract organization.

A path in our machine model is a sequence of static chunks followed by a unique dynamic chunk (see 2.2). This is modelled by a *spine*.

Definition 1 *A spine of name N is a sequence of L chunks from the source of the ID unit to the sink of the EX execution unit representing the path of name N. The spine is denoted*

$$CS_1, \ldots, CS_i, \ldots, CS_{L-1}, CD_L$$

The spine name is similar to the one used in the processor handbook to disambiguate one execution path in multi-issue architecture. It is used to identify a spine.
We will denote *critical sub-spine* the sequence of static chunks of a spine because they play a central role in the scheduling process. The last chunk of the critical sub-spine is called *issue chunk* since it is the last one that can be affected by a stall.

Since there is a bijection between the dynamic chunk of a spine and a given spine, we will denote them the same way in the sequel.

In order to model the run time execution of a serial stream of instructions each of them has to be associated with a unique spine in order to determine the executing path. The run time model of an instruction must also include input dates to each chunk of the spine. Finally a *virtual pipe* identifies the behaviour of the run time of the instruction k along a spine SP of name N.

Let $\mathcal{C}$ be the set of instruction codeops and $\mathcal{S}_n$ the set of spine names. The mapping $sn : \mathcal{C} \rightarrow \mathcal{S}_n$ associates a unique spine name with each codeop.

Definition 2 *A virtual pipe $V = (D, N)$ is a couple where D is a vector of L variables associated to each chunk of the spine called N. Let $\mathcal{I}ns$ be the set of instructions of a serial stream. vp is the mapping that associates a virtual pipe to an instruction :*

$$\begin{array}{rccl} vp: & \mathcal{I}ns & \rightarrow & \mathcal{V} \\ & k & \rightarrow & \mathcal{V}_k = (D, N) \end{array}$$

where $N = sn(codeop(k))$
The input date of a chunk is given by the mapping dc :

$$\begin{array}{rccl} dc: & \mathcal{V} \times I & \rightarrow & \mathcal{V}ar \\ & \mathcal{V}_k, i & \rightarrow & D_i \end{array}$$

where $\mathcal{V}$ is the set of virtual pipes associated with the instructions of a serial stream and $\mathcal{I}$ is the set of integers $\{1, \ldots L\}$ and D_i the ith variable of D in $\mathcal{V}_k$. D_i denotes the input date to the ith chunk of the virtual pipe.

The material to express the two classes of constraints between input dates of a sequence of instructions is now ready. Among target machine dependent relationships, we must express concurrent execution properties. The first one concerns the superscalar feature of the architecture, i.e the number of instructions that may be processed concurrently in the same chunk at a given time.

Property 1 *Let B be a serial stream of instructions. Along the critical sub-spine at any time, groups of instructions can be processed concurrently in chunk i iff*

$\forall t \geq 0, \forall i \in [1, L-1], \quad \mathbf{Card}(\{k \in B \mid \quad dc(vp(k), i) = t\}) \leq max(i)$

For the MIPS R10000, up to 16 instructions can be held in the floating point queue modelled by the chunk `IBfp` *in Figure 1.*

The second one is related to the replication of spines depending on the codeops of the instructions.

Property 2 *At any time for a given spine n, a dynamic chunk is used by atmost a_n instructions where a_n is a constant that denotes the number of replications of the spine n.*

$$\forall t \geq 0, \forall n \in \mathcal{S}_n, \quad \mathbf{Card}(\{k \in B \mid sn(codeop(k)) = n \ \ and \ \ dc(vp(k), L) = t\}) \leq a_n$$

For most chunks in superscalar processors, $a_n = 1$ because execution pipes are not fully replicated.

The last class of properties is about concurrent execution of chunks.

Property 3 *The concurrent execution of dynamic chunks is architecturally defined with respect to rules generally called slotted rules. Let $\mathcal{SL}$ be the set of valid subsets of spine names that satisfy concurrency rules.*

At any time t, the set of spine names that may be executed concurrently must belong to $\mathcal{SL}$

$$\forall t \geq 0, \ \{\exists k \in \mathcal{S}_n \mid sn(codeop(k)) = n \ \ and \ \ dc(vp(k), L-1) = t \in \ \mathcal{SL}\}$$

This is exemplified by integer add instructions that pass either through the `E0`, `E1` or `Ex` chunks in the 21164. The set of valid partitions of these chunks that satisfy concurrency rules is the following

$$\{\{\texttt{Ex}, \texttt{Ex}\}, \{\texttt{Ex}, \texttt{E1}\}, \{\texttt{Ex}, \texttt{E0}\}, \{\texttt{E0}, \texttt{E1}\}, \{\texttt{Ex}\}, \{\texttt{E0}\}, \{\texttt{E1}\}, \{\}\}$$

The reverse mapping from a spine to an actual execution box is easy to define from the priority rules given in the processor handbook for the use of actual execution boxes.

Another category of relationships concerns the time that may elapse between two chunks along a spine for a given instruction.

Property 4 *For two consecutive chunks the following holds between the input date of a chunk and the input date of the following one since slack holes may occur between chunks of the critical sub-spine. Let CS_i be the ith chunk of the spine of name N_k*

$\forall i \in [1, L-2], \forall k \in \mathcal{I}ns, \quad dc(vp(k), i+1) \geq dc(vp(k), i) + duration(CS_i)$

Now we have to express dependence conditions between instructions. This goal is achieved using latencies of instructions matrix for various classes of instructions that are provided by the reference manual architecture handbook. These latencies take into account all bypass implementations that provide quicker paths from one stage to another one whenever it is possible.

Property 5 *In our model, latencies between instructions can be expressed by inequations between input dates to issue chunks.*

If we assume that the latency between instructions can be simply expressed by a function, a very simple formulation of instruction dependencies for two instructions k and j is as follows:

$dc(vp(k),\ L-1)\ \geq\ dc(vp(j),\ L-1)\ +\ latency(k, j)$

Whenever the latency is not given by a function, a more complicated logical expression must be stated.

3.2 Our scheduling algorithm

The task of a software scheduler is to select the order of a sequence of instructions so that they execute correctly in a minimum amount of time. However a software scheduler requires an enormous amount of time to guarantee that a schedule is optimum. Such a guarantee requires that the scheduler exhaustively test the execution time of every possible correct schedule. Instead of taking this time consuming approach, we will choose an heuristics, as most scheduler approaches do. Our vision for the scheduling of a serial stream of instructions can be summarized stating that it computes the set of corresponding virtual pipes. The algorithm presented in Figure 2 strongly relies on the previous model proceeding in the following three steps:

- For each instruction, a spine and an issue date input are computed with respect to the constraints stated using the properties of the abstract model. These properties take into account the restrictions on spine names according to the codeop of the instruction and the dependencies between instructions at lines (1-8). The constraints on input dates of dynamic chunks are set with respect to property 3 at lines (9-12), then these dates are computed according to these constraints at lines (13-15).

- Once the dynamic input dates have been set up, the second step deals with grouping rules for input dates of all static chunks preceeding dynamic chunks. These grouping rules are target dependent, they have to be supplied for each superscalar architecture because their fine granularity may imped stalls between dynamic chunks of the virtual pipeline.

  From the processor point of view, there exist specific rules of concurrent execution of instructions that must be applied in the context of a group of instructions. In our algorithm a peephole of instructions is used to represent the context. The rules for all the static chunks except for the first one (fetch chunk), are exactly translated into constraints involving input dates of static chunks between the instructions belonging to this peephole. These constraints are named *ID constraints* in the sequel. For the fetch chunk, the policy of the processor is to fetch a complete group of *degree* instructions while our algorithm allows an incomplete group.

  Given a peephole of instructions and an additional instruction, the algorithm sets the constraints on the input dates of its static chunks according to the *ID constraints* at lines (16-22). Finally, virtual pipes are sorted according to the input dates of dynamic chunks at line (23), and all input dates of static chunks are chosen at line (24)

- The third step fills up the empty slots of uncompleted groups with proper nops (line 25).

Let us consider an hypothetical machine of degree 3 which can be modelled with a static chunk and 3 concurrent dynamic chunks as shown in Figure 3. The benefit of such an heuristic can be illustrated comparing the run time of the serial stream of 4 instructions in the actual processor (in column TM) and the run time of the same stream when our algorithm is performed (in column VM).

Our scheduler relaxes the constraint to fetch always 3 instructions at the same time. Therefore, 2 instructions only are fetched at cycle 1, then a nop instruction can be inserted in the slot of the first group. It follows that instruction D is ended at cycle 3 instead of cycle 4.

4 Constraint logic specification of the scheduler

A particularly valuable capacity of our target machine model is to capture the constraints between the whole stages of the pipelines in a uniform way. Moreover flexi-

```
--------- Create the set of virtual pipes V and set the constraints applying proterties 4
(1)       V = φ
(2)       for each I in Ins do
(3)           V ← V ∪ {vp(I)}
(4)           set constraints(property4, vp(I))
(5)       end do

--------- Dependencies constraints between the instructions
(6)       for each D = (i,j) in DEP do
(7)           set constraints(property5, i, j)
(8)       end do

--------- Slotting rules for the Exec chunks
(9)       for each time cycle t do
(10)          CE_t = ⋃_{V_k ∈ V} Exec_chunk(V_k)
(11)          set constraints(property3, CE_t)
(12)      end do

--------- Compute exec dates with respect to the previous constraints
(13)      for each V_k in V do
(14)          Exec_dates(dc(V_k, L))
(15)      end do

--------- Set static chunks dates according to the target machine processing rules
(16)      for each time cycle t do
(17)          for i ∈ [1, L − 1] do
(18)              CS_t(i) = ⋃_{V_k ∈ V} dc(V_k, i)
(19)              set constraints(property1, CS_t(i))
(20)              set constraints(property2, CS_t(i))
(21)          end do
(22)      end do

--------- Sort issue dates
(23)          VS = sort_issue_dates(V)

--------- Build groups and set static chunks dates
(24)          G = build_and_set_dates(VS)

--------- Complete the groups with nops
(25)          complete_groups(G)
```

Figure 2: The scheduling algorithm.

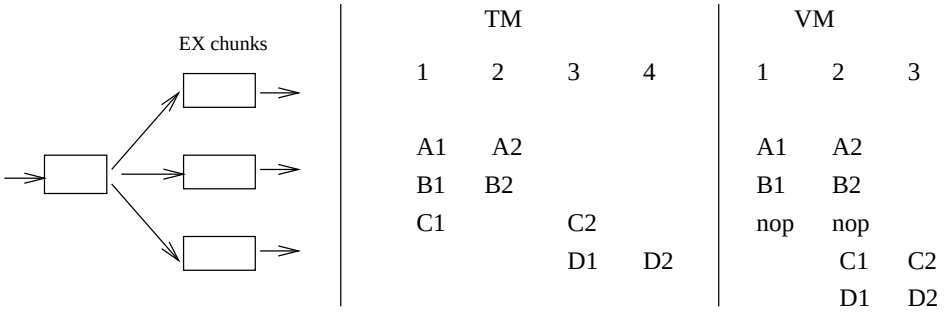

Figure 3: The abstract model of a mple machine.

ble capabilities represented by waiting times are not ignored, they allow to insert a minimal number of nops to enforce groups.

Flexible capabilities have been yet explored by means of a Constraint Logic Programming Language with finite domain constraints by [8] for the Motorola 88100, a scalar architecture which can issue only one instruction at a time. Fine grain parallelism is exploited within a basic block. The heuristic used for choosing the next variable is : choose the one with the smallest domain and the largest number of constraints. No specific heuristic is used to determine what value of the domain is chosen first.

This idea has not received further exposure in the literature because it does not rely on unrestricted multi-issue architectures models. The abstract model we propose offers capabilities to support any superscalar target architecture and a CLP with finite domains implementation can be straightforward derived from the abstract model.

This approach is of practical interest since prototypes have been implemented easily from such machine models for two DEC architectures : the 21064 and the 21164. For that purpose, we use CLP(FD) a Constraint Logic Programming Language with finite domain constraints[5] [6]. A finite domain variable behaves like an ordinary logic variable except that it can be instantiated only with values from its domain. The creation of virtual pipes is staightforward using finite domain variables for input dates of chunks. Properties 4 are simply expressed using inequalities between the FD variables that are associated to chunks. See below a simple definition of the floating-point add virtual pipeline of the DEC 21164.

```
virtual_pipe(floata, [F, D, S, I]) :-
    F in 0..MAX, D in 0..MAX, S in 0..MAX,
    I in 0..MAX_cycle
    D #> F + 1, S #> D + 1, I #> S + 1.
```

The critical sub-spine includes 3 chunks, the duration of each of them is one cycle, F, D, S are their related input date variables. I is the dynamic chunk.

The implementation must include as many virtual pipe specifications as there are spines in the target machine model. For each instruction, an instance of the virtual pipe related to its codeop is created.

The scheduler works narrowing domains according to the various constraints : data dependence constraints and architecture constraints. The running time of a scheduler which uses a symbolic constraint *depth-first branch and bound* to find the minimum optimal value of list of solutions is generally unacceptable.

A most realistic behaviour of the running time of the scheduler relies on the `labeling` symbolic constraint only. It gives a rather good approximation of the optimal solution provided that the list of FD variables given to the `labeling` predicate are in a pretty good order. This can be easily done using an heuristics.

Concurrency constraints can be stated using the `atmost` predicate: first the list `L_FD` of FD variables related to the chunks that belong to concurrent spines is built, second for each time cycle `T` the constraint `atmost(DEG,L_FD,T)` is set. `DEG` denotes the maximum amount of parallelism that may occur for this list of variables.

It is not very efficient since it must be set for each time cycle. CLP(FD) provides the much more efficient predicate `alldifferent` that assigns different values to a list of FD variables.

If we assume that all spines represent actual pipes and none of them is replicated, concurrency constraints are rather simple to state: for each spine `N`, first, build the list `L` of FD variables that belong to the dynamic chunk related to spine named `N`, next set `alldifferent(L)`.

Unfortunately the 21164 does not comply this model since basic integer arithmetic operations are supported by `E0` and `E1` as well as loads instructions from the memory. The integer multiplier and the shifter are only supported by `E0` while `E1` is responsible of write instructions and branches. In the abstract model, the three spines `E0`, `E1` and `Ex` have been designed to support these capabilities. However at a given time cycle, only two among these last three can be executed concurrently. We have introduced boolean variables to implement the disjunction choices as it is widely used in the operations research community [7]. We introduce two additionnal

FD variables in the virtual pipe denoted A0 and A1. Their values are either 2 times the value of I or 2 * I + 1. If A0 = 2 * I, it means that E0 is the actual pipe used by the instruction otherwise A1= 2 * I and E1 is used.

```
virtual_pipe(E0, [F,D,S,I,A0,A1]) :-
    F in 0..MAX, D in 0..MAX,
    S in 0..MAX, I in 0..MAX,
    A0 in 0..2*MAX, A1 in 0..2*MAX
    S #>= F + 1,
    D #>= S + 1,
    I #>= D + 1,
    A0 #= 2 * I,
    A1 #= 2 * I + 1.
```

```
virtual_pipe(E1, [F,D,S,I,A0,A1]) :-
    F in 0..MAX, D in 0..MAX,,
    S in 0..MAX, I in , 0..MAX,
    A0 in 0..2*MAX, A1 in 0..2*MAX
    S #>= F + 1,
    D #>= S + 1,
    I #>= D + 1,
    A0 #= 2 * I + 1,
    A1 #= 2 * I.
```

```
virtual_pipe(Ex, [F,D,S,I,A0,A1]) :-
    F in 0..MAX, D in 0..MAX, S in 0..MAX,
    BOOL in 0..1, A0 in 0..2*MAX, A1 in 0..2*MAX
    I in 0..MAX,
    D #>= F + 1,
    S #>= D + 1,
    I #>= S + 1,
    A0 #= 2*I + BOOL,
    A1 #= 2*I + 1 - BOOL.
```

Now the concurrency constraints can be stated with the following scheme. In a first phase, the list L0 of FD variables that belong to A0 related to both spines E0 and Ex is built. In a second phase, the constraints are set with alldifferent(L0). A similar process is done for the FD variables that belong to A1 related to the spines named E1 and Ex.

Concurrency between dynamic chunks can be implemented this way. Another important issue is about the setting of constraints for concurrency between static chunks.

This encoding allows to state constraints in parallel and finally the instantiation of the boolean variable BOOL is done by the labeling procedure. Boolean constraints are never backtracked and bound propagation proceeds much more efficiently inside the constraint solver.

Because of the relaxation on the fetch policy, the implementation includes predicates expressing alternative constraints involving the fetch chunks. The backtracking

of Prolog allows the choice of the next alternative in the case where the first alternatives used for an instruction violates the set of constraints already set. At the end of this process, the domains of static input dates are very restricted; a simple labeling is enough to set all input dates.

5 A complete example

In this section, we illustrate our algorithm with the piece of code of Table 1 that has been produced by the `cc` compiler on a DEC ALPHA for the following expression from the k22 livermore loop:

```
qa = za[k][j+1]*zr[k][j] +za[k][j-1]*zb[k][j]
```

In the assembler's instruction set, the destination of each instruction is generally the rightmost operand except for loads (with the `ldl`, `lds`, `lda` mnemonics).

5.1 Instruction latencies for the 21164

The latency to produce a valid result for most instructions is generally a constant; for a given class of instructions it can be found in column 2 of Table 2. For example, `ld` denotes the instruction class of loads: (`ldl`, `lds`, `lda`), the latency for any producer instruction of this class is 2. The class of integer additions denoted `iadd` includes the instructions of mnemonics: (`addq`, `s4addq`, `subq`,...); `shift` denotes the class of shift instructions.

A counterexample can be found for multiply instructions because they are dependent on which previous instructions produced its operands and when they are executed. Figures in column 3 of Table 2 show for each class of instructions the additional time before their result is available to the integer multiply unit.

Let us consider our example, it is easy to see that the data dependency between instructions `i1` and `i2` via register `$24` incurs a latency of two cycles because the producer instruction `i1` belongs to the `load` class.

A predicate is generated for each couple of instructions that are depending on each other in the following way:

```
delay([i1,i2,2]).
```

Whenever an instruction producer belongs to the instruction class `imulq` and there is no additional time, the latency between an integer multiply instruction and a depending one is 12. `(i4, i5)` is an example of such a couple of instructions.

i1	ldl	$24, 1512($sp)
i2	addl	$24, 1, $25
i3	ldl	$1, 1504($sp)
i4	mulq	$1, 40, $4
i5	s4addq	$25, $4, $27
i6	lda	$2, 1248($sp)
i7	addq	$27, $2, $3
i8	lds	$f27, 0($3)
i9	mulq	$24, 4, $5
i10	addq	$4, $5, $22
i11	addq	$22, $sp, $23
i12	lds	$f28, 48($23)
i13	mult	$f27, $f28, $f29
i14	addl	$24, -1, $6
i15	s4addq	$6, $4, $7
i16	addq	$7, $2, $8
i17	lds	$f30, 0($8)
i18	addq	$4, $5, $25
i19	addq	$25, $sp, $27
i20	lds	$f10, 288($27)
i21	mult	$f30, $f10, $f11
i22	addt	$f29, $f11, $f12
i23	sts	$f12, 40($sp)

Table 1: A piece of assembly instructions for the 21164.

Class	Latency	Additional Time
ld	2	1
st		–
iadd	1	2
ilog	1	2
shift	1	2
icmp	1	2
imull	8	1
imulq	12	1
imulh	14	1
fadd	4	–
fdiv	15	–
fmul	4	–

Table 2: Instruction latencies for the DEC Alpha 21164.

Otherwise special purpose predicates that deal with context dependent additional latencies are used to compute true dependencies on multiply instructions.

The data dependency graph with latencies between the instructions of Table 1 are shown in Table 3.

5.2 Spines related to the 21164

Instructions in a class behave identically with respect to their related spine. The following spines are defined on the various classes :

`Ex` with [`iadd, ilog, nop, icmp, ld`].

`E0` with [`shift, imull, imulq, st`].

`E1` with [`br`].

`FA` with [`fadd, fnopa`].

`FM` with [`fmul, fnopm`].

Once a schedule is computed, it is straightforward to compute the mapping from spines to actual pipes.

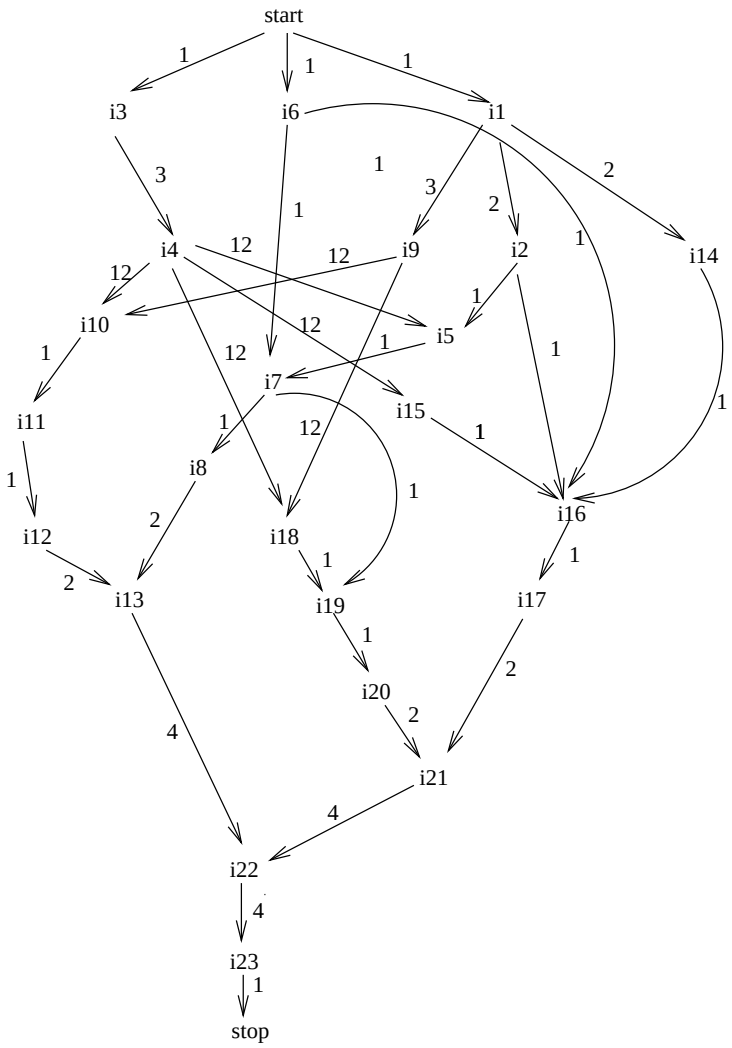

Table 3: The data dependence graph of the sequence of code.

5.3 The run time behaviour of the example sequence

The output of our algorithm is shown in part `(a)` of Figure 4. Part `(b)` of this figure is a shorthand representation of what happens in the hypothetical processor where there are no waiting times between the static chunks.

In the figures of this section, we choose to represent each dynamic chunk by a string of characters which length is equal to the duration of this chunk. Furthermore, this string catenates a mnemonic of the spine used and the mapping to the actual pipe. `IxOW`, `Ix1W` denote the dynamic chunk Ex that are respectively performed by the `E0` and `E1` actual pipes for integer instructions. `I` always denotes the input time of the dynamic chunk and `F`,`D`,`S` respectively denote the static stages of the 21164 with duration 1.

Therefore, the comparison between the run time in the hypothetical processor and in the actual processor will be more understandable using such a concatenation of mnemonics to denote the dynamic chunk.

Dynamic chunks relative to floating point aditions are denoted `IFAxxxW` with respect to the number of clock cycles required for execution. Similarly, floating point multiply dynamic chunks are denoted `IFMxxxW`.

Waiting times between static chunks are materialized by dots.

Figure 5 is a shorthand representation of what really happens in the hardware for the stream of instructions output of our algorithm. It means that the constraint of fetching four intructions in the same cycle that has been relaxed by our algorithm is now taken into account.

Part `(a)` of Figure 5 shows the execution of the schedule output. Part `(b)` of the same figure shows the same example when a `nop` is inserted after instruction `i6`. It exhibits that the code example takes one cycle longer to execute when no `nop` instruction is inserted after `i6`.

A delay of 2 clock cycles must be satisfied between the instructions `i3` and `i2`. If they are fetched in the same group, `i2` causes a pipeline freeze that has consequences on the instructions of the following group. The in-order issue policy implies that `i14` which belongs to the next group, cannot be issued at the same cycle than `i2` but at least one cycle after. Therefore, the instruction `i14` is delayed until time `7`. This propagates on the instructions `i4` and `i9` which enforce stronger constraints on the whole block.

When a `nop` is inserted after the instruction `i6`, `i2` and `i14` that belong to the same group and have no dependencies, can be issued simultaneously and hence the instruction `i14` is issued at cycle `6`.

(a) Output of our scheduling algorithm

```
ld      i3    FDSIx0W
ld      i1    FDSIx1W
iadd    i6    FD.SIx0W
iadd    i2    -F.DSIx0W
iadd    i14   -F.DSIx1W
imulq   i4    -F.D.SI00W
imulq   i9    -F.D..S.......I00W
iadd    i5    -+-F..D.......S...Ix0W
iadd    i15   -+-F..D.......S...Ix1W
iadd    i7    -+-F..D...........SIx0W
iadd    i16   -+-+F..D...........SIx1W
ld      i8    -+-+-+F...........DSIx0W
ld      i17   -+-+-+F...........DSIx1W
iadd    i10   -+-+-+F...........D.S.....Ix0W
iadd    i18   -+-+-+F...........D.S.....Ix1W
iadd    i11   -+-+-+-+-+-+-+-+-+F.D.....SIx0W
iadd    i19   -+-+-+-+-+-+-+-+-+F.D.....SIx1W
ld      i12   -+-+-+-+-+-+-+-+-+F.D......SIx0W
ld      i20   -+-+-+-+-+-+-+-+-+F.D......SIx1W
fmul    i13   -+-+-+-+-+-+-+-+-+F.......DS.IFMxxxW
fmul    i21   -+-+-+-+-+-+-+-+-+F......D..SIFMxxxW
fadd    i22   -+-+-+-+-+-+-+-+-+F......D..S....IFAxxxW
st      i23   -+-+-+-+-+-+-+-+-+-+-+-+-+-F..D.........SI00W
```

(b) Run time behaviour for the virtual processor with no static stages

```
ld      i3    FDSIx0W
ld      i1    FDSIx1W
iadd    i6    -FDSIx0W
iadd    i2    -+FDSIx0W
iadd    i14   -+FDSIx1W
imulq   i4    -+-FDSx00W
imulq   i9    -+-+-+-+-+-FDSI00W
iadd    i5    -+-+-+-+-+-+-+-FDSIx0W
iadd    i15   -+-+-+-+-+-+-+-FDSIx1W
iadd    i7    -+-+-+-+-+-+-+-+FDSIx0W
iadd    i16   -+-+-+-+-+-+-+-+FDSIx1W
ld      i8    -+-+-+-+-+-+-+-+-FDSIx0W
ld      i17   -+-+-+-+-+-+-+-+-FDSIx1W
iadd    i10   -+-+-+-+-+-+-+-+-+-+-+-FDSIx0W
iadd    i18   -+-+-+-+-+-+-+-+-+-+-+-FDSIx1W
iadd    i11   -+-+-+-+-+-+-+-+-+-+-+-+FDSIx0W
iadd    i19   -+-+-+-+-+-+-+-+-+-+-+-+FDSIx1W
ld      i12   -+-+-+-+-+-+-+-+-+-+-+-+-FDSIx0W
ld      i20   -+-+-+-+-+-+-+-+-+-+-+-+-FDSIx1W
fmul    i13   -+-+-+-+-+-+-+-+-+-+-+-+-+-FDSIFMxxxW
fmul    i21   -+-+-+-+-+-+-+-+-+-+-+-+-+-+FDSIFMxxxW
fadd    i22   -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+FDSIFAxxxW
st      i23   -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-FDSI00W
```

Figure 4: Run time behaviour of the code example.

Finally, Figure 5 shows that the execution of the code example requires `45` cycles in part `(a)` and `44` cycles in part `(b)`.

6 Conclusion

The use of CLP(FD) for prototyping provides flexible, ease of development mainly because the abstract model of the target machine can be implemented straight-away using finite domain variables. Furthermore, constraint propagation and constraint solving are abstracted away from the programmer. Nevertheless, the main drawback of CLP(FD) is its inefficiency whenever too many variables and equations are involved.

We can briefly say that our approach models rather the concurrency of tasks than the concurrent usage of resources in the processor. It follows that the concepts underlying in our target machine description are more high level than the other approach. Another consequence is a smaller size of the specification.

Experiments with the DEC 211164 and 21064 scheduler indicate that the scheduled code is executed with a speedup comparable to usual schedulers. Furthermore the number of nops necessary to improve the speedup of the code scheduled is small, hence the benefit of slackness in our model has been demonstrated.

References

[1] Mario Tokoro, Eiji Tamura, Takashi Takizuka. Optimization of Microprograms. In *IEEE Transactions on Computers 30(7)*, pages 491–504, July 1981.

[2] Alexandre Eichenberger, Edward Davidson. A Reduced Multipipeline Machine Description that Preserves Scheduling Constraints. In *ACM SIGPLAN '96 Symp. on Compiler Construction*, pages 12–21, Philadelphia, Pen, May 1996.

[3] T.Muller. Employing finite automata for resource scheduling. *Proc. of the 26th Annual Symposium on Microarchitecture*, pages 12-20, Austin, Texas, Dec 1993.

[4] T.A. Proebsting, C.W. Fraser. Detecting pipeline structural hazards quickly. In *ACM SIGPLAN SIGACT'96 Symp. on POPL*, pages 280-286, Portland, Oregon, Jan 1994.

[5] P. Codognet and D. Diaz. Compiling Constraint in `clp(FD)`. *Journal of Logic Programming*, Vol. 27, No. 3, June 1996.

```
       (a)  Fetching the instructions like the processor without nops :
            The issue cycle of i23 is 41

ld      i3    FDSIx0W
ld      i1    FDSIx1W
iadd    i6    FD.SIx0W
iadd    i2    FD.S.Ix0W
iadd    i14   -F.D.SIx0W
imulq   i4    -F.D..SI00W
imulq   i9    -F.D...S.......I00W
iadd    i5    -F.D...S...........Ix0W
iadd    i15   -+-F...D...........SIx0W
iadd    i7    -+-F...D...........SIx1W
iadd    i16   -+-F...D............SIx0W
ld      i8    -+-F...D............SIx1W
ld      i17   -+-+-+-F............DSIx0W
iadd    i10   -+-+-+-F............DS.....Ix0W
iadd    i18   -+-+-+-F............D......SIx0W
iadd    i11   -+-+-+-F............D......SIx1W
iadd    i19   -+-+-+-+-+-+-+-+-+-+F......DSIx0W
ld      i12   -+-+-+-+-+-+-+-+-+-+F......DSIx1W
ld      i20   -+-+-+-+-+-+-+-+-+-+F......D.SIx0W
fmul    i13   -+-+-+-+-+-+-+-+-+-+F......D.S.IFMxxxW
fmul    i21   -+-+-+-+-+-+-+-+-+-+-+-+-+-+-F.D.SIFMxxxW
fadd    i22   -+-+-+-+-+-+-+-+-+-+-+-+-+-+-F.D.S....IFAxxxW
st      i23   -+-+-+-+-+-+-+-+-+-+-+-+-+-+-F.D.S........I00W

       (b)  Fetching the instructions like the processor with nops :
            The issue cycle of i23 is 40

ld      i3    FDSIx0W
ld      i1    FDSIx1W
iadd    i6    FD.SIx0W
N       nop   FD.SIx0W
iadd    i2    -F.DSIx0W
iadd    i14   -F.DSIx1W
imulq   i4    -F.D.SI00W
imulq   i9    -F.D..S.......I00W
iadd    i5    -+-F..D.......S...Ix0W
iadd    i15   -+-F..D.......S...Ix1W
iadd    i7    -+-F..D...........SIx0W
iadd    i16   -+-F..D...........SIx1W
ld      i8    -+-+-+F...........DSIx0W
ld      i17   -+-+-+F...........DSIx1W
iadd    i10   -+-+-+F...........D.S.....Ix0W
iadd    i18   -+-+-+F...........D.S.....Ix1W
iadd    i11   -+-+-+-+-+-+-+-+-+F.D.....SIx0W
iadd    i19   -+-+-+-+-+-+-+-+-+F.D.....SIx1W
ld      i12   -+-+-+-+-+-+-+-+-+F.D......SIx0W
ld      i20   -+-+-+-+-+-+-+-+-+F.D......SIx1W
fmul    i13   -+-+-+-+-+-+-+-+-+F......DS.IFMxxxW
fmul    i21   -+-+-+-+-+-+-+-+-+F......D..SIFMxxxW
fadd    i22   -+-+-+-+-+-+-+-+-+F......D..S....IFAxxxW
st      i23   -+-+-+-+-+-+-+-+-+F......D..S........I00W
```

Figure 5: Execution of the code example, fetching instructions like the processor.

[6] D. Chemla, D. Diaz, P. Kerlirzin and S. Manchon. Using `clp(FD)` to Support Air Traffic Flow Management. In *3rd International Conference on the Practical Application of Prolog*, Paris, France, 1995.

[7] S. Breitinger and H. Lock. Using Constraint Logic Programming for Industrial Scheduling Problems. *Logic Programming: Formal Methods and Practical Applications. edited by C. Beierle and L. Plumer Elsevier Science Publishers*, 1994.

[8] Anton Ertl, Andreas Krall. Optimal Instruction Scheduling using Constraint Logic Programming. *Proceedings of the PLIPL'91, pages 75-86.*

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399