Static Analysis and Scheduling for Asynchronous Processor Architectures

Thesis Proposal

Salvador Sotelo Salazar

September 27, 1996

Abstract

This work addresses the problem of compiler optimisations for ILP in multiple-issue asynchronous architectures. Instruction scheduling for a Micronet-based asynchronous target presents interesting challenges. First, the model of the underlying target is more complex: the functional units have latencies which ranges over an interval. Secondly, the dynamic behaviour of the asynchronous architecture makes it impossible to consistently predict the order of execution. Finally, the presence of several issue units implies the use of aggressive methods to increase ILP.

This thesis proposal presents some ideas for identifying good schedules and proposes future work in this area.

Contents

Chapter 1

Introduction

Asynchronous architectures differ from their synchronous counterpart in that the central control unit in the processor does not have a clock to synchronize the different stages. The method that asynchronous architectures use to control the flow of instructions along the datapath is usually a handshake that "communicates" between stages in the pipeline.

In particular, the datapath of MAP (Micronet-based Asynchronous Processor) architectures is modelled as a network of functional units (entities) that communicate with each other asynchronously [2]. This model shows fine grain concurrency -both spatial and temporal- so each instruction takes the necessary stages for its execution and only for the time needed. In the synchronous world, the instructions have to go through all the stages and even if they complete one stage sooner they have to wait for the next cycle to start another operation, keeping that resource idle.

It has been shown that reordering the instructions of a program in a RISC processor is effective in improving performances [21][12][14][17]. The reason for rearranging the order of the instructions in a code is to exploit the benefits of the pipeline and avoid stalls. This is achieved because the compiler has a precise model about the rate of execution of instructions. This determinism in synchronous RISC processors has helped the scheduling theory to substantially improve execution times [26].

However, in the asynchronous approach this determinism is not valid, therefore the way to schedule a program in a scalar processor is a large field for research. For instance, the difficulties in predicting the MAP's behaviour comes from the variations in the latencies and the possible out-of-order execution. The scheduling research done by the community has deeply considered the uniformity of the execution rate in high performance synchronous processors, so most of the work has to be reappraised and adapted, if possible, to the asynchronous approach in MAP.

The importance of *Instruction-level Parallelism* (ILP) in the code is a well known attribute in the scheduling theory, the more independent instructions we have, the more chances there are to allocate them in free units, and therefore avoid stalls. Eventually, the semantic content of a program destined to be executed in a scalar processor is quite sequential, even though there are some instructions that stay independent. It is our work to sort out the best way to rearrange the instructions before putting them in the final stream before execution.

A way to measure how much concurrency (ILP) is in a program would help to know how "concurrent" that program is (depending on its nature) and therefore to know its bound. This is important due to the fact that if a *control intensive* program shows a "low" measure then we could expect slight improvements after scheduling it.

In order to know how many concurrent instructions we can get from a source code before trying any scheduling technique, we need an effective way of measuring the concurrency in the instructions. The main goal is to make some static analysis in the code to help us see the concurrent properties on it and from these make conclusions and take decisions to the former scheduler.

This thesis proposal presents the work conducted in the static analysis of programs. Firstly, we examine the concurrency measure for distributed computations [5] and extend the work for a scalar uniprocessor. Secondly, we propose a static and inexpensive method to classify the way a code has been scheduled, and evaluate its goodness after simulating it. Next, we show preliminary results from the proposed technique and its performance in a real scheduler. Finally, we establish the different paths to follow and future work for the realisation of a scheduler for MAP.

Chapter 2

Concurrency Measure

2.1 Background Theory

The method used to compute the concurrency measure in a code is proposed in [5]. It uses the antichain lattice of the partial order to model the distributed execution. The principal feature over other measures is that the measure can be computed for individual events.

In this chapter we describe the necessary framework in which the concurrency measure has been defined. Then we mention the considerations taken, as this measure is oriented to regular distributed programs and our objective is to apply it to a scalar uniprocessor.

2.1.1 Partial order and lattice

A partially ordered set, E, is denoted by $\langle E; \leqslant \rangle$ and is commonly named *poset*, for short. $\leqslant$ is the partial-order relation associated with E.

$x, y \in E$ are said to be *comparable* if $x \leqslant y$ or $x \geqslant y$, and *incomparable* (or $x \| y$) when $x \nleqslant y$ or $x \ngeqslant y$. E is a *chain* if all $x, y \in E$ are comparable. Conversely, if every $x, y \in E$ are incomparable, then E is an *antichain*.

The *covering relation* in a poset E is defined when for $x, y, z \in E$, $x < y$ and there is no z where $x < z < y$. It is said that x is *covered* by y (or y *covers* x) and is noted as $x \multimap\!\!< y$ or $y \succ\!\!- x$. Due to the nature of the work and just for a matter of simplification, we will refer to the covering relation in $\mathbb{N}$ [1].

The graphical representation of a poset E is a directed graph, where the nodes represent the elements of E and the edges the covering relation from a pair of E. Such a graph is called *Hasse Diagram*.

For any given $x \in E$, $\downarrow e$ (read 'down x') is the set of predecessors of x, where $\downarrow e = \{y \in E \mid y \leqslant x\}$, and for any given subset F of E, the set

[1] $\mathbb{N} = \{1, 2, 3, ...\}$

of predecessors $\downarrow F = \bigcup_{y \in F} (\downarrow y)$. Similarly, the set of successors of x is $\uparrow e = \{y \in E \mid y \geqslant x\}$, and $\uparrow F = \bigcup_{y \in F} (\uparrow y)$ for an arbitrary subset F of E.

We denote $x \vee y$ for $\sup\{x, y\}$ and $x \wedge y$ for $\min\{x, y\}$. If for all $x, y \in E$, $x \vee y$ and $x \wedge y$, E is called a *lattice*. A lattice is called a *distributive* lattice, if and only if, for any $x, y, z \in E$, $\ a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$.

We define the tuple $\langle E; \leqslant; P; \varphi \rangle$ as a *labelled* poset Θ from a non-empty poset $\langle E; \leqslant \rangle$ with a non-infinite width ($w(E)$ being the largest antichain in E), with a finite decreasing chain and a non-empty set of processes P. φ is the labelling function defined as $\varphi\colon E \longrightarrow P$.

$\mathcal{A}(\Theta)$ is said to be the set of antichains of Θ and is known to be a distributive lattice with the partial order property $\sqsubseteq$ which we define as: $\forall\, X, Y \in \mathcal{A}(\Theta),\ X \sqsubseteq Y \iff \downarrow X \subseteq \downarrow Y$.

To describe $\mathcal{G}(\Theta)$ we have to define $\mathcal{T}(\Theta)$ as $\mathcal{T}(\Theta) \subseteq \mathcal{A}(\Theta) \times E \times \mathcal{A}(\Theta)$, and is the set of edges (X, e, Y) such that $(\downarrow Y) \setminus (\downarrow X) = \{e\}$.

$\mathcal{G}(\Theta)$ is called the **labelled Hasse diagram** and its definition is $\mathcal{G}(\Theta) = \langle\, \mathcal{A}(\Theta), \mathcal{T}(\Theta)\, \rangle$.

2.1.2 Discrete model of a distributed computation

The former approach used in [5] and [23] is oriented to a distributed platform where N independent *processes*, $(P_i)_{i \in \mathbb{N}}$, run in parallel and communicate *asynchronously* by exchanging messages. It is assumed that N, the number of processes, will run on N processors.

The labelled poset Θ, defined previously, becomes the tuple $\langle E; <; P; \varphi \rangle$ called a *distributed* order. Now the elements of E leave the abstraction level and become 'precise' events through the computation. The partial order relation, $<$, expresses the order in which computations take place and how they are related, either sequentially or in synchronization events. φ now assigns the process identifier for each event.

The way in which this approach will be modified for an asynchronous scalar processor will be described in section 2.2.

The partial order relation $<$ differs from $\leqslant$ discussed above, because the former respects the order of execution in a continuous and strictly ascending way. In $<$, events depend on previous events so they cannot take place at the same time; they must evolve in some way. The notion of time on its own is not defined in the poset, even though there is a clear concept of ascending progression that keeps the computation order from the start to the end.

As we have said, if one event $b \in E$ depends on another event $a \in E$ ($a \prec b$ which means "a is before b"), b obviously takes place after a without depending in which process both events take place. Again, if $a \not\prec b$ and $a \not\succ b$, a and b have no causal relation, so they are concurrent ($a \| b$).

2.1.3 Measuring the antichain lattice

In figure 2.1 we can see from the lattice a computation with three processes (P_1, P_2 and P_3) and their actual events. We are interested in the sets whose events are not causally related (antichains) and therefore concurrent. For instance, we can see the following relations: $a \prec b$, $a \prec g$, $b \prec d, c \prec d$, $c \prec e$, $d \prec f$, $f \prec g$, $f \prec h$, $g \prec i$, $h \prec i$, $h \prec j$, $j \prec k$ and $e \prec k$. Similarly, if $a \prec b$ and $b \prec d$, we get $a \prec d$. With the transitive property we can get all the possible relations.

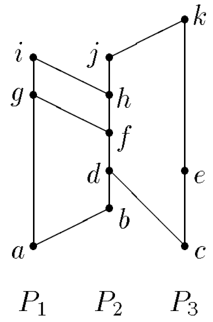

Figure 2.1: The distributive poset Θ.

Thus, the set of antichains $\mathcal{A}_\Theta(e)$ contains the set of "concurrent" events of e that we want to underline, and is defined as: $\mathcal{A}_\Theta(e) = \{X \in \mathcal{A}(\Theta) \mid e \in X\}$ read as the antichains that contain e. The proper concurrency measure is $\forall\, e \in X,\ \mu(e) = |\, \mathcal{A}_\Theta(e)\, |$.

In the example from fig. 2.1, $\mu(d) = |\, \{\{d\}, \{d, e\}\}\, | = 2$, which is clearly a synchronization event. On the other hand, event e's measure is $\mu(e) = |\, \{\{e\}, \{e, a\}, \{e, b\}, \{e, d\}, \{e, f\}, \{e, g\}, \{e, h\}, \{e, i\}, \{e, j\}, \{e, g, h\}, \{e, i, j\}, \{e, g, j\}\}\, | = 12$.

It is said that a "high" measure μ of some event (as is e in the previous example) is an event with a high number of concurrent events, and a "low" measure of μ (like in $\mu(d)$ above), is a point of strong synchronization. Events with a high measure can be scheduled in several places, and depending on other considerations (heuristics), their final position in the code will be eventually decided.

The methods for computing μ for each event consist in counting the edges (antichains) of the labelled Hasse diagram $\mathcal{G}(\Theta)$. An example of $\mathcal{G}(\Theta)$ can be seen in figure 2.5.

2.1.4 Vector clocks

For a number N of processes, $(P_i)_{i \in \mathbb{N}}$, there is an integer vector of length N associated with each process P_i that keeps the state of a "logical clock" and is defined as follows for $i \in N$:

$$C_{i+1}(P_i) = C_i(P_i) + 1 + max[C_i(P_i), T_j(P_j)].$$

$T_j(P_j)$ represents the vector clocks from all the events in other processes that synchronizes $C_i(P_i)$. In other words, the vector clock C_i of an event $e \in P_i$ is updated with a copy of its predecessor's vector clock in P_i, and incremented by one. Thus the event is taken a consecutive stamp applying the $<$ relation. To update e with all the synchronization events from other processes, a max function must be applied to all the elements of the vectors. Doing this will reflect the relation of sending a message from one process to another. Figure 2.2 shows how the poset Θ is represented with vector clocks.

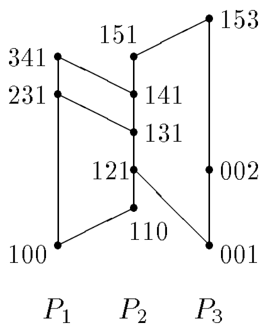

Figure 2.2: Vector clocks for the poset Θ.

2.2 Applying concurrency measures for the MAP Architecture

In our case, the program consists of a sequential stream of instructions that exposes fine grain concurrency. The ability of the MAP datapath to exploit both temporal and spatial concurrency [1][2] leads us to model it with a "micro-distributive" behaviour, in which N independent *units*, $(U_i)_{i \in \mathbb{N}}$, can execute instructions concurrently as they show some degree of independence. The asynchronous control in MAP allows the different units in the processor to operate freely, so that they are only regulated by their own latency as shown in [3]. We will ignore the micro-operations and the overheads in the communication protocol.

We wish to capture the concurrency at the instruction-level in a sequential program so an event $e \in E$ models an instruction from the program. It has to be executed in an unit that corresponds to its nature, e.g. an `add` instruction has to be executed by an arithmetical unit, and so on. The dependencies between instructions impose the order of execution, and act as synchronization events. The data shared in these operations are the registers of the processor as shown in fig. 2.3. These considerations respect the characteristics and properties of the poset $\Theta = \langle E; <; P; \varphi \rangle$.

An important fact is that if we consider an unlimited number of different resources (at least the number of instructions), the measure will reveal a concurrency based only on data dependencies, and give an upper limit. If we constrain the number of resources (as will surely be the case at some point) the measure obtained would take into account the data dependencies and some *resource* dependencies implied from the lack of resources. Obviously this measure would be much lower meaning a more sequential order of execution. The new resource dependencies would hide the possible ILP improvements from the compiler work and eventually would render an effective measure to any change. A measure that is useful for us is one that considers pure data dependencies, although in the future we will consider the target to see the concurrency *sensitivity*.

In the example in figure 2.3 we consider U_1 as a *memory* unit, U_2 as a *add/subtract* unit, and U_3 as a *multiply/divide* unit. The values with an `$` symbol correspond to real processor registers and the ones without it, represent just immediate numbers. The way that the source and destination are placed is like in many RISC-type instruction sets: the destination is the left and the operands are in the right (there are instructions with only one source operand).

a	`la`	`$12`	`$29`	`0`
b	`addui`	`$13`	`$12`	`8`
c	`muli`	`$14`	`$8`	`2`
d	`addu`	`$15`	`$14`	`$13`
e	`muli`	`$8`	`$9`	`4`

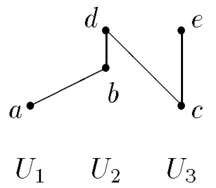

Figure 2.3: The representation of MAP's code in the poset Θ.

It can be seen that each instruction i matches every event e in the lattice and the data dependencies are respected conveniently. For example, instruction d requires registers `$13` and `$14` that are computed in instructions b and c respectively. Similarly, event d in the lattice, demands that the events b and c must have been computed beforehand.

The data dependencies considered are RAW (read after write) true dependency, WAR (write after read), false or anti-dependency, and WAW (write after write) output dependency, as in every scheduling study.

It is important to mention that for an *add/subtract* unit, for example, we do not make any distinction between an `addui` instruction and an `add` or `subui` instruction. If we define an *add/subtract* unit, only add and subtract instructions will be executed there; if it is an *add* unit, only add instructions can be executed.

These interpretations are necessary if we want the concurrency measure algorithm to give us a realistic and confident measure of the code we want to apply to.

The sets of antichains of $\mathcal{A}_\Theta(i)$ (where i is an instruction in the code, equivalent to an event e in the lattice) and their respective concurrency measures in the example of figure 2.3 are the following:

- $\mathcal{A}_\Theta(a) = \{\{a\},\{a,c\},\{a,e\}\} \implies \mu_\Theta(a) = |\mathcal{A}_\Theta(a)| = 3$
- $\mathcal{A}_\Theta(b) = \{\{b\},\{b,c\},\{b,e\}\} \implies \mu_\Theta(b) = |\mathcal{A}_\Theta(b)| = 3$
- $\mathcal{A}_\Theta(c) = \{\{c\},\{c,a\},\{c,b\}\} \implies \mu_\Theta(c) = |\mathcal{A}_\Theta(c)| = 3$
- $\mathcal{A}_\Theta(d) = \{\{d\},\{d,e\}\} \implies \mu_\Theta(d) = |\mathcal{A}_\Theta(d)| = 2$
- $\mathcal{A}_\Theta(e) = \{\{e\},\{e,a\},\{e,b\},\{e,d\}\} \implies \mu_\Theta(e) = |\mathcal{A}_\Theta(e)| = 4$

It can be seen that instruction e is more concurrent than any of the other four instructions, as it could occur concurrently either with a, b or d; conversely, instruction d is a point of synchronization from its low measure, meaning that it could occur concurrently only with instruction e.

The lattice clearly reflects the behaviour of the small code, as all the dependencies that the code expose are captured and respected.

2.3 Limits of the concurrency measure

2.3.1 Concurrency in different schedules

The intention to apply the concurrency measure algorithm to a sequential program is to see if, after some scheduling optimisations to it, we can obtain information that quantifies an improvement or a decrease in bulk over the original code.

Before thinking about the concepts and heuristics to overcome the scheduling problem of an asynchronous processor, we need to establish an evaluation workbench for any piece of code. In this way, we would be able to make distinctions between a potentially "good" scheduled code and another that is not so "good", and therefore evaluate whether or not the scheduling heuristic was better.

If we reconsider the example in figure 2.3, and move the order of some instructions respecting the semantics of the program and the number and type of units, in order to see any variation from the previous results, the code and its lattice would look like in the figure 2.4. As it can be seen, it shows a new order of the code which has an equivalent output to the previous version and respects the known dependencies. The units U_1, U_2 and U_3 stay dedicated to memory, *add/subtract* and *multiply/divide* operations respectively.

In this occasion, the first instruction `muli $14 $8 2` will be considered as the instruction a' with its equivalent event a' in the lattice. c means instruction a' was the third instruction in the code in the previous example, and so on.

a'	(c)	`muli`	`$14`	`$8`	`2`
b'	(e)	`muli`	`$8`	`$9`	`4`
c'	(a)	`la`	`$12`	`$29`	`0`
d'	(b)	`addui`	`$13`	`$12`	`8`
e'	(d)	`addu`	`$15`	`$14`	`$13`

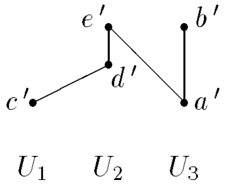

Figure 2.4: Another representation of an equivalent code from fig. 2.3.

After looking the lattice, we realise that it has the same shape as in fig. 2.3, but the events are placed in different positions. Naturally, we expect that this lattice will show similar concurrency measures for the new events.

Table 2.1 displays measures of $\mu_{(\Theta)}(i)$ for all the 10 instructions in both examples. It is clear that the values of $\mu_{(\Theta)}$ for a and c' are preserved in the same way as e and b', etc. We know that those pairs of events correspond to the same instructions: they are just placed in different parts of the code.

Instructions i	a	b	c	d	e		a'	b'	c'	d'	e'
$\mu_{(\Theta)}(i)$	3	3	3	2	4		3	4	3	3	2

Table 2.1: Concurrency measures from figures 2.3 and 2.4.

This observation enabled us to see any other differences in the measure to help us to make some distinction between both schedules.

Any metric in the concurrency measure shows the same numbers: for example, the mean $\overline{\mu_{(\Theta)}}(i) = 3$ and the variance $\sigma^2_{\mu_{(\Theta)}}(i) = 2/5$ are exactly the same, since every measure in both schedules is the same point-to-point, with the sole difference in the order.

After extensive checking with the different programs and with their possible schedules, this pattern stayed constant. For every allowed schedule in a code, every instruction has a proper concurrency measure and will be constant in any position in the code if all the dependencies are **respected**. This is proven in the next section.

2.3.2 Results

First of all, we have to prove that for any allowed schedule in a code we have the *same number* of instructions and the *same set* of them. To call two schedules *equivalent* (they produce the same output), we have to prove that they have the same number of instructions, the same set of dependencies and their sets of $\mathcal{A}(\Theta)$ are equivalent. Then we show that if their dependencies are maintained but with different opcodes, the number of nodes and edges in the labelled Hasse diagram $\mathcal{G}(\Theta)$ is the same, so they have the same *topology.* We call for short, code Θ_E and Θ_F two different schedules.

> **(1)** Two codes (Θ_E and Θ_F) are called *equivalent* if they have the same instructions, the same number of them and they produce the same output.
>
> **(2)** Two codes (Θ_E and Θ_F) have the *same concurrency measure* if they have the same number of instructions, the same set of dependencies and same number of them. It is said that they have the same *topology.*

To show (2) graphically we will show two Hasse diagrams Θ_E and Θ_F with their labelled Hasse diagrams $\mathcal{G}(\Theta_E)$ and $\mathcal{G}(\Theta_F)$ respectively. They are not equivalent because they do not have the exactly the same instructions, but they have the same set of dependencies. Therefore, they do not have the same shape, but the same topology.

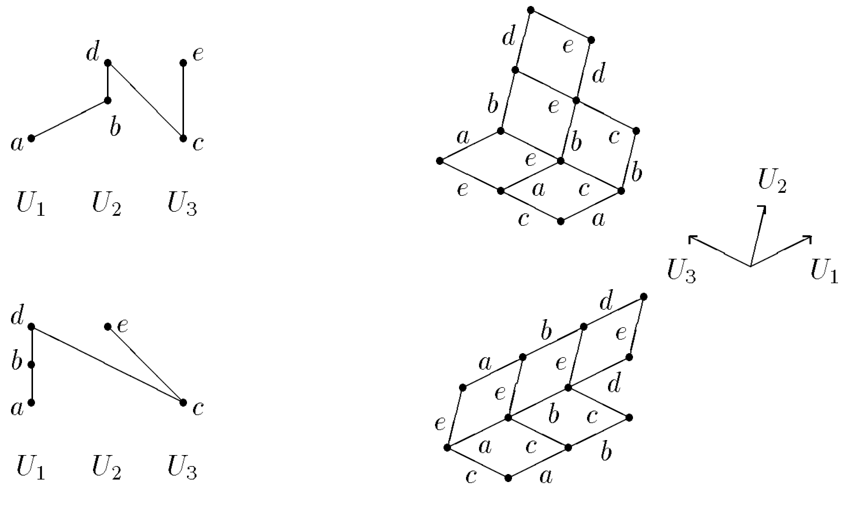

Figure 2.5: Θ_E with $\mathcal{G}(\Theta_E)$ on top, and Θ_F with $\mathcal{G}(\Theta_F)$ on bottom.

From figure 2.5, it can be seen that Θ_E is not the same lattice as Θ_F; they have the same number of instructions and the same number of relations between them, but the units where those events take place do not correspond. From (1), we say that $\Theta_E \neq \Theta_F$. Obviously, $\mathcal{G}(\Theta_E) \neq \mathcal{G}(\Theta_F)$, but again, they both have the same number of nodes and the same number of edges so, as the method to count the concurrency measure is based on the number of edges from each event, the measure from both lattices is the same.

If we have $|\,\mathcal{G}(\Theta_E)\,| = |\,\mathcal{G}(\Theta_F)\,|$ from the same set of relations and the same number of them as in the previous example, we obtain the same measures. The advantage of this is that even if we have a small difference in the opcode, e.g. `muli $14 $8 2` instead of `divi $14 $8 2`, the dependencies are maintained.

2.4 Preliminary conclusions

We have seen that for any correct schedule of a program, the concurrency measure of each instruction stays constant whenever it is placed. This is because the equivalent lattice from the code is the same as all the data dependencies between instructions are consistently respected. This means that every possible schedule of a code has a constant *concurrency mean* which we did not wish for.

The point is that, even for small changes in a schedule (swap two instructions), the there are dramatic changes in the total execution time. This stems from the fact that the simulations are performed in a more complex behaviour where not only the dependencies are involved, but also unit latencies, synchronization overheads, out-of- order execution, cache hit-miss, etc.

This result has guided us to change the particular approach of the concurrency measure, since we want a measure that would be more sensitive to the order of the instructions and would distinguish a good schedule from a bad one. If we could come up with such a measure, then we would be in good position to determine if a schedule is near the optimal set or not.

The labelled Hasse diagram $\mathcal{G}(\Theta)$ is a very compact representation of a program as it captures all the dependencies and the possible schedules on it, without growing exponentially. This particular feature is of interest and will be studied in future work.

Chapter 3

Alternative Measure

The idea of using the concurrency measure in the code to evaluate the schedule was a first attempt, but as seen in the previous chapter, several schedules give consistent measures of $\mu_\Theta(i)$. This is good in the sense that it reflects the points of strong synchronization and the points that are more concurrent, but we want more from a static analysis is to make distinctions between the multiple schedules.

Our need for an effective and inexpensive way to separate schedules that have longer execution times and those that have faster response times is very important, and is even more so if we have to consider techniques and heuristics to increase ILP for the former scheduler.

3.1 Measuring consecutive dependencies

3.1.1 Introduction

The concept behind this way of analysing the code is based on the number of consecutive dependencies in instructions. The reason for the scheduling theory's success is its ability to issue more concurrent instructions and to exploit them conveniently in highly sophisticated architectures with multiples units.

In the past, it has been shown in [7], [12] and [26] that pipeline architectures suffer from pipeline hazards when one instruction needs the result from a previous instruction that has not finished or when a resource is needed by two or more instructions simultaneously. In [17], a schedule is called *greedy* when hardware interlocks have to be inserted if an instruction has to wait for a value that has to be loaded from memory, or when `nop` (no operation) instructions are inserted in the code.

Micronet-based Asynchronous Processor (MAP) Architectures, in common with other pipeline architectures, suffer from functional unit contentions when there are consecutive dependencies. Although it has been shown in [2] that a major advantage of MAP networks over micropipelines and synchronous pipelines is the effective exploitation of temporal and spatial concurrency, consecutive dependencies in the code is detrimental to its performance. The results of scheduling MAP architectures [4][18] reveal that reordering the instructions can be as beneficial as for any other pipeline architecture.

In the rest of this chapter we present an inexpensive way to evaluate different schedules and the preliminary conclusions obtained by it.

3.1.2 The Concept

As it was seen in section 2.4, the results from the concurrency measure algorithm give interesting insights but not very useful for our interests: our bigger concern is to come up with a simple and intuitive way to categorize the schedules from their execution times.

Scheduling theory is based on the idea of detecting and placing concurrent instructions between dependent ones to enhance resource utilisation without producing pipeline stalls; the policy to place the instructions now depends on each author's particular considerations. We will focus our attention where consecutive instructions take place as it is a crucial matter in producing hazard interlocks and lengthening the execution times.

The idea is to penalise an instruction that depends on its immediately previous one. In other words, if a pair of instructions, x and y, are placed consecutively in a code, and y depends on x, we penalise y, otherwise we do not. Let S be a schedule; we define the *weight* of S as the integer and non-negative value ω:

$$\forall\, x, y \in S \;\wedge\; x \lessdot y,\;\; \exists\, \omega > 0 \;\mid\; \omega + dep\,(x, y)$$

$$\text{Where :} \qquad dep\,(x, y) = \begin{cases} 1 & \text{if } x \!-\!\!< y \\ 0 & \text{if } x \parallel y \end{cases}$$

We will refer to $\lessdot$ as the operator for two consecutive instructions in a a schedule. If the instructions are consecutive and the second one depends on the first one, they are said to be *consecutive-dependent* instructions.

If we take again the schedule in figure 2.3, instruction b is positioned immediately after instruction a, so $a \lessdot b$; moreover, as there is a dependency from register `$12` in instruction a, to b ($a \!-\!\!< b$), the algorithm penalises b as it is a consecutive-dependent instruction of a. In that example, only instructions b and d are penalised, but if we apply the same concept to the code in figure 2.4, the instructions that are penalised are b, d and e.

The first impression from both schedules is that the one with less penalties is "statically better" than another that has more because it might avoid more hardware interlocks. In the previous two schedules (from section 2.2 and 2.3.1), the weight for schedule one is 2 ($\omega_1 = 2$), and for schedule two $\omega_2 = 3$.

This principle seems inexpensive, and states the differences between valid schedules as wished, but extensive tests are necessary to allow acceptance of this concept.

3.1.3 Concept example test

The only way to confirm that the measure of a schedule's weight is valid and useful is to exhaust all the possible combinations from it: to simulate one by one all the schedules and to relate the results to a distribution.

However, looking at all the possibles schedules from a code is very expensive as their number grows exponentially with the number of instructions. We will therefore concentrate on very small examples (less than 20 instructions).

The platform under which the simulations are done is based on that in [18], which is a stochastic simulator for MAP that executes source programs, produces execution timings and displays a graphical representation of MAP's behaviour. As it uses a stochastic model, it produces real results bounded between a worst and a best latency time from each unit. An architecture configuration file describes the different units with their latencies and the instructions that are intended to be executed by them.

Table 3.1 displays all the permissible ways to place the five instructions from the example in fig. 2.3 with their actual "weight" measures and the average execution times from their simulations.

Weight ω	Schedule	$T_{avg.}(1)$	Pos(1)	$T_{avg.}(100)$	Pos(100)
0	$a\ c\ b\ e\ d$	90.64 ns	1	4305.77 ns	1
1	$a\ c\ b\ d\ e$	93.56 ns	2	4624.71 ns	2
1	$c\ a\ e\ b\ d$	94.06 ns	3	4650.70 ns	3
1	$c\ a\ b\ e\ d$	97.21 ns	5	4905.36 ns	5
2	$a\ c\ e\ b\ d$	96.41 ns	4	4868.65 ns	4
2	$a\ b\ c\ e\ d$	99.16 ns	6	5148.14 ns	7
2	$c\ a\ b\ d\ e$	99.89 ns	7	5222.71 ns	8
2	$a\ b\ c\ d\ e$	100.05 ns	8	5144.12 ns	6
3	$c\ e\ a\ b\ d$	102.49 ns	9	5528.17 ns	9

Table 3.1: Weight measures and executions times from figure 2.3.

For a matter of simplification, all the experiments were done under the same architecture with the same worst and best latency timings in all the units. The number of units was devised to match all the instructions and avoid possible resource dependencies at execution time (in this particular case there were two adders, two multipliers, two logical units and one memory unit). These considerations were intended to capture the response time only in relation to the variations in the order of the instructions (data hazards).

Each schedule was simulated 20 times under a program to obtain a set of executions of each one and state an average execution time. The average times appear in table 3.1 where $T_{avg}(1)$ represents the average time for one loop-pattern from each schedule, and $T_{avg}(100)$ is the time from a 100 iterations-loop containing the schedules's pattern. The ranking position of each schedule in both experiments is noted as Pos(1) and Pos(100) in accordance with the execution time.

The reason for putting each schedule pattern under loops with different iterations was to see the "stability" of the measure. We wanted to see the order of the execution time from one pattern $T(1)$ and the change when this pattern was repeated multiple times as in $T(100)$. $T(\infty)$ tends to give a better approximation to the real separation from different schedules by spacing the timings in proportion with the number of iterations. Thus, we expect that Pos(100) gives a ranking closer to the real order than Pos(1).

It can be seen that there is a continuous rise in the execution time that corresponds with the increase in the weight ω from the schedules. This is a very small example, even though the schedule with the fewest penalties ($a\ c\ b\ e\ d$) tends to be executed in the fastest way, and schedule $c\ e\ a\ b\ d$, that has the highest weight with 3, has an execution time that is clearly the longest.

There is a good proportional relation seen between the weight ω and the execution times, but we have to remember that as this is a very small example, we have considered an appropriate target configuration to eliminate the chance of resource dependencies (structural hazards). In other words, there are no stalls caused by instructions waiting for a unit to perform them as there are enough units to execute the instructions. Therefore the results in table 3.1 show an execution close to an only-data-dependency behaviour.

Interestingly, schedule $a\ c\ e\ b\ d$ with a weight $\omega = 2$ runs faster than schedule $c\ a\ b\ e\ d$ with $\omega = 1$ in both experiments. This means that there is a possible gap between schedules with consecutive weights. Particular attention shall be given to seeing this in the simulations of bigger examples in the next section.

3.2 A program example

The example in section 3.1.3 shows a good relationship between the weight ω assigned to each schedule and their execution times, but the size of the code and the considerations taken under the configuration target are not realistic.

Our base example is a small loop program written in C language extracted from the *tomcatv* benchmark pictured in figure 3.1(a) with its assembly code in figure 3.1(b). The Assembly code is obtained from the SUIF compiler [36] (in MIPS processor format [16]) and it corresponds to the main

```
main() {

  int i, n = 10;
  int x[10][1];

  for (i = 1; i <= n; i++)
    x[i][1] = ((i - 1) / (n - 1));

}
```

```
L2.main:
        muli    $11,$9,4
        la      $12,$29,0
        addui   $13,$11,4
        addu    $14,$12,$13
        subui   $15,$9,1
        subui   $24,$8,1
        div     $25,$15,$24
        sw      $25,$14,0
        addui   $9,$9,1
        sle     $10,$9,$8
        bt      $10,L2.main
```

Figure 3.1: (a) C Program (b) Assembly code

loop in the program.

The underlying architecture is restricted to one unit of each type: one ALU unit, one logical unit, one memory unit and one branch unit. Again, the worst case, the average case and the best case latency times are the same in all the units. The reason for this is to observe how the program behaves in a more limited configuration where *data*, *structural* and *control* hazards can take place. We use a MAP simulator that models the asynchronous datapath from MAP.

This small example (11 instructions) provides a large enough set of possible schedules to test (3732). For each one, the weight measure ω is computed and each one is simulated 20 times to obtain a minimum, a maximum and an average execution time. This provides a good methodology to see the performance of the measures.

Due to the fact that we are working with an asynchronous target we have to remind some observations from the MAP's behaviour as in [1]. Firstly, we have to penalise consecutive-dependent instructions if they really stall the pipeline. WAR and WAW dependencies, called *false* dependencies, do not avoid concurrent execution and they just imply a limitation to ensure that the source register has the value before writing to it (WAR), and that write-backs (in WAW) are forced to occur in-order. These dependencies do not stall the pipeline as a "true" RAW dependency does, thus we only penalise RAW dependencies. Secondly, the stall produced from a true (consecutive) dependency in branch instructions is considerably longer because it is combined with a control hazard. As the branch instruction decides whether the branch is taken or not all the instructions in the loop have to be completely executed before the control flow can continue. Consequently, we will penalise strongly these dependencies.

3.2.1 Results

The results obtained from each simulation appear in table 3.2. They show the distribution of the execution times from the whole set of schedules in accordance with the penalisations in each one. Again there is a tendency of having better execution times with less penalties. The number of schedules from each measure with their minimum, maximum and mean times are displayed as well. Last column points the percentage of schedules from each measure that are within the 1.5% of the best execution time from the program.

ω	Num.	Min.	Mean	Max.	Best 1.5%
0	177	692.78 ns	698.21 ns	704.87 ns	95.48%
1	513	698.39 ns	723.41 ns	757.22 ns	13.64%
2	826	700.92 ns	741.98 ns	793.18 ns	2.30%
3	1,005	728.31 ns	771.08 ns	825.02 ns	0.00%
4	665	760.04 ns	802.08 ns	855.92 ns	0.00%
5	359	795.18 ns	832.83 ns	886.23 ns	0.00%
6	139	845.90 ns	867.13 ns	914.02 ns	0.00%
7	36	879.43 ns	894.69 ns	905.06 ns	0.00%
8	12	930.75 ns	932.12 ns	934.45 ns	0.00%

Table 3.2: Distribution from the execution times according to the weight.

3.2.2 Observations

The function $t = f(\omega)$ has a domain in $\mathbb{N}^*$ [1] and maps to an $\mathbb{R}^+$ co-domain ($f : \mathbb{N}^* \longmapsto \mathbb{R}^+$). As f is not continuous, it cannot have an inverse function and therefore it is not a one-to-one (bijective) function.

Ideally, the function that we would expect is a *monotonic increasing* function $t = f(\omega)$ where for two weights $x, y \in \omega$ such that $x < y$, we have $f(x) < f(y)$. In other words we would expect that for a schedule with a bigger weight than another, its execution time would be longer than the latter one. Realistically, we do not obtain such a function; we find that for some $x, y \in \omega$, such that $x < y$, $f(x) > f(y)$.

Although the function presents an overlap with the different weights, the advantage of looking at the set with the least measure ensures, with a good probability, of getting a good schedule. In our program example, the set of schedules with a weight $\omega = 0$ ensures that all of them have execution times within the 5% and 95.48% (169 of 177) of them are located in the 1.5% of the best execution time.

[1] $\mathbb{N}^* = 0 \cup \{1, 2, 3, ...\}$

Looking at it in another way, we can see that the variance from the set with the minimum weight is very small. This means that all the possible execution times are close together, and if we add that almost all the set is within the 1.5% of the best execution time, choosing one schedule from this set gives good chances that the schedule performs with a good execution time.

3.3 Location of our analysis

Among the three general ILP architectures described in [28], in the *sequential* architectures, where the superscalar processors are located, the compiler is not expected to communicate to the architecture in any way the parallelism in the code. Conversely, in *dependence* architectures (dataflow processors) and *independence* architectures (VLIW processors), the compiler has to identify the dependencies and independencies between operations respectively.

In our case, MAP processors are considered as a scalar processors, but differ from the classification above in the sense that the compiler aids the asynchronous target to improve ILP even without sending this information to the hardware.

Our analysis, as it has been shown, is based on the appearance in the code of consecutive-dependent instructions. This could be compared with the dataflow processors where the compiler has to expose explicitly the dependent operations in the program. The main difference is that we identify these dependencies (if they are consecutive) and avoid to generate schedules with them. In the case of the dataflow processors each instruction is attached with a list of its successors, and the hardware has to check which instructions have all their operands ready in order to issue them.

In contrast, in VLIW processors, the compiler provides specific information to the hardware about which operations are intended to be executed simultaneously. It must evaluate which operations will be packed to form the final instruction and depending of the target, it decides in which functional units each operation has to be executed. Our static analysis actually looks for independent instructions not in a global way as the independence architectures do, but in a consecutive way. Another big difference is that the final code for our target is not expected to be executed exactly in the same order as the compiler intended as in VLIW architectures, where the hardware virtually makes no run-time decisions in the order of execution; instead instructions will naturally reorder depending on the contention for resources and the length of the path.

In the case of superpipelined (independence) architectures, the compiler again, has to identify all the independent instructions in the code and schedule them knowing the unit's latencies. The complexity of MAP's model certainly forces the compiler to analyse the code and to identify the independent instructions even if it does not communicate explicitly this information to the processor.

3.4 Preliminary conclusions

The preliminary results show that the weight measure based on consecutive dependencies between instructions effectively captures the schedules with lower stalls that seriously affect the execution time. However, due to the large number of possibilities that a code can be scheduled, this concept does not sufficiently restrict the set of good schedules. This implies that further inexpensive criteria must be considered in the near future.

In the next chapter we present the future work around the scheduler for MAP architectures and different optimisations that might be worth considerably for future implementations.

Chapter 4

Thesis Proposal

4.1 A Local Scheduler for MAP

Scheduling code on synchronous RISC processors is simplified by the fact that the latencies of pipeline stages are often considered equal and the instruction's execution times tend to be one cycle [16]. For example, in [12] the proposed scheduler has an $\theta(n^2)$ complexity which considers only 0 or 1 cycle latencies. Palem and Simons [26] present a polynomial order scheduler based on the *rank* algorithm that does not guarantee a feasible schedule if latencies are greater than one. Proebsting and Fisher [27] propose a linear time optimal code scheduler for delayed-load architectures with all instruction latencies of one cycle duration.

The balanced scheduler [17] introduces an algorithm to measure *load* level parallelism that produces beneficial results when the load latency is unknown, with a worst case complexity order of $\theta(n^2 \, \alpha \; n)$ [1]. This is one good attempt to look at the non-uniformity of some systems. However it has been shown in [18] that the balanced scheduler needs different and important considerations for an asynchronous target, such as considering if one instruction's unit has been used previously.

Taking into consideration that the order of the scheduling problem is NP-hard [33] and the non-determinism of MAP asynchrony makes it hard to predict its behaviour, the complexity of a scheduler algorithm could be expected to be high using traditional synchronous approaches. Thus, the importance of an inexpensive way to determine a schedule without consecutive dependencies becomes apparent. An algorithm capable of being less sensitive to the variance in latencies will effectively perform better.

From results in the previous chapter, the concept of consecutive dependencies reveals better schedules than others by avoiding hardware stalls. A heuristic based on this concept is good to discard all the schedules that do not have the minimum weight from the total possible set; however, the num-

[1] α is the inverse of the Ackerman function.

ber of them could be considerably higher, so a second criteria is necessary to reduce the set of optimum static schedules.

4.1.1 Heuristic applied to a local List Scheduler

The results presented in the last chapter have motivated us to incorporate a heuristic based on the "penalise consecutive dependencies" (PCD) concept into the balanced -local- scheduler presented by Ko in [18]. The aim is to see how the heuristic performs in practice with programs of reasonable sizes such as the Livermore Loops [10].

The only difference with the former scheduler is the criteria to decide which instruction from the ready list will be scheduled first. Ko uses a combination of the critical path, number of successors, number of used registers and if the instruction's unit has been previously used. Our heuristic only considers the consecutive (true) dependency along with the unit's dependency. This involves a more simple computation.

The benchmarks were simulated under two architectures with same number of units (one arithmetic unit, one logical unit and one memory unit): one with the same worst, average and best latency timings in all the units, and the other one with a range of latencies for the units. In this way the heuristic is tested in different environments to check its consistency.

It is understood that the heuristic is boosted by the effect of the balanced-scheduler algorithm (and eventually inherits its complexity) but the property that avoids the consecutive dependencies ensures a good performance.

The following charts in figure 4.1 and 4.2 show the execution times from the Livermore Loops without scheduling, with a list scheduler (Traditional list scheduler from Gibbons and Muchnick [12]), with the MAP balanced scheduler, and with a new version with the PCD heuristic. It can be seen that our heuristic performs slightly slower than the other two schedulers, although in some cases it runs equally well.

Interestingly, the results from figure 4.2, where the range of latencies are different, the PCD scheduler has comparable times with the other two schedulers. The reason for this might be that the heuristic is less sensitive to the variance in the latencies.

4.1.2 Improve the local scheduler

Part of the future work involves the optimisation of the local scheduler. From the simulations in the previous section, it is clear that the heuristic is useful and ensures less stalls in the pipeline. Two possible ways to undertake the improvement of it will be considered in the near future. The first one points to determine other suitable criteria to the actual local scheduler, and the second one points to improve the complexity of the actual algorithm.

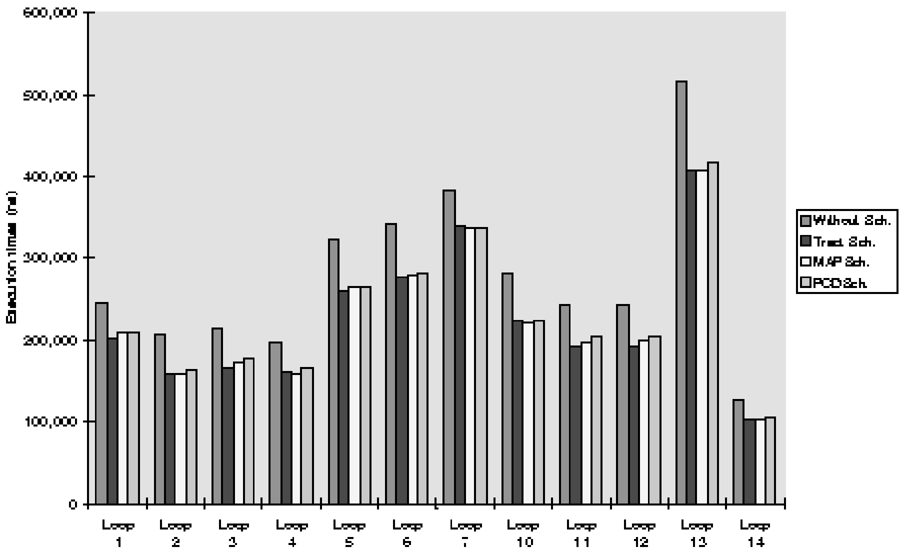

Figure 4.1: Execution times with the same latencies in the units.

Improve the local scheduler with our heuristic

The local scheduler with the consecutive dependency heuristic has shown good times, but we think that there is scope for improvement that has to be exploited. This direction obviously implies that the complexity of the scheduler will stay at the order of $\theta(n^2\ \alpha\ n)$. We will focus our attention at the set of schedules with the least weight ω where the optimal schedule is located.

Improve the complexity of the local scheduler

Looking at the complexity of the balanced scheduler, another possibility to improve it is by implementing an algorithm that also ensures the least weight but with a better complexity order. This means that we will obtain similar results such as the times from the Livermore Loops (in the program example in last section), but in a faster *static* time. Our upper bound is the one from the former balanced scheduler and our expectations are not to exceed it.

We have to evaluate if we want a fast scheduling algorithm and concentrate in further global optimisations (see next section) or if we want a close-to-the-optimal schedule with the the former complexity.

4.1.3 Heuristic upper bound

We have seen that when applying the heuristic to get schedules with the least weight measure we obtain good execution times. However, we do not

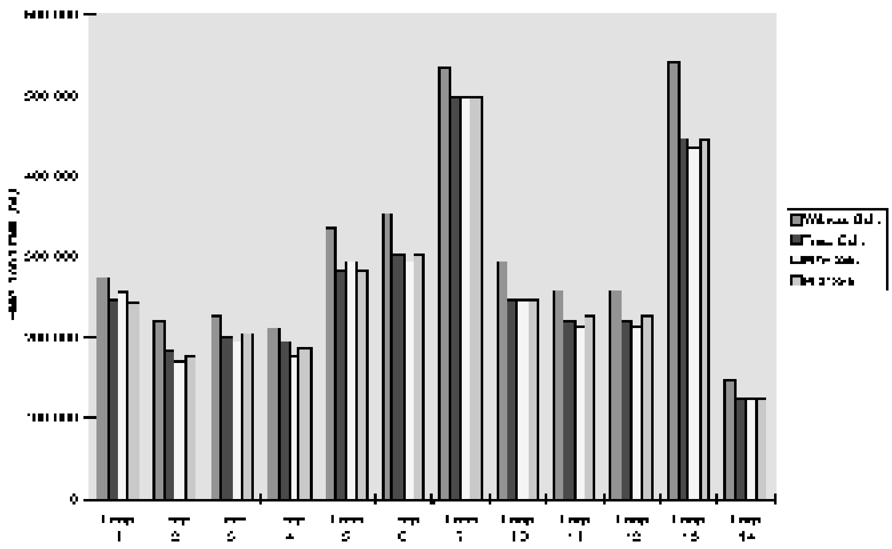

Figure 4.2: Execution times with different latencies in the units.

know the upper bound that this concept could give. This is important to establish the worst case if we want that the heuristic look for *any* schedule with the minimum weight.

In the near future we have to find this upper bound to be able to ensure that any of the schedules in the set with the least measure produce really good execution times.

4.1.4 Consistency from the heuristic

Other important issues in order to observe the consistency of the heuristic are the following:

- Choosing schedules without consecutive true dependencies in the code, ensures no data hazards that are very costly; however, it does not ensure structural hazards produced from the use of a common resource. The big advantage from the Micronet model comes from the fact that different stages are initiated as soon as their micro-operations are ready and they can be stalled if there is a contention from one resource. Most contentions are produced by data hazards, but they can also be produced by structural hazards during the execution. In other words if two instructions are independent does not mean that they do not produce a stall; if those instructions are *related* in some way (e.g. depend on the same unit for example) they can produce a stall that maybe is not as costly as a data hazard.

We will have to investigate the different degrees of stalls from data, structural and control hazards. The importance of independent instructions can be followed by finding *unrelated* instructions. This can be the base for a second criteria on this static analysis.

- Other main issue that has to be investigated is the variance in latencies. The tests so far, have been done with the same ranges in the latencies with a worst -and a best- case latency timings relatively close. The difficulty to predict when the resources are available grows considerably if there is wider variance in their latencies; so if we increment this variance, we will be in position to test how the the weight measure reacts to the unpredictability of the asynchronous behaviour.

  From the Livermore Loops simulations we found that the PCD scheduler performs better in the case where the architecture had different latencies in the units. The results were comparable to the other schedulers. The reason for this is that the heuristic can be less sensitive to the latencies variations in the units. Extending this work by increasing the range of variation will be something to look at shortly.

4.2 Other techniques to increment ILP

4.2.1 Global scheduling

It has been pointed out by the community that there is not enough ILP available in basic blocks [15][34][32][28]. Superscalar and multiple-issue architectures [3][24] require large amounts of fine-grain parallelism to exploit the architectural advantages of their datapaths. Scheduling beyond basic blocks has been an effective solution to improve the amount of ILP as it has seen in [6],[14] and [22].

In our future work we will consider seriously the introduction of global techniques to improve the ILP available and therefore to improve the resource utilisation and to obtain the least of possibles stalls.

Scheduling instructions beyond basic blocks increases the options to the compiler to allocate more independent instructions and avoid stalls; it takes advantage of the branch probabilities. This technique effectively performs better in *control-intensive* programs as there are insufficient instructions to select from the schedule.

4.2.2 Region-based compilation

Region-based compilation is a recent optimisation technique [13] that uses *regions* as the compilation unit instead of functions as traditional compilers do. Functions have been a natural option for the compilation unit as they provide a simple way to partition the process of compiling a program.

However, the major disadvantage of this approach is that the compiler does not have a complete control of the program's size and it has a limited view of the rest of the functions. By *inlining* functions (forming the actual regions) there are more potential opportunities for classical and ILP optimisations.

The region-based compilation effectively exposes hidden code to the compiler, but applying aggressive inlining can lead to register pressure and excessive code expansion.

4.3 Other fields for future research

4.3.1 Instruction compounding

In [25], an instruction compounding mechanism is proposed to avoid register pressure. The idea is to *compound* instructions from two or more (true) *dependent* instructions and the implementation in hardware allows forwarding of intermediate results to other functional units without rewriting the results back to the register bank. Preliminary results look promising.

The compounding mechanism imposes important and different issues to the compiler that must be considered. The consecutive-dependency concept in instructions could be modified to analyse the code statically, to produce schedules suitable to this new architecture.

4.3.2 Multithreading

Multithreading is a well known solution to program multiprocessor architectures that has been used for quite some time. Recently, its usefulness in uniprocessor architectures has become a motivation for researchers to improve processor utilisation [31].

The concept arises from the fact that multiple independent threads of code can be issued to several units at a time. The potential parallelism with this technique is promising, although the complexity of the processor could be considerably higher.

The future work proposed in [25] also looks at the *issue* stage with more than one single unit: it is evident that the natural bottleneck from one issue unit will be relieved by the introduction of several parallel issue units. The multithreading technique will be relevant since we will need massive amounts of parallelism to keep the issue units busy.

Future work to the compiler includes the issues around the implementation of multithreading techniques and the implication of having more than one issue unit.

Bibliography

[1] Arvind, Damal; Rebello, Vinod. *Instruction-level Parallelism in Asynchronous Processor Architectures.* In Proc. 3th. Int. Workshop on Algorithms and Parallel VLSI Architectures. August 1994, pp. 203-215.

[2] Arvind, Damal; Rebello, Vinod. *On the Performance Evaluation of Asynchronous Processor Architectures.* In Proc. 3rd. Int. Workshop on Modelling Analysis and Simulation of Computer and Telecommunication Systems MASCOTS'95, January 1995, pp. 100-105, IEEE Press.

[3] Arvind, Damal; Mullins, Robert. *Micronets: A Model for decentralising Control in Asynchronous Processor Architectures.* In 2nd. Working Conference on Asynchronous Design Methodologies. May 1995, pp. 190-199,IEEE Press.

[4] Arvind, Damal; Rebello, Vinod. *Static Scheduling of instructions on Micronet-based Asynchronous Processors.* In Proc. 2th. Int. Symp. on Advanced Research on Asynchronous Circuits and Systems ASYNC'96 March 1996, pp. 80-91, IEEE Press.

[5] Bareau, Cyrille; Caillaud, Benoît; Jard, Claude; Thoroval, René. *Measuring Concurrency of Regular Distributed Computations.* In Proc. Workshop on Orders, Algorithms and Applications, 1994.

[6] Bernstein, David; Rodeh, Michael. *Global Instruction Scheduling for Superscalar Machines.* ACM SIGPLAN '91. Conf. on Programming Language Design and Implementation. June 1991, pp. 241-255.

[7] Bradlee, David; Eggers, Susan; Henry, Robert. *Integrating Register Alocation and Instruction Scheduling for RISCs.* Transactions on computers. 1991, pp. 122-131.

[8] Chang, Pohua; Chen, William; Mahlke, Scott; Hwu, Wen-mei. *Comparing Static and Dynamic Code Scheduling for Multiple-Instruction- Issue Processors.* Proc. 24th. Int. Symp. on Microarchitectures MICRO 24. 1992, pp. 1-9.

[9] Chang, Pohua; Lavery, Daniel; Mahlke, Scott; Chen, William; Hwu, Wen-mei. *The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors.* IEEE Transactions on Computers. vol. 44, no. 3, March 1995, pp. 353-370.

[10] Feo, John; *An analysis of the computational and parallel complexity of the Livermore Loops.* Parallel Computing Vol. 7, 1988, pp. 163-185.

[11] Fisher, Joseph. *Trace Scheduling: A Technique for Global Microcode Compaction.* IEEE Transactions on Computers. vol. C-30, no. 7, July 1981, pp. 478-490.

[12] Gibbons, Phillip; Muchnick, Steven. *Efficient Instruction Scheduling for a Pipelined Architecture.* ACM SIGPLAN Symp. on Compiler Construction, July 1986, pp. 11-16.

[13] Hank, Richard; Hwu, Wen-mei; Rau, Ramakrishna. *Region-Based Compilation: An Introduction and Motivation.* Proc. 28th. Int. Symp. on Microarchitectures MICRO 28. 1995, pp. 158-168.

[14] Hwu, W.W.; Mahlke, S.A.; Chen, W.Y.; Chang, P.P.; Warter, N.J.; Bringmann, R.A.; Ouellette, R.G.; Hank, R.E.; Kiyohara, T.; Haab, G.E.; Holm, J.G; Lavery, D.M. *The Superblock: An Effective Technique for VLIW and Superscalar compilation.* The Jounal Supercomputing 7, 1/2. May 1993, pp. 229-248.

[15] Jouppi, N.P; Wall, D.W. *Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.* Proc. 3th. Ann. Int. Conf. Architectural Support for Programming Languages and Operating Systems. April 1989, pp. 272-282.

[16] Kane, Gerry; Heinrich, Joe; *MIPS RISC Architecture.* Prentice Hall, 1992.

[17] Kerns, Daniel; Eggers, Susan. *Balanced Scheduling: Instruction Scheduling when Memory Latency is uncertain.* Transactions on computers. 1993, pp. 278-289.

[18] Ko, Michael. *Instruction Scheduling for Micronet-Based Asynchronous Processors.* M.Sc. Project Report. 1995, University of Edinburgh, Department of Computer Science.

[19] Lam, Monica. *Sortware Pipelining: An Effective Scheduling Technique for VLIW Machines.* In Proc. ACM SIGPLAN '91. Conference on Programming Language Design and Implementation. June 1991, pp. 318-327.

[20] Lam, Monica; Wilson, Robert; *Limits of Control Flow on Parallelism.* In Proc. 19th. Int. Symp. on Computer Architecture. May 1992, pp. 43-57.

[21] Lowney, P.G.; Freudenberger, S.M.; Karzes, T.J.; Lichtenstein, W.D.; Nix, R.P.; O'Donell, J.S.; Ruttenberg, J.C. *The Multiflow Trace Scheduling Compiler.* The Journal of Supercomputing 7, 1/2. May 1993, pp. 51-142.

[22] Mahadevan, Uma; Ramakrishnan, Sridhar. *Instruction Scheduling over Regions: A Framework for Scheduling Across Basic Blocks.* pp. 419-434.

[23] Mewissen, Muriel. *Concurrency Measures for Distributed Computations.* M.Sc. Project Report. Department of Computer Science, University of Edinburgh, 1994.

[24] Mullins, Robert. *An Asynchronous Superscalar RISC Architecture.* M.Sc. Project Report. Department of Computer Science, University of Edinburgh, 1995.

[25] Mullins, Robert. *Micro-Networking for Scalable Asynchronous Processors.* Ph.D. Thesis Proposal. Department of Computer Science, University of Edinburgh, 1996.

[26] Palem, Krishna; Simons, Barbara. *Scheduling Time-Critical Instructions on RISC Machines.* In Proc. Transactions on Programming Languages and Systems. September 1993, pp. 632-658.

[27] Proebsting, Todd; Fischer, Charles. *Linear-time, Optimal Code Scheduling for Delayed-Load Architectures.* SIGPLAN 1991, pp. 256-267.

[28] Rau, Ramakrishna; Fisher, Joseph. *Instruction-Level Parallel Processing: History, Overview, and Perspective.* Journal of Supercomputing 7, 1993, pp. 9-50.

[29] Rebello, Vinod. *On the Distribution of Control in Asynchronous Processor Architectures.* Ph.D. Thesis, Department of Computer Science, University of Edinburgh, 1996.

[30] Theobald, Kevin; Gao, Guang; Hendren, Laurie. *On the Limits of Program Parallelism and its Smoothability.* ACAPS Technical Memo 40 McGill University, School of Computer Science.

[31] Tullsen, Dean; Eggers, Susan; Levy, Henry. *Simultaneous Multithreading: Maximizing On-Chip Parallelism.* Proc. 22nd. Int. Symp. on Computer Architecture. 1995.

[32] Uht, Augustus. *Extraction of Massive Instruction-Level Parallelism.* ACM SIGARCH, Computer ARchitecture News. March and June 1993, pp. 5-12.

[33] Ullman, John. *NP-complete scheduling problems.* Journal of Computer and Systems Sciences. October 1975, 384-393.

[34] Wall, David. *Limits of Instruction-Level Parallelism.* In Proc. 4th. Int. Conf. on Architectural Support for Programming Languages and Operating Systems. April 1991, pp. 176-188.

[35] Wall, David. *Speculative Execution and Instruction-Level Parallelism.* Report Western Research Laboratory, 1994.

[36] Wilson, Robert; French, Robert; Wilson, Christopher; Amarasinghe, Saman; Anderson, Jennifer; Tjiang, Steve; Liao, Shih-Wei; Tseng, Chau-Wen; Hall, Mary; Lam, Monica; Hennessy, John. *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers.* Computer Systems Laboratory. Stanford University.