IFP-architectures

In the 1st section we briefly introduce the basic IFP classes, emphasising their common and distinctive characteristics. Here, we also point out that the evolution of pipeline, VLIW and superscalar processors inevitably had to happen in that order. Next, in the 2nd section we shall overview the constraints, which are imposed on the parallel execution of instructions, if a sequential consistency is required. These constraints are the data, control and resource dependences, which are common to all IFP classes and can, therefore, be discussed jointly. Finally, in the 3rd section we are going to investigate how much speed-up potential functional parallelism has, which is to be utilised by all kinds of IFP processors.

1. Overview of IFP processors

The common attribute of IFP processors is their capability to execute more than one instruction at a time. However, as far as their internal operating principle is concerned, pipelined processors, and VLIW-, as well as superscalar processors, follow quite different lines. Whereas *pipelined processors* work like an *assembly line*, in contrast both *VLIW and superscalar processors o*perate basically *in parallel*, making use of a number of concurrently working functional units.

As a first introduction to pipelined processors, let us emphasise their *key characteristics.* They will be highlighted by considering a straightforward pipelined processor which executes integer, RISC-like, register-register instructions. Such a conventional pipelined processor can be characterised by the following features:

- Instruction processing is subdivided into a number of successive subtasks, such as instruction fetch (F), decode, (D), execute (E) and writeback of the result (W).
- Each subtask is performed by an associated pipeline stage. In our example, these are the F, D, E and W stages. Each stage is implemented by an appropriate operation performed usually by a dedicated hardware. Let us consider again our example. Here, the F stage is implemented by an Funit, which fetches instructions from a cache, or from the memory. The D stage is realised by the Dunit which sends only the fitting instructions to the next stage for further execution and performs, if needed, some additional tasks, like checking for pipeline hazards (discussed in the next section). The aim of the E stage is to perform the required operation with the contents of the referenced registers. This is carried out by the Eunit, which, in our case, is a

traditional integer ALU. Finally, the W stage is executed by writing back the result into the specified destination register. This operation does not require additional hardware, it can be performed simply by a write operation into the register file.

- All the pipeline stages operate like an assembly line, with synchron timing of the operation. At the beginning of every pipeline cycle each stage accepts a new input and at the same time it delivers its output to the next stage, except the last one, which writes the result into the specified destination.

It is a significant feature of pipelined execution that *in each cycle a new instruction* can enter the pipeline. Evidently, if an instruction has completed the last phase of its execution, it retires. Thus, as many instructions can be executed in parallel, as there are pipeline stages, that is four in our example.

Pipelining offers higher performance at a cost of increased hardware complexity. Accordingly, pipelined processors were first employed in very expensive super- and vector processors, around 1970. Subsequently, they became the standard implementation of mainframes and, in the 80s, of microprocessors. At present, pipelined execution units (such as fixed- or floating point execution units) are commonly used as building blocks of advanced processors, like VLIW or superscalar ones. For example, the PowerPC 601 has three pipelined execution units, each one for fixed-point, floating-point- and branch execution. The number of required pipeline stages depends, first of all, on the kind of the operation to be performed. Usually, two to ten stages are used. For instance, the pipelined execution units of the PowerPC 601 use two to six stages.

A related new term is **superpipelining**, which means advanced pipelining with a reduced execution cycle at a cost of increased number of pipeline stages. Superpipelined execution units, such a the floating-point units of DEC Alpha 21064 or INTEL Pentium, usually have 8 - 10 stages. However, at this point we stop introducing pipelined processors.

With pipelined processors and subsequently, with multiple execution units, such as separate fixed- and floating-point units, the *main line of development* may be characterised by increasing the parallelism of the execution. As a consequence, instruction issue became a bottleneck, as Flynn had foreseen it a long time ago (Flynn 1966). For instance, assuming that a processor has already two pipelined execution units, each of them capable of starting a new instruction to execute in every pipeline cycle, there would not be much sense, to try to boost performance by further increasing the number of the execution units. Instead, while staying at the same computational model, the only way to get higher performance is, to raise the number of execution units and, at the same time, provide multiple instruction issue. This is exactly what VLIW and superscalar processors do. Thus, their development might be considered as an indispensable step in an endeavour for more performance along the line of von

Neumann processors. Furthermore, the VLIW processors expect dependence free (multi-operation) instructions. This is obviously a less complex situation than coping with dependences on their own in hardware, which is demanded from superscalar processors in the general case. Therefore, it is apparent why VLIW processors preceded the superscalar ones in the main line of evolution.

It is worth noting that the indicated sequence of development coincides with the sequence of increased performance, for obvious reasons.

A common feature of VLIW and superscalar processors is that both are made up of a number of execution units (EU), each of them capable of parallel operation on data fetched from a set of registers. Figure 1. shows the basic structure of conventional VLIW and superscalar processors. Now, multiple EUs execute instructions and write back results into the register file in parallel.

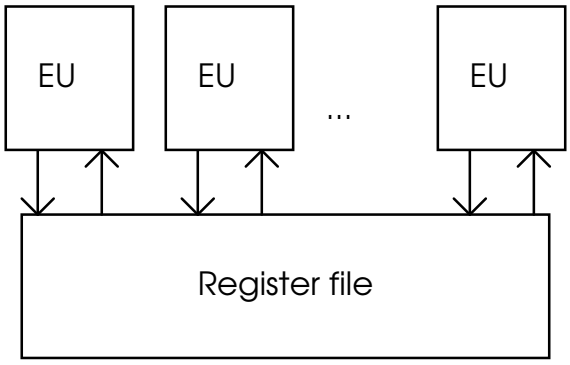


Figure 1. Common basic structure of conventional VLIW and superscalar processors

Here, the label *conventional* refers to the assumption of using RISC-typical register-register operations for data manipulations. Consequently, we take for granted that the results will be written back into the register file. However, we will not be concerned with whether there is a single register file or for any reason, there are multiple register files. For the time being, however, only one point should be stressed: both VLIW and superscalar architectures require highly *multiported register files.* This means that two input ports and one output port are required for each EU. This requirement has to be seen in the light of present day technology, which restricts register port numbers for such processors to less than about 20.

Assuming pipelined execution units, the maximum number of instructions that can be executed in parallel in a VLIW, or superscalar processor, is the sum of the instructions which may be processed in parallel in each of the available units.

Next, let us be specific about the *difference* between VLIW and superscalar processors. These processor classes differ from each other in how instructions

are presented to them. VLIW architectures receive multioperation instructions, that is instructions with multiple fields, each one for controlling one of the available EUs. On the other hand, superscalar processors accept a traditional sequential stream of instructions but are able to issue more than one instruction in each cycle to the EUs.

Here, we made the assumption that instructions will be fetched and decoded in a way similar to how it is done by pipelined processors.
The *VLIW approach* obviously needs *very long instruction words* in order to specify what each EU should do. The necessary length of a VLIW instruction is n-times the length of a traditional RISC instruction word length, if n EUs are presumed. Thus, for VLIW processors usually embodying of the order of ten execution units, the required word lengths are of the order of hundreds of bits. For instance, the TRACE VLIW family can execute 7 to 28 instructions in parallel, depending on the number of modules incorporated (1 - 4), and has word lengths of 256 - 1024 bits (Colwell 1987). Note that our explanation here is somewhat simplified equalling the number of the EUs with the number of the fields of the VLIW instruction.

On the other hand, *superscalar processors* receive a sequential stream of conventional instructions. It is then the task of the Decode and issue unit to issue, in each cycle, *multiple instructions* for the multiple EUs. Current superscalar processors issue usually two (for instance DEC Alpha 21064, Intel Pentium), three (e.g. i960, PowerPC 6O1) four (as the IBM RISC/6000 or PowerPC 604) or seldom more (e.g. 6 in case of the Power2) instructions.

In this preliminany desciption of the VLIW and superscalar processors, we shall not be concerned with more details. It suffices to say that parallel instruction execution has certain *constraints* in common for all IFP architecture classes. These constraints arise from the fact that, although, IFP architectures allow instructions to be executed in parallel, they have to *preserve* the *sequential execution semantics*. It means that IFP architectures have to produce the same computational result as if they were sequentially operating processors. These constraints are crucial for all IFP architecture classes since they restrict the amount of available parallelism, which these architectures aim to utilise. The corresponding issues are discussed in the next section.

2. Constraints on parallel instruction execution

2.1. Introduction

In general, while executing instructions in parallel we have to cope with *two basic types* of constraints. Constraints of the first type are needed to preserve *sequential consistency of the execution.* As we have pointed it out in the previous section, this requirement is needed to guarantee the same results by parallel execution as would be produced during a traditional sequential run. However, the term 'execution' is to be understood in a broader sense as one that covers both, the instruction processing and the exception processing.

Thus, sequential consistency should be preserved for both of these aspects. As a consequence, we have to deal with certain constraints on the parallel execution, which ensure sequential consistency of instruction execution. These constraints are usually referred to as data- and control dependences.

Data dependences are precedence requirements between instructions, needed to keep data, which are referenced by the instructions, consistent with those of a sequential execution. For instance, a precedence requirement is that a data value has to be produced sooner than it can be used.

Conditional instructions, such as conditional branches, also impose restrictions upon the sequence of instructions, which are referred to as **control dependences**. For example, in case of a conditional branch, the choice of which succeeding path will be taken, depends obviously, on the result of the evaluation of the referred condition. Data and control dependences are discussed in sections 2.2. and 2.3., respectively.

On the other hand, in order to guarantee *sequential consistency in exception processing* as well, a further constraint has to be taken into consideration while executing instructions in parallel. This constraint is expressed by the requirement of **precise exceptions**. The related notion is introduced in section 2.5.

The other type of constraints, called *resource constraints,* arises from the limited availability of hardware resources. These constraints, termed **resource dependences**, might also impose restrictions on instruction issue and processing. As an example, consider a processor having two fixed-point units and a single floating-point unit, all pipelined and capable of accepting instructions in each machine cycle. Then, evidently, not more than two fixed-point instructions and a single floating-point instruction can be issued in each cycle.

Summing up, there are three possible *kinds of dependences* (sometimes also called constraints hazards or conflicts), which can prevent particular instructions of the program to be executed in parallel. These are referred to as data, control and resource dependences. Since all IFP architecture classes, such as pipelined, VLIW or superscalar processors, are subject to dependences in the same way, issues related to dependences can be discussed at the same time for all of these classes.

The task, or process, of generating or issuing dependence free code is called, with more or less justification, **scheduling**. The output of scheduling is reorganised code such that for the new code sequence no dependences exist.

There have been *two basic approaches to* IFP scheduling: to perform it either statically and dynamically. *Static scheduling* is accomplished by the compiler. The compiler reorganises a code in order to avoid dependences, but beyond

that, *it also* performs a *code optimisation* to expose as much parallelism to the architecture as possible. Usually, for scheduling, the compiler utilises its knowledge of a portion of the program structure, as well as of resource constraints, to achieve the most favourable code schedule.

The output of the static scheduling is reorganised code. This code has to be constraints free and in addition it is to some extent optimised in respect of the execution model. Note, that VLIW processors always presume a static code scheduling, whereas pipelined and superscalar processors may be scheduled either statically or dynamically, or using both approaches.

In contrast, *dynamic scheduling* is performed by the architecture. For this reason, the processor maintains, typically, two gliding windows. The issue window contains all prefetched instructions which are intended for an issue. On the other hand, those instructions which are being executed, are retained in an execution window. In each cycle, all the instructions in the issue window, are checked for dependences. On one hand, this check involves a data and control dependence check in respect to the instructions being executed, as well as between the instructions themselves, which are in the issue window. On the other hand, a resource depencence check is to be performed to examine the availability of the required resources. As result of these checks, none, one or more independent instructions will be issued to the execution units. For pipelined processors the instruction issue unit attempts to issue one independent instruction, whereas for superscalar processors it tries to issue as many independent instructions as demanded by the specified issue rate. Note that dynamic scheduling *merely preserves sequential consistency*, it does not include code optimisation. Therefore, it is a common practice, also in the case of dynamically scheduled processors, to use additionally enhanced compilers which perform code scheduling to boost performance.

2.2. Data dependences

2.2.1 Overview of data dependences

Let us consider two instructions i_k and i_l of the same program, where i_k precedes i_l. If these two instructions refer to the same data item and their correct execution presumes that the execution of instruction i_k precedes that of i_l, then, i_k and i_l are said to be **data dependent.** In order to retain sequential consistency, data dependence requests instructions to be executed in the same sequence as specified.

Data dependence may occur either in "loop-free code" between subsequent instructions, or in a loop between instructions belonging to subsequent iterations of a loop. Here, by "straight line code" we understand any code sequence, even instructions of a loop body, which does not contain instructions

from successive loop iterations. As we shall see in the next section, "loop-free code" may contain three different types of dependences, called RAW, WAR and WAW dependences. Dependences found between instructions belonging to different loop iterations are termed recurrences (see section 2.2.3.). Furthermore, in any one case of a data dependence, either register data or memory data may be involved.

2.2.2. RAW, WAR and WAW dependences

We shall now discuss the abovementioned types of data dependences. However, to simplify the discussion, we shall be concerned *only* with *register data.* Evidently, data dependences for memory data are understood in the same way.

RAW-dependence (Read after Write dependence or flow dependence)

Let us consider the following two assembly language instructions:

i1: load r1, a;

i2: add r2, r1, r1;

Here, the instruction i2: uses r1 as a source. As a consequence, i2: cannot be correctly executed until r1 have been loaded by i1:. Therefore, i2: is said to be *RAW-dependent* on i1:.

In general, an instruction i_l is said to be **RAW-dependent** on another instruction i_k, if i_l references as a source, a data value which will be produced by i_k. RAW-dependences are also called *flow dependences.* They are, at the same time, *true dependences*, since they cannot be abandoned. This is in sharp contrast to WAR- and WAW-dependences which can be circumvented by proper methods.

WAR-dependence (Write After Read dependence, anti-dependence)

As an example, let us discuss the execution of the following instructions:

i1: mul r1, r2, r3;

i2: add r2, r4, r5;.

In this case i2: writes the content of r2 while i1: uses r2 as a source. If, for any reason, i2: were to be executed ahead of i1: then the original content of r2 would be rewritten earlier than read by the instruction r1, which would evidently lead to an erroneous result.

Accordingly, a **WAR-dependence** exists between two instructions i_k and i_l, if i_l specifies, as destination, one of the source addresses of i_k (which can be either a register or a memory address). Note that WAR-dependences are also termed as *anti-dependences.* They are, at the same time, *false dependences* (along with WAW dependences) since they can be eliminated through register renaming. This means that in case of a false dependence, the destination register of the affected instruction i_l should be renamed to a not yet used register name. For instance, the RAW-dependence in the above example can be eliminated by renaming r2 in i2: to, say, r6:

```
i1:  mul r1, r2, r3;

i2:  add r6, r4, r5;.
```

False dependences, and first of all anti-dependences, may seriously impede the performance of IFP processors. Therefore, it is common practice to eliminate them either statically by the compiler, or dynamically by the IFP processor. There are various techniques for eliminating false dependencies by dynamic register renaming used more and more in superscalar processors.

WAW-dependence (Write After Write dependence, output dependence)

Two instructions are said to be **WAW-dependent** (or also output dependent) if both write the same destination, as, for instance, in the following example:

```
i1:  mul r1, r2, r3;
i2:  add r1, r4, r5;.
```

This kind of dependence is obviously also a *false dependence* and can be eliminated in the same way as WAR-dependences through register renaming.

2.2.3. Recurrences

As we have already seen in the previous section, instructions of the loop body may have RAW, WAR or WAW dependence among themselves. But beyond this kind of data dependence, instructions belonging to a particular iteration, may additionally be dependent on instructions belonging to previous loop iterations. We term this kind of data dependence as *recurrence.* Note that the designations *inter-iteration data dependences,* or *loop carried dependences,* are also used.

From an other point of view, recurrences are references in loop bodies to values that have been computed in previous iterations. An example for a recurrence is shown by the sample high level language loop below:

do I = 2, n

X(I) = A ### X(I-1) + B

enddo

Here, in order to compute the value of X(I), the value of X(I-1), that is the result of the previous iteration, is required.

In general, a **recurrence** is said to be **of k-th order** if the associated computation needs results which have been computed in the preceding k iterations, such as:

$$X(I) = f(A(I), X(I-1), X(I-2), \ldots, X(I-k))$$

The most common form of recurrences are the *first order linear arithmetic recurrences*, like the one shown in the loop above.

A **first order linear arithmetic recurrenc**e has the general form:

$$X(I) = A(I) * X(I - 1) + B(I)$$

Recurrences may be a major concern for some types of architectures, also for some methods used by compilers. This is always the case when the result of a previous iteration is required earlier that its computation has been completed. Thus, recurrences are obstacles in polycyclic architectures, which are a variant of VLIW processors. They can also inhibit vectorising loops for vector processors . A further example of a cause for concern is the modulo scheduling technique of software pipelining.

2.2.4. Data dependence graphs

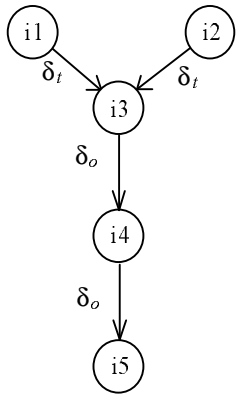
In this section we show how data dependences may be represented by *directed acyclic graphs*. This method of representation is used most frequently by IFP compilers to describe data dependences. Data dependence graphs also serve, as the base for code scheduling techniques.

Since data dependence between two instructions may be regarded as a *precedence requirement*, it can be represented by a *directed arc* connecting two nodes which are associated with the instructions involved. In the example, the instruction i2: is dependent on i1:, therefore i1: must precede 12:.

Let us now consider a straight line code sequence:

```
i1 : loadr1, a;
i2: loadr2, b;
i3: add r3, r1, r2;
i4: mul r1, r2, r4;
i5: div  r1, r2, r4;
```

Evidently, all the dependences among the instructions can be represented by a directed graph, as indicated in Figure 2.



Instruction interpretation: $r3 \leftarrow (r1) + (r2)$ etc.

Figure 2. A straight line assembly code sequence and its corresponding DDG

The graph has a node for each instruction considered. *Directed arcs* indicate *dependences* (precedence relations) between instruction pairs. In additional, as a clarification, we labelled each arc ###*t*, ###*a* or ###*o* for denoting true (RAW), anti (WAR) or output (WAW) dependence, respectively.

If we restrict ourselves to s*traight line code,* the directed graph representing data dependences is *acyclic.* Consequently, data dependences among instructions of a straight line code can be represented by a **directed acyclic graph** (**DAG**). This graph is called by many, diferent names in various publications, emphasising either the form or the semantics of the representation, or both, like *DAG*, *dependence DAG*, *code DAG*, *dependence graph,* or *data dependence graph.* Although, there are some minor differences among the representations, basically, they all indicate the same: the dependence, or precedence relations among the instructions or operations concerned. Therefore, from now on, we shall call this graph data dependence graph (DDG).

A DDG differs in two aspects from the dataflow graphs (DFG). First, a DDG does not comprise control structures, in contrast to the DFGs. Furthermore, a

DDG shows all possible dependence relations among operations, or instructions (that is all RAW, WAR and WAW dependences), while a DFG indicates only RAW dependences, because of the single assignment character of the dataflow model of computation.

Usually, both, traditional optimising compilers and IFP compilers construct a DDG for each basic block. The principle of the construction is straightforward: examine all possible instruction pairs for dependences, and if there is a dependence between the two instructions considered, construct the graph accordingly.

When the basic block embodies 'n' instructions, $(n^2 - n)/2$ such examinations are required. Thus, the construction of a DDG is the most time consuming task of schedulers (Warren l991). For details of the DDG construction see for instance Landskov ### al. 1980.

2.3 Control dependences

2.3.1 Notion and impact of control dependences

Conditional branches are also a major cause to impede the parallel execution of instructions. For instance, in the code sequence

```
sub r1, r2, r3;
jz   zproc;
mul r4, r1, r1;
   :
   :
zproc: load r1, x:
   :
```

the actual path of execution depends on the outcome of the subtraction. This means that the instructions following a conditional branch, are dependent on it. In a similar way, all conditional control instructions, such as conditional branches, calls, skips etc., cause dependences on the subsequent instructions which are referred to as **control dependences.**

How *frequent* are conditional branches? This is a point which deserves special attention in order to see how seriously conditional branches impede performance.

Published data of evaluated execution traces (Lee, Smith 1984, Stephens & al. 1991, Yeh, Patt 1992) show that general purpose programs on one side, and

scientific/technical programs, on the other, behave quite differently as far as the frequency of branches, or conditional branches, are concerned. Here, we are using the term *general purpose program* to stand for compilers, operating systems or non-numeric application programs. The data statistic indicate that *general purpose code* have a very high percentage of *branches,* which can amount to *20 - 30 %*. By contrast, in *scientific floating point programs* we can expect much fewer branches, the probable frequency is as low *as 5 - 10 %*. The ratio of unconditional branches and the total number of branches seems to be quite stable in different programs, remaining within the range of 75 - 85 %. As a consequence, the expected frequency of unconditional branches in general purpose code is about 20 %, whereas in scientific programs it is merely 5 -10 %. This means that as far as IFP processors are concerned, control dependences constitute a more serious hurdle in speeding up a general purpose code than a scientific one. A further consequence is that in considering conditional branches, it makes more sense to distinguish between general purpose and scientific code than discussing average values.

For general purpose (integer) and scientific (floating point) programs the average ratio of branches varies considerably. Nevertheless general purpose programs clearly include more branches than scientific ones, as we have pointed out previously. The simple mean values of the average branch distances in general purpose and scientific programs, are 4.6 and 9.2 respectively. The related values of branch ratios are 21,9% and 10.9%. Furthermore, published data show that the ratio of unconditional branches and total branches varies between about 62 % - 99 %, with mean values of 83 % for general purpose code and 85 % for scientific code.

All in all, in running a general purpose code, such as operating systems or non-numerical applications, we can expect every 3th -5th instruction, on the average, to be a conditional branch. By contrast, while executing scientific programs, every 10th - 20th instruction on the average, is most likely to be a conditional branch.

Frequent conditional branches, however, impose a *heavy performance constraint* on IFP architectures. This is due to the fact that IFP processors can boost performance mainly by executing more and more instructions in parallel. Increased parallel execution, on the other hand, forces multiple instruction issue, in order to feed multiple execution units properly. However, the more instructions will be issued, on the average, in each cycle, the higher the probability is to encounter a conditional branch, that is a control dependence, during an instruction issue. *Control dependences* are, therefore, a *major obstacle* in an endeavour for increasing the performance of IFP architectures. Consequently, one of the focal points of recent research in this class of architectures is how to cope with control dependences.

2.3.2. Control dependence graph

Like data dependences, control dependences can also be represented by directed graphs.
Although, there are several slightly differing interpretations and designations, basically, in order to represent control dependences, nodes with two successors are introduced (see Figure 3.). The outgoing arcs now represent the true (T) and the false (F) path, and are usually labelled accordingly, with T and F.

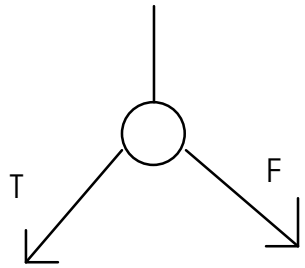


Figure 3.: Representation of a control dependence

Nodes with only one outgoing arc, are interpreted either as operational instructions or as a sequence of conditional branch free operating instructions (straight-line code). The usual designation of directed graphs expressing control dependences is **Control Dependence Graph (CDG).** Alternative terms are (often with minor differences in their interpretation) *Program Flow Graph*, *Program Graph* and *Program Control-Flow Graph.* An example of a CDG is shown in Figure 4.

```
i0:    r1 = op1;
i1:    r2 = op2;
i2:    r3 = op3;
i3:    if (r2 > r1)
i4:       { if (r3 > r1)
i5:            r4 = r3;
i6:         else r4 = r1}
i7:      else r4 = r2;
i8:           r5 = r4 * r4
```

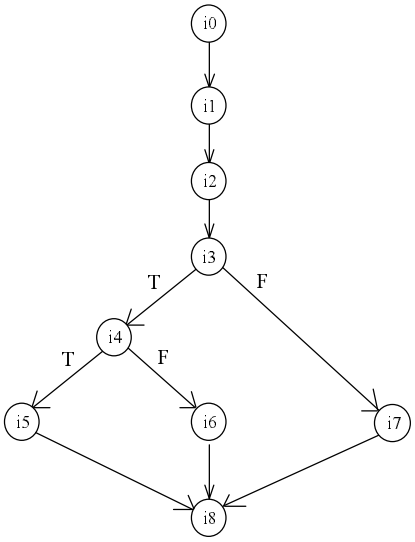


Figure 4.: An example for a CDG

2.4. Resource dependences

An instruction is **resource dependent** on any previously issued instruction if it requires a hardware resource, in most cases an execution unit, which is in use by a previously issued instruction. If, for instance, there is only a single multiplication unit available, then in the code sequence

```
mul r1, r2, r3:
mul r4, r2, r5;
```

the second multiplication instruction is resource dependent on the first and can not be executed in parallel with the first one. By providing a large enough number of high performance execution units, resource dependences usually do not reduce seriously the performance of up-to-date IFP architectures. Note that in the next table auxiliary units, such as load/store units etc., are not taken into consideration.

Processor type	**Year of appearance**	**Number of execution units**
Motorola 88110	1991	8
DEC Alpha 21064	1992	2
Intel Pentium	1993	3
PowerPC 601	1993	3

PowerPC 604	1994	5

Number of execution units provided by recent superscalar processors

2.5. Precise exceptions

As emphasized previously, IFP processors are expected to behave like their sequential peers as for as program execution is concerned. However, sequential consistency might also be required related to the handling of exceptions. This requirement, which is **called precise exception** handling, will be understood as follows. If a particular instruction causes an exception during its execution, and as a consequence, some time later the execution will be interrupted in order to process the exception, the state of an IFP processor by its interruption, must be the same as a sequential processor would have. Practically, this requirement can be satisfied by an IFP processor by delaying the acceptance of an exception until all instructions preceding the faulty one have been executed, and by preventing all subsequent ones from execution.

3. The speed-up potential of functional parallelism

In this section, we discuss first, how much parallelism is available in programs. This indicates the limits for IFP-compilers and processors, for speeding-up computation. Finally, we overview the methods of how functional parallelism can be extracted from conventional sequential programs.

3.1 How much speed-up potential has functional parallelism ?

Since IFP processors rely on instruction level functional parallelism in attempting to speed-up computation, it is a crucial point how much potential speed-up is there. As a consequence, a considerable number of papers have been published dealing with the amount of available parallelism, mostly at the beginning of the 70s and in the previous years.

Perhaps, the best way to overview the disturbingly large amount of published data is to investigate the potential speed-up in connection with the constraints which prevent parallel instruction execution. Remember at this point that parallel instruction execution may be prevented by data-, control- and resource dependences. In considering the potential speed-up, it is a common practice to *neglect resource constraints*, assuming that there are sufficient hardware resources. Furthermore, it is convenient to assume that, through appropriate

techniques (register renaming), false dependences are also eliminated. Thus, for the moment, we have to investigate only the *impact of the true data as well as of the control dependences.* At the end of this section we shall evaluate how false dependences impede speed-up potential.

How we can summarises a number of the available results concerning the potential speed-up of programs when both, true data and control dependences restrict parallel instruction execution? The published data are divided into two groups: general purpose and scientific programs. Compilers, operating systems and non numeric application programs, are regarded as **general purpose programs** whereas **scientific programs** are dedicated ones containing massive floating point computations usually on vectors, or matrices.

It is fact that *general purpose programs* which embody predominantly irregular parallelism, have only a *very low amount* of *available parallelism.* Most investigations have revealed an astonishingly low potential speed-up in the range of *1.2 - 3.0* with an *average* value of *about 2* (Tjaden, Flynn 1970, Kuck & al. 1972, Riseman, Foster 1972, Jouppy 1989, Lam, Wilson 1992). The corresponding speed-up values for scientific programs are a little higher being in the range of 1.2 -17 with an average of about 2 - 4.

These data are certainly disappointing and would discourage any motivation to develop processors relying on potential speed-up values like this. There are, however, *two* encouraging *approaches.* The first approach attempts to utilise regular parallelism; more precisely, potential speed-up embodied in loops. The second one is based on the proper handling of control dependences.

It has been shown that *loops* are a promising source of parallelism which can be exploited by proper techniques, such as software pipelining.

The second promising approach concerns the appropriate *handling of control dependences.* Riseman and Foster have recognised as early as 1972 that control dependences restrict parallel execution very much. They have investigated the potential speed-up assuming for each conditional statement a *perfect oracle*, as well as *false dependence removal.*

The *control dependences* are the *real obstacle* in utilising instruction level parallelism. If we were able to make perfect oracles, and could assume the availability of sufficient resources of hardware, we could expect speed-up values of the order of 10 to 100, in some cases even higher. In practice, of course, no oracles can be made, and these are no unlimited resources available. An apparent substitution for an oracle would be to pursue all possible execution paths by multiple, multiway branches. This would require for each conditional branch to follow both (or all) possible paths and at the end of the computation discard all paths except one which turned out to be the all-

true one. This approach would require for a chain of 'n' conditional statements, the execution of computations along 2^n paths. Obviously, pursuing an exponentially increasing number of paths would also be an unrealistic approach. However, based on this approach, a number of applicable approximations have been developed, either by making a guess for the most probable path (called *branch prediction*), or pursuing both possible execution paths (termed as *multiway branching*) for a single conditional branch, or for a restricted number of subsequent branches.

How close can *real systems* come to the upper limits of speed-up? This is evidently an other crucial point.

On this topic, Wall published remarkable results and Wall's investigations show that ambitious real systems can expect to achieve speed-up figures of *about 4 for general purpose programs* and *around 10 - 20 for scientific programs.* An ambitious system in his sense predicts branches and jumps, has 256 CPU and 256 FPU registers, eliminates false dependences through register renaming, performs a perfect memory disambiguation and maintains a gliding instruction window of 64 items. For the time being, both his hypotheses concerning an ambitious machine as well as the obtainable speed-up figures seem to demarcate the real limits.

Finally, we want to assess, how much gain might be expected as a result of *eliminating false dependences.* Wall has investigated the achievable speed-up figures assuming either no elimination, imperfect elimination using 32, 64 or 256 renaming registers, or a perfect elimination assuming an unlimited number of available registers. His results show that *false dependence elimination* has, for the majority of benchmarks, a *considerable impact* on increasing speed-up. Therefore, an advanced IFP execution is expected to remove false dependences in one or another form . However, in order to achieve higher values of gain, at least 64 renaming registers appear to be required.

Finally, it should be emphasised again that the crucial point in the development of IFP processors seems to be how well they can handle conditional statements. Novel developments apparently ensure this.

3.2. Parallelism extraction

As we know, instruction-level functional parallelism is implicit. Therefore, to utilise it, it must be extracted first. To decide which instructions can be executed in parallel and which cannot, is referred to as *parallelism extraction.* This is a task which is performed during static or dynamic instruction scheduling (see section 2.). Remember, that *dynamic scheduling* comprises only a dependence check as well as the issue of independent instructions. Here, we are not concerned about the particular way, of how instructions are issued. On the other hand, *static scheduling* includes both dependence check

as well as code optimisation. Therefore, we can expect static scheduling to be more effective than dynamic scheduling, as far as parallelism extraction is concerned.

However, static scheduling and, consequently, static parallelism extraction may be performed at three different levels; at basic block-, loop- and global level. The associated scheduling and the embodied parallelism extraction techniques are termed accordingly, as *basic block-* (or local), *loop-* and *global techniques.* These techniques have an increasing performance in the listed order.

The easiest way to extract parallelism is to do it at the so called basic block level. In this case, parallelism will be extracted, and a code optimisation accomplished separately for each basic block, one after another.

Basic block is a concept arising from compiler technology. It denotes a straight-line code sequence that can only be entered at its beginning and left at its end. Usually, conditional branches and target addresses of branches break programs into basic blocks.

Usually, basic blocks for general purpose programs, contain, on the average, no more than 3-8 instructions (DeRosa, Levy 1987). For several C benchmarks on a RISC II architecture these authors measured basic block lengths of 3.5-6.3, with a total average of 4.9. As another example, we quote basic block lengths measured by Lam and Wilson (1992) for MIPS 3000 programs (see Table 5.3.5). They reported somewhat higher basic block length figures than quoted above. Furthermore, their results show that scientific programs contain apparently longer basic blocks, on average 31.6, than general purpose programs, with an average of 7.5. This is in accordance with data concerning the average branch distance.

```
i1:   calc:   add r3,r1,r2;
i2:             sub r4,r1,r2;
i3:             mul r5,r3,r4;
i4:             mul r7,r6,r6;
i5:             sub r8,r7,r5;
i6:             jn   negproc;
```

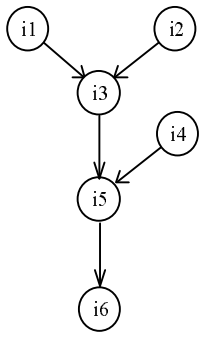


Figure 5: An assembly level basic block and its graphical representation by a DDG

Obviously, for RISC-style as well as scientific programs, the basic block size is somewhat longer than for CISC-programs or for general purpose programs. Note that basic block scheduling can not cope with the restrictions arising from control dependences. Therefore, the achievable speed-up values for basic block scheduling are limited to those low figures, which correspond to this case, that is, when both true data dependences and control dependences restrict parallel execution. Practically, this restricts the *expectable speed-up* potential of basic block scheduling to values of *about 2* for general purpose and to 2 - 4 for scientific programs.

The next possible level of parallelism extraction is *loop level extraction.* Here, instructions belonging to consecutive iterations of a loop can usually be executed in parallel resulting in considerable speed-up. However, recurrences may, to a certain extent, impede achievable speed-up.

It is widely accepted that loops are a significant source of parallelism, first of all in scientific programs. Therefore, it is apparent, that highly parallel ILP architectures, such as VLIW processors, should extract parallelism at least at the loop level. Accordingly, a large number of techniques have been developed to perform scheduling at this level.

The most effective way to extract parallelism is to do it at the highest possible level, using *global scheduling techniques.* Here, parallelism is searched for and extracted beyond basic blocks and simple loops, in such constructs as compound program sections involving loops as well as conditional control constructs. Evidently, in case of a chunk of code larger than a basic block, there are much more possibilities for parallelisation. However, the price to pay is increased complexity.

To sum it up, parallelism may be extracted either dynamically by the processor, or statically by the compiler. In the latter case, parallelism can be extracted at three levels: at basic block, loop and global level. These techniques have increased potential for parallelism extraction and speeding-up computation in the same sequence as listed, but at the price of increased

complexity. Obviously, the more parallel an ILP architectures is, the more powerful parallelism extraction techniques are expected to be used. Therefore, highly parallel VLIW or superscalar processors require the use of loop- or global scheduling.

Tartalom

1. Overview of IFP processors....2
2. Constraints on parallel instruction execution....5
    2.1. Introduction....5
    2.2. Data dependences....6
        2.2.1 Overview of data dependences....6
        2.2.2. RAW, WAR and WAW dependences....7
        2.2.3. Recurrences....8
        2.2.4. Data dependence graphs....9
    2.3 Control dependences....11
        2.3.1 Notion and impact of control dependences....11
        2.3.2. Control dependence graph....12
    2.4. Resource dependences....13
    2.5. Precise exceptions....14
3. The speed-up potential of functional parallelism....14
    3.1 How much speed-up potential has functional parallelism ?....14
    3.2. Parallelism extraction....16