Instruction Level Parallelism: scalar, superscalar, and multithreaded architectures

These notes integrate the Course Notes, Part 1, Sect. 19.4, about the CPU architectures based upon the Instruction Level Parallelism (ILP) concept. After reviewing Sect. 19.4 with some examples of program evaluation and optimization, we will extend the basic pipelined architecture (scalar architecture, in which each stream element contains just one instruction) to superscalar architectures (pipelined execution in which each stream element contains more than one instruction). Then, we introduce the concept of multithreading, a technique for exploiting ILP in order to achieve parallelism between concurrent threads too on a time slot basis, in particular we will study Simultaneous Multithreading (SMT). Finally we will discuss the relationship between multicore and multithreading and their integration in a multiprocessor architecture.

The treatment of these topics is not formalized in these notes, since a complete formal study of IPL is out of scope of the course. The reader, interested to a more formal and deep treatment, can consult the teaching material of “Architettura degli Elaboratori” by the author.

Contents

1. Scalar pipelined CPU architectures .................................................................2
   1.1 Performance evaluation and optimizations .....................................................2
   1.2 Data dependencies .......................................................................................4
   1.3 Impact of parallel-pipelined EU on performance and optimizations ...............5
   1.4 In-order vs out-of-order behavior: static vs dynamic optimizations .................7

2. Superscalar CPU architectures ........................................................................10
   2.1 VLIW superscalar architecture .....................................................................10
   2.2 Examples ......................................................................................................11
   2.3 Parallel subsystems in n-way superscalar architectures ..................................14

3. Multithreading ..................................................................................................16
   3.1 A simple example of multiprocessor vs multithreading ..................................16
   3.2 Scalability of multithreading vs single thread ................................................19
   3.3 Multithreaded architectures ..........................................................................20
   3.4 Simultaneous Multithreading ........................................................................21
      3.4.1 Instruction composition and scheduling ..................................................22
      3.4.2 From multithreading to multicore ...........................................................23
      3.4.3 Two examples of multithreading exploitation in multiprocessor architectures ....24
1. Scalar pipelined CPU architectures

In Course Notes, Part 1, Sect. 19, we have introduced:

- 19.4.1: the concept of pipelined execution of an instruction stream, the principles of pipelined CPU architectures, the parallelism forms that actually take part in the definition of such an architecture, and the typical performance degradations (branch degradation and logical dependence degradation);
- 19.4.2: the parallel-pipelined implementation of the Execution Unit.

In this section we review such issues in more depth, and introduce some compile time optimizations.

Before starting, let us describe an important architectural issue: synchronization mechanisms for machine state consistency.

Two main synchronization problems exist for ensuring consistency of machine state: 1) consistent usage of general register copies (main copy RG[64] in EU, secondary copy RG1[64] in IU, assuming 64 general register as in D-RISC), and 2) consistent usage of program counter copies (main copy IC in IU, secondary copy IC1 in IM, or more precisely in MMU_I). The solutions are the following:

1) an array of non-negative semaphore registers SEM[64], initialized to zero, is associated to the array of general registers copy RG1[64] in IU. When IU processes an arithmetic or Load instruction with destination register RG[i], SEM[i] is incremented. When EU modifies RG[i], the same value is also sent to IU, where it is written into RG1[i] and SEM[i] is decremented. When IU wants to read RG[i] (e.g. to compute an address in Load/Store instructions, to test a predicate in conditional branch instructions), it waits until SEM[i] = 0. Of course, IU has a nondeterministic behavior: each communication from EU is listened during any clock cycle in parallel with the “normal” behavior (analogously to interrupts);

2) when a branch is actually executed, IU modifies IC and the same value is also sent to IM (MMU_I). As soon as MMU_I, having a nondeterministic behavior, receives a new value of IC, this value is written into IC1, and a new instruction stream is initialized starting from this address. When IU receives an instruction from IM, the instruction is accepted only if it belongs to the valid instruction stream, otherwise it is discarded. In order to distinguish the instruction validity, a unique identifier is associated to each stream element, whose value has been communicated by IU when a branch target address is sent to IM. IU accepts the received instruction if the associated identifier is equal to the last identifier IU sent to IM. To implement such an identifier, a simple solution is that it coincides with IC value itself.

1.1 Performance evaluation and optimizations

Let us consider the following program for integer array addition:

\[
\begin{align*}
\text{int } A[N], B[N], C[N]; \\
\forall i = 0..N-1: C[i] = A[i] + B[i]
\end{align*}
\]

compiled in D-RISC as follows:

```
1. L : LOAD RA, Ri, Ra
2. LOAD RB, Ri, Rb
3. ADD Ra, Rb, Ra
4. STORE RC, Ri, Rc
5. INCR Ri
6. IF < Ri, RN, L
7. END
```

The execution simulation of a generic loop iteration is shown in the following figure:
Notice that all the arithmetic operations have unit-latency in the execution phase (EU). In the next section the performance will be evaluated for examples using a parallel-pipelined EU.

There are degradations due to logical dependencies between instructions (instruction 3 induces a logical dependence of distance 1 on instruction 4, instruction 5 induces a logical dependence of distance 1 on instruction 6) and due to the branch caused by instruction 6 (with probability $\sim 1$ for large $N$). The effective service time is (6 instructions are processed in a time interval of size $10t$):

$$T = \frac{10t}{6}$$

Thus the relative efficiency of CPU:

$$\varepsilon = \frac{t}{T} = \frac{6}{10}$$

This program can be optimized at compile time. Any optimization is based on a proper reordering of instructions (“code movements”), in such a way that the performance degradation causes are alleviated (possibly eliminated), while preserving the program semantics.

Logical dependence degradations can be reduced by trying to increase the distance between the instruction inducing the logical dependence (for example, 5) and the instruction affected by the logical dependence (for example, 6).

In our example, INCR can be anticipated in order to increase the distance between INCR and IF:

1. L : LOAD RA, Ri, Ra
2. LOAD RB, Ri, Rb
3. INCR Ri
4. ADD Ra, Rb, Rc
5. STORE RC, Ri, Rc
6. IF < Ri, RN, L
7. END

on condition that the base address of C is initialized to the real value minus one.

In so doing, the logical dependence of INCR on IF has been eliminated, since now the distance is such that it is resolved when IF is fetched in IU. The logical dependence of ADD on STORE remains. Notice that STORE depends on both INCR and ADD, however, in order to have both operands ready, STORE has to wait that the last (ADD) is executed. It can be verified (see the execution simulation) that:

$$T = \frac{9}{6} t \quad \varepsilon = \frac{6}{9}$$
Branch degradations can be alleviated in several ways, for example by loop unfolding or in-line procedure expansion (in order to eliminate some branch instructions), or by code movement. In the last case, an interesting technique, called delayed branch, consists in instruction reordering in such a way that the “bubble” caused by the branch is actually filled with useful work (of course without altering the program semantics). An instruction annotation (“delayed_branch”) must be provided at the assembler level machine, according to which (at the firmware level) the IU interpreter does not discard the instruction received after the branch instruction (as it happens when the delayed branch technique is not applied).

In our example, the STORE instruction can be moved after the IF instruction, so that STORE is usefully fetched during the time slot after IF (thus, no bubble actually occurs, without altering the program semantics):

```
1. L : LOAD RA, Ri, Ra
2. LOAD RB, Ri, Rb
3. INCR Ri
4. ADD Ra, Rb, Ra
5. IF < Ri, RN, L, delayed_branch
6. STORE RC, Ri, Rc
7. END
```

In this way, only the logical dependence of distance 2 of INCR on IF remains (ADD on STORE is interleaved with INCR on IF, thus it has no effect), with no branch degradation. By the execution simulation, we can verify that:

\[ T = \frac{7}{6} t \quad \varepsilon = \frac{6}{7} \]

with a notable improvement. For example, the completion time passes from the non-optimized version

\[ T_c = 6 N T = 6 N \frac{10}{6} t = 10 N t = 20 N \tau \]

to the final optimized version:

\[ T_c = 6 N T = 6 N \frac{7}{6} t = 7 N t = 14 N \tau \]

For example, with \( N = 10^9 \) and \( \tau = 0.4 \) nsec, \( T \) passes from 8 sec to 5.4 sec.

**Cost model**

Instead of evaluating a program by the graphical simulation, the theory of ILP architectures (see M. Vanneschi, Architettura degli Elaboratori, Chapter XI, and integration teaching material) provides an analytical cost model for every architectural variant: unit-latency EU and parallel-pipelined EU, scalar and superscalar, multithreaded architecture. It is out of the scope of this document to describe these cost models: the interested reader is referred to the mentioned book.

### 1.2 Data dependencies

Formally, the feasibility of optimizations is based on the application of Bernstein conditions, to discover data dependencies between instructions. As studied in Part1, Section 10, in general a sequential computation

\[ F_1(D_1) \rightarrow R_1; F_2(D_2) \rightarrow R_2 \]

can be transformed into a computation in which \( F_1 \) and \( F_2 \) are executed simultaneously, or in the reverse order, if
Conversely, the conditions for the existence of data dependencies are, in general:

\[
\begin{align*}
R_1 \cap D_2 &= \emptyset \\
R_1 \cap R_2 &= \emptyset \\
R_2 \cap D_1 &= \emptyset
\end{align*}
\]

The terminology associated to the kinds of data dependencies reveals important aspects of their meaning.

*True dependencies* are the fundamental ones to ensure computation consistency at any level and for any computational model and architecture.

*Output dependencies* are fundamental too. However, in the ILP case, this kind of dependence is not so critical to be avoided if a sufficiently large number of general registers is available (as in Risc machines) and they are allocated properly. When only few registers exist (as in many Cisc machines), register reuse is forced and the output dependencies become quite critical. In these cases, since the compiler is not able to avoid all possible output dependencies, the so-called *Register renaming* technique is applied. It consists in a *dynamic allocation* of assembler-visible registers into a much larger number of additional firmware-visible registers. In our architectural model, this technique can be implemented in EU Master using suitable firmware resources, such a mapping table.

In the ILP case, *anti-dependencies* deserve a special attention. As it happens in parallel micro-operations, in ILP architecture this kind of dependence is a *source of useful parallelism*, instead of being an impediment to parallelism exploitation. For example, the following sequence

\[
\begin{align*}
&j. \quad \text{LOAD} \ RX, Ri, Rx \\
&j-1. \quad \text{INCR} \ Ri
\end{align*}
\]

is affected by an anti-dependence on register RG[Ri]. However, in an ILP machine instructions \(j\) and \(j+1\) are not dependent each other: as soon as instruction \(j\) is initiated by IU, the value of RG[Ri] is saved or, better, it is used to compute the address RG[RX] + RG[Ri]. Thus, when EU will modify RG[Ri] in executing instruction \(j+1\), this modification has no effect on the correct IU state. In conclusion, the compiler can usefully introduce anti-dependencies IU-EU in the optimized code.

### 1.3 Impact of parallel-pipelined EU on performance and optimizations

Let us study the performance impact of a parallel-pipelined EU to execute *fixed-point multiplication/division* operations. Let us assume (as it is realistic) that EU is has a parallel-pipeline structure, where the fixed-point multiplication/division operations are implemented with 4 pipeline stages.

The *dispatcher unit* (see Section 19.4.2, figure 9) is in charge of directly executing “short” arithmetic instructions, like addition/subtraction, and the final part of Load instructions. The dispatcher unit will also be called *EU Master* unit.
Fixed and floating point General Registers are contained in EU Master, that dispatches the proper operand values to the arithmetic pipeline when needed, updates registers according to the received results, and sends such results to the Instruction Unit for consistent local update.

Let us consider the following program:

\[
\text{int } A[N], B[N], C[N];
\]

\[
\forall i = 0..N-1: C[i] = A[i] \times B[i] + A[i] / B[i]
\]

compiled in D-RISC as follows, by using only the delayed branch optimization:

1. LOAD RA, Ri, Ra
2. LOOP: LOAD RB, Ri, Rb
3. MUL Ra, Rb, Rx
4. DIV Ra, Rb, Ry
5. ADD Rx, Ry, Rc
6. STORE RC, Ri, Rc
7. INCR Ri
8. IF < Ri, RN, LOOP, delayed_branch
9. LOAD RA, Ri, Ra

The execution simulation of a generic loop iteration is the following:

A single pipelined fixed-point unit (INT Mul in the figure) is sufficient to execute in parallel the multiplication (3) and the division (4), because they are fully independent (the reader can verify such independence through the application of Bernstein conditions).

In addition to logical dependencies induced by EU on IU (called IU-EU logical dependence; in the example, 5 on 6 and 7 on 8), with a parallel EU we have also to deal with EU-EU logical dependencies. In our example, MUL and DIV induce (actually, DIV induces) an EU-EU dependence on ADD (5).
In general, EU-EU dependencies have effect on IU-EU dependencies, according to the ordering of some instructions: in fact, the latency of the pipelined operations introduce a further increase of the EU response time, with respect to the ideal case in which such operation has unit latency. In general, the net effect is a greater delay affecting the IU-EU dependencies. In our example, the delayed execution of 5 (waiting in EU Master for the result of 4, because of the EU-EU dependence induced by 4 on 5) sensibly increases the bubble in IU caused by the dependence of 5 on 6.

On the contrary, the IU-EU dependence induced by 7 on 8 is not affected by the parallel-pipelined EU structure, since instruction 7 is “short” (executed by EU Master with unit latency) and there is no residual effect of “long” operations (which have been completed, because 7 is executed after 5).

The performance metrics are:

\[ T = \frac{15}{8} t \quad \varepsilon = \frac{8}{15} \]

The reader is invited

- to compare these metrics with the corresponding ones in the ideal case of a CPU with unit-latency EU;
- to introduce additional optimizations against logical dependencies too, and to evaluate their effect.

1.4 In-order vs out-of-order behavior: static vs dynamic optimizations

In all the previous examples the CPU units behave in-order, that is they process instructions in the same order in which they are read from IM. For example, observe the in-order behavior of both IU and EU in the previous execution simulation.

In general, the CPU architecture contains hardware queues in front of every unit, notably a QI queue from IM to IU subsystems, a QR queue from IU to DM, a QIE queue from IU to EU, and a QD queue from DM to EU. If these queues have a FIFO discipline, in an in-order architecture a new element from a queue is not extracted until the previous one has not been fully processed by the unit.

In-order architectures rely on static (i.e. at compile time) optimizations to exploit the CPU resources in the most efficient way as possible.

Out-of-order behavior is possible: the goal is to introduce dynamic (i.e. at run time) optimizations. In principle, the same instruction reordering, that is performed statically by an optimizing compiler, can be performed dynamically.

In an in-order architecture the role of data-dependence discovery is played by the compiler, in an out-of-order architecture by the firmware interpreter of the various CPU units.

The simplest form of out-of-ordering can be exemplified as follows. Let us consider the previous example with a parallel-pipelined EU: Instruction 7 could have been processed in IU before instruction 6 without altering the program semantics. That is, while IU is waiting for the result of instruction 5 as input value of instruction 6, IU observes the next instruction at the head of the QI queue (instruction 7), and, by the application of the Bernstein conditions, verifies that it is independent from instruction 6. So, instruction 7 is prepared by IU, sent to EU, and executed by EU, as soon as possible: in this way, the resolution of the logical dependence of 7 on 8 is anticipated, with a notable improvement of the service time.

The example is meaningful also because of the useful application of anti-dependencies in

6. STORE RC, Ri, Re
7. INCR Ri

to exploit useful parallelism.
The reader is invited to show the execution simulation by applying this form of out-of-ordering.

It is interesting to know that the same, or comparable, service time can be achieved by applying static optimizations in an in-order architecture. In the example, we apply optimizations to logical dependencies by replicating the two LOAD instructions after the INCR, and by applying delayed branch by moving the STORE after the IF:

1. LOAD RA, Ri, Ra
2. LOAD RB, Ri, Rb
3. LOOP:
   - MUL Ra, Rb, Rx
   - DIV Ra, Rb, Ry
   - ADD Rx, Ry, Rc
   - INCR Ri
4. LOAD RA, Ri, Ra
5. LOAD RB, Ri, Rb
6. IF < Ri, RN, LOOP, delayed_branch
7. STORE RC, Ri, Rc

The graphical simulation is similar (though not identical) to the out-of-order execution:

This “equivalence” result (between performance of statically and dynamically optimized versions of the same program) cannot be generalized rigorously, since it is affected by too many parameters charactering the firmware-assembler architecture and the compiler optimizations. However, the example is meaningful to show the essence of the problem.

Out-of-ordering can be applied to optimize the behavior of EU too, as it can be verified on the examples.

The out-of-ordering form, which has been exemplified, consists in preserving the FIFO queuing discipline. It is the most simple, yet effective, form of out-of-ordering.

More aggressive forms of out-of-ordering exist in many machines: they are based on an associative search in the input queues, whose discipline is no more FIFO. In principle, this forms introduce more powerful optimizations. The main problem of out-of-ordering is the architecture complexity, which grows very rapidly with the parallelism degree that is allowed in presence of dynamic reordering. The firmware architecture must adopt many sophisticated, yet complex, techniques to control and to ensure the consistency of the machine and program state (registers and data cache contents), such as

- unique identifiers associated to each stream element,
- replication of registers and/or register check-pointing,
- register renaming,
- branch prediction,

as well as a set of associated firmware algorithms. Though efficiently implemented (in some cases with elegant solutions), these techniques greatly contribute to increase the CPU chip complexity, thus its size and, most important, its power consumption.

Coupled with the superscalar and multithreading techniques (studied in the next sections), this fact has caused the phenomenon of the “Moore’s law stop”: not because the physical implementation of regularly increasing clock frequencies is no more feasible, instead because the cost and the power
consumption of CPU chips has become intolerable, and no relevant increase of performance corresponded to this cost and power consumption increase.

On the contrary, the simple *FIFO-preserving* approach to out-of-ordering does not increase the architecture complexity appreciably and often gives interesting performances, especially if used in connection with optimizing compilers. According to this approach, the compiler has a further degree of freedom in choosing the optimizations according to the program characteristics: in some cases static optimizations are effective, in other cases they are of uncertain application (e.g. in presence of many branches and data dependent loop durations). In the latter situations, the compiler can rely on the predictable behavior of FIFO-preserving dynamic optimizations too.

Last but not least: *why* complex out-of-ordering techniques and dynamic optimizations? The main motivation has a strictly market-driven nature: the requirement to utilize portable programs for which only the executable binary code is available (binary *legacy* code) for a specific assembler machine (e.g. x-86 or mips). Passing from a computer model to the next (with the same assembler machine), the computer player tries to exploit the technological and architectural evolutions at best, also for programs that (by definition) cannot be recompiled.

Only when a technological revolution actually occurs, this trend is abandoned in favor of “new” application programs for which the source version is available. At this point, a good trade-off is represented by simpler (in-order or “controlled” out-of-order) architectures and optimizing compilers.

As a matter of fact, we are just in the middle of such a revolution, caused by the multicore approach to computer architecture and programming.
2. Superscalar CPU architectures

The superscalar CPU is an extension of the pipeline architecture in which each stream element contains $n$ instructions, with $n > 1$ (n-way superscalar). Ideally, every time slot $t = 2\tau$, $n$ instructions are fetched by IM and prepared by IU, and at most $n$ instructions will be processed also by the other subsystems DM and EU. That is,

$$T_{id} = \frac{2\tau}{n} \quad \psi_{id} = \frac{n}{2\tau}$$

Typical values of $n$ are in the range $n = 2$ (base case and current trend in highly parallel multicore) to $n = 4$. CPU for more powerful servers, built in the last decade, had $n = 8$ or even more.

The base architecture (2-way superscalar) does not pose realization problems, since in two clock cycles a sequential IM is able to read two instructions, a sequential IU to prepare two instructions, DM to start reading/writing of at most two data, EU to start or to complete at most two instructions.

Higher values of $n$ implies a parallel realization of instruction and data caches (interleaving, or long word).

The parallel-pipeline EU structure, with a suitable number of functional units, is able to offer the requested average bandwidth. A typical realization provides from $n/2$ to $n$ long fixed-point and floating-point functional units.

In principle, a single-unit IU can process any number of instructions per second, since the requested functions could be implemented by combinatorial circuits. However, the complexity of such an IU increases rapidly with $n$. For high values of $n$, a parallel realization of IU is mandatory.

All the architecture complexity problems are much more tractable for Risc machines. For this reason, even Cisc machines adopt the technique to interpret a primitive Cisc instruction through a sequence of Risc-like instructions. All the more reason, out-of-ordering techniques greatly contributes to increase the architectural complexity.

With respect to the scalar architecture, performance degradations are relatively greater, i.e. performance gradually increases and efficiency sensibly decreases with $n$. This is also due to the fact that sequential programs lack parallelism even at instruction level, and the utilization factor, in the client-server queueing model of CPU, increases with $n$.

2.1 VLIW superscalar architecture

Grouping $n$ instructions in a stream element could be done respecting the order of the original program code, independently of their semantics and contents. This solution is often a scarcely efficient solution. It is sufficient to think about logical dependencies, IU-EU and EU-EU, between instructions of the same stream element. Moreover, if a branch instruction is in position $m < n$ of a stream element, then all the successive $n-m$ instructions could be wasted. In fact, the branch prediction technique has been introduced just to alleviate this problem in superscalar machines.

A clean and elegant solution has been offered by the so-called Very Long Instruction Word (VLIW) model, which makes intensive use of compile-time optimizations. In VLIW terminology, a long instruction is the implementation of a stream element. The main feature of a VLIW long instruction is that it is composed of $n$ independent instructions, of which at most one may be a branch instruction provided that the target instruction is the first of a distinct long instruction.

In order to respect these constraints, when it is not possible to fill the long instruction with useful independent instructions, the compiler inserts a proper number of NOP instructions. In a sense, NOP are "static bubbles", i.e. bubble that cannot be eliminated and, for this reason, are recognized at compile time in order to have a predictable abstract machine and to sharply reduce the architectural complexity of IU and all the other units.
2.2 Examples

The following examples are referred to programs executed by a 2-way superscalar D-RISC CPU.

Example 1

1. LOAD RA, Ri, Ra
2. LOAD RB, Ri, Rb
3. ADD Ra, Rb, Rx
4. INCR Rb

In the scalar architecture we have: \( T = t = T_{id} \) and \( \varepsilon = 1 \).

In the 2-way superscalar architecture, let us adopt the following syntax for long instructions:

1-2. LOAD RA, Ri, Ra | LOAD RB, Ri, Rb
3-4. ADD Ra, Rb, Rx | INCR Rb

Observe that the constraint of independent instructions per long instruction is satisfied. In this example, no NOP is necessary. The execution simulation is the following:

<table>
<thead>
<tr>
<th>IM</th>
<th>1-2</th>
<th>3-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>IU</td>
<td>1-2</td>
<td>3-4</td>
</tr>
<tr>
<td>DM</td>
<td>1-2</td>
<td></td>
</tr>
<tr>
<td>EU</td>
<td></td>
<td>1-2</td>
</tr>
</tbody>
</table>

The performance metrics are

\[ T = \frac{2}{4} = \frac{t}{2} = T_{id} \quad \varepsilon = 1 \]

The relative gain with respect to the scalar architecture is:

\[ s = \frac{T_{scalar}}{T_{superscalar}} = 2 \]

That is, we are in an ideal case in which a 2-way superscalar architecture actually doubles the performance of a scalar architecture. This is due to the absence of any performance degradation: branches, logical dependencies, and NOPs.

Example 2

Let us consider a program with one logical dependence IU-EU:

1. LOAD RA, Ri, Ra
2. LOAD RB, Ri, Rb
3. ADD Ra, Rb, Rx
4. STORE RC, Ri, Rx

In the scalar case: \( T = 6t/4, \varepsilon = 0.66 \).

In the superscalar case, the following long instructions are recognized:

1-2. LOAD RA, Ri, Ra | LOAD RB, Ri, Rb
3-x. ADD Ra, Rb, Rx | NOP
4-x. STORE RC, Ri, Rx | NOP
As shown in the graphical simulation,

<table>
<thead>
<tr>
<th></th>
<th>IM 1-2</th>
<th>3-x</th>
<th>4-x</th>
</tr>
</thead>
<tbody>
<tr>
<td>IU</td>
<td>1-2</td>
<td>3-x</td>
<td>4-x</td>
</tr>
<tr>
<td>DM</td>
<td>1-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EU</td>
<td>1-2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

the performance metrics are:

\[ T = \frac{5}{4} t, \quad \varepsilon = 0.4, \quad s = \frac{6}{5} = 1.2 \]

Because of degradations (one logical dependence IU-EU and two NOPs), the service time, though better than the scalar case, is far from one half \((s = 1.2)\), thus the efficiency is lower than the scalar case.

**Example 3**

This simple benchmark contains a branch (a jump):

1. LOOP: LOAD RA, Ri, Ra
2. LOAD RB, Ri, Rb
3. ADD Ra, Rb, Rx
4. GOTO LOOP

In the scalar case: \( T = \frac{5}{4} t, \varepsilon = 0.8 \).

In the superscalar case:

LOOP: 1-2. LOAD RA, Ri, Ra | LOAD RB, Ri, Rb

\[ T = \frac{3}{4} t, \quad \varepsilon = 0.66, \quad s = \frac{5}{3} = 1.66 \]

**Example 4**

This example contains an IU-EU logical dependence and a branch:

1. LOOP: LOAD RA, Ri, Ra
2. LOAD RB, Ri, Rb
3. ADD Ra, Rb, Rx
4. IF \( \neq 0 \) Rx, LOOP

In the scalar case: \( T = \frac{7}{4} t, \varepsilon = 0.57 \).

In the superscalar case:

LOOP: 1-2. LOAD RA, Ri, Ra | LOAD RB, Ri, Rb

\[ T = \frac{3}{4} t, \quad \varepsilon = 0.66, \quad s = \frac{5}{3} = 1.66 \]
Example 5: vector addition

Non-optimized version:

\[
\text{LOOP:} \quad 1. \text{LOAD} \ RA, R_i, Ra \\
2. \text{LOAD} \ RA, R_i, Rb \\
3. \text{ADD} \ Ra, Rb, Rx \\
4. \text{STORE} \ RC, R_i, Rx \\
5. \text{INCR} \ Ri \\
6. \text{IF} < \ Ri, RN, \text{LOOP}
\]

In the scalar case: \( T = 10 \frac{t}{6}, \varepsilon = 0.6 \)

In the superscalar case:

\[
\text{LOOP:} \quad 1-2. \text{LOAD} \ RA, R_i, Ra... | \text{LOAD} \ RA, R_i, Rb \\
3-x. \text{ADD} \ Ra, Rb, Rx | \text{NOP} \\
4-5. \text{STORE} \ RC, R_i, Rx | \text{INCR} \ Ri \\
6-7. \text{IF} < \ Ri, RN, \text{LOOP} | \text{NOP}
\]

Optimized version:

\[
\text{LOOP:} \quad 1. \text{LOAD} \ RA, R_i, Ra \\
2. \text{LOAD} \ RA, R_i, Rb \\
3. \text{INCR} \ Ri \\
4. \text{ADD} \ Ra, Rb, Rx \\
5. \text{IF} < \ Ri, RN, \text{LOOP, delayed_branch} \\
6. \text{STORE} \ RC, R_i, Rx
\]

In the scalar case: \( T = 7 \frac{t}{6}, \varepsilon = 0.86 \).

In the superscalar case:

\[
\text{LOOP:} \quad 1-2. \text{LOAD} \ RA, R_i, Ra | \text{LOAD} \ RA, R_i, Rb \\
3-4. \text{INCR} \ Ri | \text{ADD} \ Ra, Rb, Rx \\
5-6. \text{IF} < \ Ri, RN, \text{LOOP, delayed_branch} | \text{STORE} \ RC, R_i, Rx \\
7-8. \text{LOAD} \ RA, R_i, Ra | \text{LOAD} \ RA, R_i, Rb
\]
### Example 6 with parallel EU

Internal loop of matrix-vector product, with static optimizations:

1. LOAD RA, Rj, Ra
2. LOOPj:
   1. LOAD RB, Rj, Rb
   2. INCR Rj
   3. MUL Ra, Rb, Rx
   4. ADD Rc, Rx, Rc
   5. IF < Rj, RM, LOOPj, delayed_branch
   6. LOAD RA, Rj, Ra

In the scalar case: \( T = t, \varepsilon = 1. \)

In the superscalar case:

\[
\begin{align*}
1-2 &. \quad \text{LOAD} \ RA, Rj, Ra & | & \text{LOAD} \ RB, Rj, Rb \\
3-4 &. \quad \text{INCR} \ Rj & | & \text{MUL} \ Ra, Rb, Rx \\
5-6 &. \quad \text{ADD} \ Rc, Rx, Rc & | & \text{IF} < \ Rj, RM, \text{LOOPj, delayed\_branch} \\
7-8 &. \quad \text{LOAD} \ RA, Rj, Ra & | & \text{LOAD} \ RB, Rj, Rb
\end{align*}
\]

With high values of \( n \), in order to reduce the complexity of the various units, it is convenient to adopt parallel structures, instead of centralized ones.

The scheme in the following, with \( n = 8 \), contains four 2-way superscalar identical IUs, each one operating on two instructions of the long instruction. In this version, the program counter and the general register array, with associated synchronization semaphore registers, are centralized in a single copy (RU unit).

It can be shown that this centralization point can be eliminated, obtaining a fully replicated version.

The high-bandwidth data cache is implemented by an interleaved structure.

A fully replicated scheme for EU is shown in the successive figure.

<table>
<thead>
<tr>
<th>IM</th>
<th>1-2</th>
<th>3-4</th>
<th>5-6</th>
<th>7-8</th>
<th>3-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>IU</td>
<td>1-2</td>
<td>3-4</td>
<td>5-6</td>
<td>7-8</td>
<td>5-6</td>
</tr>
<tr>
<td>DM</td>
<td>1-2</td>
<td>3-4</td>
<td>5-8</td>
<td>7-8</td>
<td>5</td>
</tr>
<tr>
<td>EU-Mas</td>
<td>1-2</td>
<td>3-4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>INT Mul</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

\[
T = 5 t/6 \quad \varepsilon = 0.6 \quad s = 6/5 = 1.2
\]
Large bandwidth (interleaved or long word) IM

IU

IU

IU

IU

IU-EU

IU-DM

RU

IC

RG1

SEM-RG1

Interface and queues

IU-EU

IU-DM

Interface and queues from IU

EU_Master

RG0, sem

RF0, sem

RG1, sem

RF1, sem

RG2, sem

RF2, sem

RG3, sem

RF3, sem

Switch EU_Master – EU_Master, n x n

Switch EU_Master – FU, n x m

Queue

FU

FU

FU

FU

FU

FU

FU

FU
3. Multithreading

The transition from scalar to superscalar CPU architecture is not accompanied by proportional improvements of performance, mainly because of the limited degree of parallelism between instructions of a single sequential program. Degradations are caused by frequent data dependencies, and optimizations aim to mask latencies causing such dependencies. For these reasons, it has been introduced the idea of exploiting parallelism among instructions of distinct sequential programs running on the same ILP CPU. Provided that proper architectural supports to program concurrency are available, this idea implies that, during the same time slot \((t)\) or a limited number of time slots, the CPU units are simultaneously active in processing instructions belonging to distinct programs. In other words: using instruction level parallelism to achieve program level parallelism too.

This idea, called multithreading \((MT)\) for reasons to be explained successively, can be interpreted in two ways:

1. as a technological evolution of ILP architectures, in order to exploit the CPU resources more efficiently,
2. as an alternative to multiprocessor architectures, in particular multicore, or chip-multiprocessor \((CMP)\), architectures.

Point 2 deserves a first comment:

a) both MT and CMP aim to exploit the silicon chip area for the simultaneous execution of more than one sequential programs;

b) there is no doubt that the multiprocessor, and in particular the CMP line, is characterized by much higher degrees of parallelism, notably from \(10^1\) to \(10^3\) and over in the current trend, while a MT CPU can support only very few concurrent programs \((2 \sim 4\) in typical cases);

c) the two lines are not in contrast: the current trend in HPC is to have highly parallel machines, with a high number of CMP-based processing nodes, each node consisting of one or more processors, and each processor providing a limited MT capability.

In order to understand the nature of MT architecture, we just start by informally discussing the following (hard) problem: is a MT CPU, capable of executing \(m\) simultaneous threads, equivalent to a multiprocessor with \(m\) processors?

3.1 A simple example of multiprocessor vs multithreading

Let us consider the following simple parallel computation \(P\), composed of two independent modules:

\[
P \quad P_1 \quad P_2
\]

\[
P_1::
\begin{align*}
1. & \text{ MUL Ra1, Rb1, Rx1} \\
2. & \text{ DIV Rc1, Rd1, Ry1} \\
3. & \text{ ADD Rx1, Ry1, Rz1} \\
4. & \text{ STORE RC1, Ri1, Rz1}
\end{align*}
\]

\[
P_2::
\begin{align*}
a. & \text{ LOAD RA2, Rj2, Rp2} \\
b. & \text{ LOAD RB2, Rj2, Rq2} \\
c. & \text{ SUB Rp2, Rq2, Rr2} \\
d. & \text{ INCR Rr2} \\
e. & \text{ STORE RD2, Rj2, Rr2}
\end{align*}
\]
Let us execute \( P \) on two distinct architectures, realized with compatible technologies (single chip, same clock cycle, same cache, etc).

1) The first architecture is a **multicore** CMP with \( n = 2 \) identical, **scalar** CPUs. Assuming that \( P_1 \) and \( P_2 \) are loaded in the primary caches of CPU\(_1\) and CPU\(_2\), the service and completion times are given by (see graphical simulation):

\[
T_1 = 9 \frac{t}{4} = 2.25t \\
T_2 = 7 \frac{t}{5} = 1.4t \\
\]

\( T_{c1} \sim 9t \) \hspace{2cm} \( T_{c2} \sim 7t \)

For \( P \):

\[
T_c = \max (T_{c1}, T_{c2}) \sim 9t
\]

\[
\text{Total number of instruction executed on the multiprocessor: } N = 9
\]

\[
T \sim T_c / N = t
\]

That is, CMP is equivalent to a single CPU able to execute one instruction every \( t \) seconds.

2) The other architecture is a single \( n = 2 \)-way **superscalar** CPU with **multithreading**. This architecture is able to execute, in the same long instruction, two instructions both belonging to \( P_1 \), or both to \( P_2 \), or one to \( P_1 \) (\( P_2 \)) and the other to \( P_2 \) (\( P_1 \)). For example, the generated MT code for \( P \) may be:

\[
P::
\]

1-a. MUL Ra1, Rb1, Rx1 | LOAD RA2, Rj2, Rp2  
2-b. DIV Rc1, Rd1, Ry1 | LOAD RB2, Rj2, Rq2  
3-c. ADD Rx1, Ry1, Rz1 | SUB Rp2, Rq2, Rr2  
4-d. STORE RC1, Ri1, Rz1 | INCR Rr2  
x-e. NOP | STORE RD2, Rj2, Rr2

with the following service and completion time:

\[
T = 10 \frac{t}{9} \quad T_c \sim 10t
\]

which are comparable to the performance metrics of CMP. They can even be equal to the CMP metrics with suitable static optimizations, for example:

\[
\text{1-2. MUL Ra1, Rb1, Rx1 | DIV Rc1, Rd1, Ry1}
\]

\[
\text{a-b. LOAD RB2, Rj2, Rq2 | LOAD RA2, Rj2, Rp2}
\]

\[
\text{3-c. ADD Rx1, Ry1, Rz1 | SUB Rp2, Rq2, Rr2}
\]

\[
\text{4-d. STORE RC1, Ri1, Rz1 | INCR Rr2}
\]

\[
x-e. NOP | STORE RD2, Rj2, Rr2
\]

obtaining:

\[
T = t \quad T_c \sim 9t
\]

In conclusion, the two architectures exploit the parallelism of \( P \) according to different approaches, but the performance results are comparable or even equal.

CMP executes processes \( P_1 \) and \( P_2 \) on the distinct scalar CPUs, each one with its own degradations (bubbles), as shown in the following figure. We observe that there are some time slots (\( t \)) during which both IUs are active, or both are blocked, or one active and the other blocked.

This simple observation is the intimate motivation for the MT architecture:

- \( a) \) if the single IU is able to process both instructions of a long instruction, then the maximum parallelism between processes is exploited,

- \( b) \) even more meaningful: if a bubble occurs in a process, it is possible that the other process is able to fill the bubble itself.
Points \(a\) and \(b\) exemplify the two “souls” of MT parallelism: \(a\) finding independent instructions to be executed simultaneously, \(b\) masking latencies.

This can be shown by comparing the execution simulations of CMP and MT (the non-optimized version is shown, since it is the most explanatory case):

### CMP

<table>
<thead>
<tr>
<th>IM</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>IU</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>DM</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>EU-Mas</td>
<td>1</td>
<td>2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>INT Mul</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>IM</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>IU</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>DM</td>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EU-Mas</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>INT Mul</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### MT

<table>
<thead>
<tr>
<th>IM</th>
<th>1-a</th>
<th>2-b</th>
<th>3-c</th>
<th>4-d</th>
<th>x-e</th>
</tr>
</thead>
<tbody>
<tr>
<td>IU</td>
<td>1-a</td>
<td>2-b</td>
<td>3-c</td>
<td>d</td>
<td>x-e</td>
</tr>
<tr>
<td>DM</td>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EU-Mas</td>
<td>1</td>
<td>a</td>
<td>2</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>INT Mul</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
In general it is interesting to compare the following architectures:

1) multiprocessor with \( p \) superscalar CPUs, each with \( n/p \) ways,
2) multithreaded architecture on a single superscalar \( n \)-way CPU,
or a variety of mixed cases.

Of course, lines 1 and 2 are not comparable for any parallelism degree \( n \):

1) the multiprocessor architecture is scalable by nature, even with very high values of \( n \), provided that “good” parallel programs are designed;
2) the single-CPU multithreaded architecture is meaningful only for low values of \( n \).

Currently, several CMP products have each core supporting a limited number of threads \((2 – 4)\). The clear trend is to provide many cores with simple (scalar or superscalar with very few ways) architecture, notably with in-order behavior.

The efficient exploitation of the MT feature in parallel programs, for high parallelism multiprocessors, is still an open research issue. The following section contains a first basic characterization to understand the problem.

### 3.2 Scalability of multithreading vs single thread

Let us consider again the example:

\[
\begin{align*}
P_1:: & \\
1. & \text{MUL } Ra_1, Rb_1, Rx_1 \\
2. & \text{DIV } Rc_1, Rd_1, Ry_1 \\
3. & \text{ADD } Rx_1, Ry_1, Rz_1 \\
4. & \text{STORE } RC_1, Ri_1, Rz_1 \\
\end{align*}
\]

\[
\begin{align*}
P_2:: & \\
a. & \text{LOAD } RA_2, Rj_2, Rp_2 \\
b. & \text{LOAD } RB_2, Rj_2, Rq_2 \\
c. & \text{SUB } Rp_2, Rq_2, Rr_2 \\
d. & \text{INCR } Rr_2 \\
e. & \text{STORE } RD_2, Rj_2, Rr_2 \\
\end{align*}
\]

As seen, the MT 2-way superscalar CPU has performance metrics:

\[ T = t \quad T_c \sim 9 t \]

Let us execute the same computation on a 2-way superscalar architecture with a single thread:

\[
\begin{align*}
1-2. & \text{MUL } Ra_1, Rb_1, Rx_1 \ | \ \text{DIV } Rc_1, Rd_1, Ry_1 \\
3-x. & \text{ADD } Rx_1, Ry_1, Rz_1 \ | \ \text{NOP} \\
4-x. & \text{STORE } RC_1, Ri_1, Rz_1 \ | \ \text{NOP} \\
a-b. & \text{LOAD } RA_2, Rj_2, Rp_2 \ / \ \text{LOAD } RB_2, Rj_2, Rq_2 \\
c-x. & \text{SUB } Rp_2, Rq_2, Rr_2 \ | \ \text{NOP} \\
d-x. & \text{INCR } Rr_2 \ | \ \text{NOP} \\
e-x. & \text{STORE } RD_2, Rj_2, Rr_2 \ | \ \text{NOP} \\
\end{align*}
\]

obtaining:

\[ T = 14 t/9 \quad T_c \sim 14 t \]

The scalability of the 2-thread version with respect to the 1-thread version is:

\[ s_{MT} = 14/9 = 1.6 \]

That is, the superscalar architecture is better exploited in MT modality. However, the performance gain is sensibly lower than 100%, in our example it is 60%. In fact, some initial benchmarks for Intel Xeon with Hyperthreading declared an improvement of 65% with respect to the Xeon architecture of the previous generation.

More in general, the following figure show very meaningful performance evaluation measures on the standard benchmark suite Spec95 and Splash2 in multiprogramming and parallel processing version. The scalability ranges is in the interval \(1.4 – 2.4\) with a number of threads in the interval \(2 – 8\).
This evaluation has been done assuming that each long instruction contains instructions belonging to distinct threads.

### 3.3 Multithreaded architectures

The parallel activities executed by a MT CPU, on a time slot basis, are referred to using the term thread, with a more specific meaning with respect to the traditional notion of OS-supported multiprogramming and/or of a set of concurrent activities defined in the same addressing space.

The meaning is a concurrent computational activity which is supported directly at the firmware level (to reinforce this idea, sometimes the term “hardware thread” is used). For this purpose the firmware architecture of a MT CPU, able to execute at most $m$ threads simultaneously, has to provide:

1. $m$ independent contexts, where, as usually, a context is represented by the program counter and the registers (general registers, floating point registers) visible at the assembler level;
2. a tagging mechanism, to distinguish instructions in execution, belonging to distinct threads, using shared resources of the pipeline structure (notably, EU Master and arithmetic functional units, or data cache);
3. a thread switching mechanism to activate context switching at the thread level.

Independent contexts could be emulated on a single register set. However, often they are implemented by multiple register sets ($m$ physically distinct copies of program counter, general registers, and floating point registers).

The key to understand MT is the existence of the specific run-time support of threads implemented directly at the firmware level (characteristics 1, 2, 3). From the programmer viewpoint, this may be not visible: a thread can be declared in the usual/traditional way, for example

- a user or a system thread (e.g. Posix thread)
- an applicative program or process.

However, the compiler of a MT machine is different from a traditional compiler: it produces proper code to link the firmware-level run-time support for threads.

Moreover, a suitable compiler makes it possible also to implement as threads some activities which are not explicitly defined by the programmer or by the system services, for example:

- a concurrent activity recognized automatically by the compiler in a sequential process (e.g. the so-called subordinated threads, microthreads, nanothreads),
- a concurrent activity which is generated directly at run-time.
To further characterize the MT architectures, let us introduce the term instruction issuing, or instruction emission, to denote the generation phase of a stream element by IM, in a single time slot, and its consequent processing in IU. The following class of MT architecture can be distinguished:

I. **single-issue** architectures: during a time slot only instructions belonging to a single thread are issued. Instructions belonging to distinct threads are issued in distinct time slots. This class can be implemented either on a scalar or on a superscalar architecture;

II. **multiple-issue** architectures: during a time slot instructions belonging to more than one thread can be issued. This class, which can be implemented only on superscalar architectures, is also called simultaneous multithreading (SMT). All the examples presented till now are relative to a SMT CPU.

In class I a further distinction consists in **interleaving** (IMT) vs **blocking** (BMT) architectures.

In IMT a fixed ordering is established among concurrent threads, e.g. if \( m = 2 \), instructions of thread_1 and of thread_2 alternate during consecutive time slots (thread switching occurs every time slot). In BMT there is no fixed ordering, and the thread switching occurs when an event that blocks the execution of an instruction of the running thread occurs (e.g. a logical dependence).

IMT and BMT models are not studied in these notes (the interested reader is referred to the teaching material of Architettura degli Elaboratori, or to the paper “A survey of processors with explicit multithreading”, ACM Computing Surveys, Vol. 35, Issue 1, March 2003).

In the following, we will focus on the SMT class, commercially introduced by Intel with the Hyperthreading model. Currently and in perspective, SMT represents the most relevant application of the MT architecture concept.

In the following, we will assume that the threads belonging to the same concurrent computation share the same addressing space. This is a necessary prerequisite to exploit the pipeline behavior, on a time slot basis, in presence of multiple simultaneous threads: their individual instructions and data must be allocated, or allocatable, in the primary instruction and data cache.

### 3.4 Simultaneous Multithreading

In a SMT \( n \)-way superscalar architecture, the \( n \) instructions of the same long instruction (or, in general, simultaneously issued) can belong to \( m \) threads, where

\[
1 \leq m \leq n
\]

In general \( m \) is variable, i.e. every long instruction has its own value of \( m \).

The presence of instructions, belonging to distinct threads, in the same stream element makes much easier to respect the independence constraint of a VLIW superscalar architecture.

Let us consider some simple examples.

In the first example we have \( m = n = 2 \) in all long instructions:

**Thread 1:**

1.1 MUL RA1, Ri1, Rx1
1.2 INCR Rx1
1.3 STORE RB1, Ri1, Rx1

**Thread 2:**

2.1 MUL RA2, Ri2, Rx2
2.2 INCR Rx2
2.3 STORE RB2, Ri2, Rx2

encoded and executed as:

```
1.1-2.1 MUL RA1, Ri1, Rx1 | MUL RA2, Ri2, Rx2
1.2-2.2 INCR Rx1 | INCR Rx2
1.3-2.3 STORE RB1, Ri1, Rx1 | STORE RB2, Ri2, Rx2
```
The two threads evolve with perfect synchronous parallelism at the clock cycle level. The service time of the whole parallel computation is one half with respect to the sequential execution in a non-MT architecture.

This example serves to introduce the SMT model, however it shows only a part of its potential power (maximum parallelism exploitation).

For example, latency masking is allowed too:

**Thread 1:**
1.1 LOAD RA1, Ri1, Rc1
1.2 STORE RC1, Ri1, Rc1

**Thread 2:**
2.1 LOAD RX2, Ri2, Rx2
2.2 ADD Rx2, Ry2, Rz2
2.3 INCR Ri2
2.4 INCR Rv2
2.5 MUL Rz2, ..., ...

Possible encoding and execution:

1.1-2.1 LOAD RA1, Ri1, Rc1 | LOAD RX2, Ri2, Rx2
2.2-2.3 ADD Rx2, Ry2, Rz2 | INCR Ri2
2.4-2.5 INCR Rz2 | STORE ...
1.2-x. STORE RC1, Ri1, Rc1 | NOP

The bubble in Thread 1 has been filled by other long instructions (in this example, belonging to the other thread).

### 3.4.1 Instruction composition and scheduling

The key of the SMT model is just the *proper composition of long instructions*, in order to meet the double goal: exploiting thread-level parallelism on a time slot basis, and masking latencies.

Instructions to be issued in the same time slot can be recognized *statically* (long instruction in the true meaning of the term) or *dynamically*.

In the dynamic case, the firmware interpreter of IM-IU is in charge of choosing instructions to be issued simultaneously and of composing them in the same steam element at run-time. This is done according to a *dynamic scheduling strategy*, that can also depend on non-predictable events, e.g. logical dependencies induced by variable execution time instructions, or cache misses.

Actually, the most challenging problem of SMT architectures is just *instruction composition* and *thread scheduling*. As usually, a way of reducing the complexity of a design problem is to exploit *constraints* in the form of computation patterns that are common to many applications: notably,
exploit the properties of stream-parallel and data-parallel structured paradigms. For example, if two map workers, \( W_{i1} \) and \( W_{i2} \), are implemented as threads allocated onto the same MT processing node, the best composition strategy is merely to insert in the same long word the same instructions of \( W_{i1} \) and \( W_{i2} \). Analogous examples can be done in other structured parallelism paradigms.

### 3.4.2 From multithreading to multicore

The two following figures show, respectively,

1. a possible schema of MT CPU chip with replication of IU and EU Master,
2. the structure of a dual-core CMP chip,

realized with the same silicon technology.
The architecture shown in the first figure can support the SMT model with \( m = 2 \) threads, where each thread utilizes a private subsystem (IU, EU Master), while the other resources (IM, DM, EU functional units, secondary cache) are shared by the threads in execution.

The second figure shows a shared-memory multiprocessor architecture, in which the two single-threaded CPUs occupy about the same area of the chip in the first figure, also taking into account that the number of EU functional units is one half for each CPU. Similarly to the MT architecture, the secondary cache is shared.

More in general, it is interesting to compare the CMP and SMT approaches from several viewpoints. The reader is invited to investigate this issue, along with reflections about the possibilities offered by the, currently popular, mixed approach: multiprocessor with SMT CPUs.

### 3.4.3 Two examples of multithreading exploitation in multiprocessor architectures

**Example 1: data parallelism**

Let us consider the vector addition computation with static optimizations, as studied in Example 5 of Section 2.1. In absence of cache misses, the completion time per iteration is

\[
T_{\text{iter,scalar}} = 7 t \quad T_{\text{iter,superscalar},2} = 5 t
\]

The same computation can be implemented, at the process level, as a data-parallel map, with data partition size \( g = N/p \). If each core is 2-way 1-thread superscalar, the completion time per worker is:

\[
T_{c,\text{superscalar},2} = g T_{\text{iter,superscalar},2} = 5 g t
\]

If each core is 2-way 2-thread superscalar, each worker can be implemented as two parallel, independent, identical threads, each one operating on \( g/2 \) integers (remember that they share the same addressing space). As said, the simplest, yet most efficient, way to define such threads is to insert correspondent instructions in the same long word: in this case, both thread parallelism exploitation and latency masking are achieved. Each thread has a completion time per iteration:

\[
T_{\text{iter,SMT},2} = 7 t
\]

The completion time of the whole 2-thread computation is:

\[
T_{c,\text{SMT},2} = g T_{\text{iter,SMT},2} = 7 \frac{g}{2} t = 3.5 g t
\]

In conclusion, with a 2-thread implementation of each worker, we achieve a performance gain equal to the scalability of 2-thread SMT with respect to 1-thread superscalar:

\[
s_{\text{MT},2} = \frac{T_{c,\text{superscalar},2}}{T_{c,\text{SMT},2}} = 1.4
\]

In other words, with \( p \) CPUs, each one with 2-thread SMT architecture, we actually achieve the effect of a parallelism increase of the architecture: the equivalent parallelism degree is not \( 2 \) \( p \), instead in general it is lower, in this example \( 1.4 \) \( p \):

\[
p * s_{\text{MT},2} = 1.4 \ p \ cores
\]

Assuming that the optimal degree of parallelism of the parallel program (see Part 1) is greater or equal, the multiprocessor version with SMT nodes has clear advantages, though not proportional to the number of threads per core.

The reader is invited to extend this multithreading application to other structured parallelism paradigms studied in Part 1.
Example 2: communication processor

By a multithreaded architecture of processing nodes, it is possible to emulate the solution with communication processor (KP): a thread is dedicated to the KP functions.

When a send primitive is invoked by the calculation thread IP, the IP instructions are composed (dynamically) with the KP instructions. Parameter passing is easy since the two threads share the same addressing space.

Similar considerations apply to the overlapping of CPU and I/O activities.