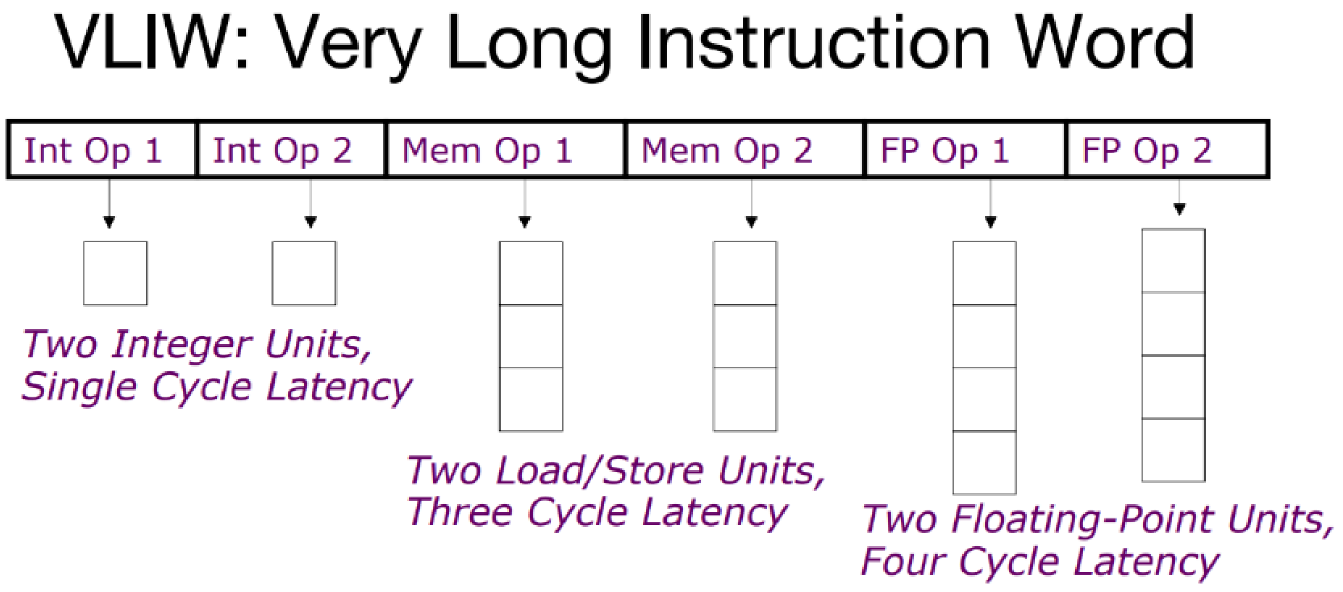
12.VLIW

Difference between ILP and Parallel programming is that when you try to program in parallel you want to execute chunk of the program at the same time. To have this execution you have to have different processors(transparent to the user). In ILP you have the ovrlap of instruction and is completely managed by processor/HW components.

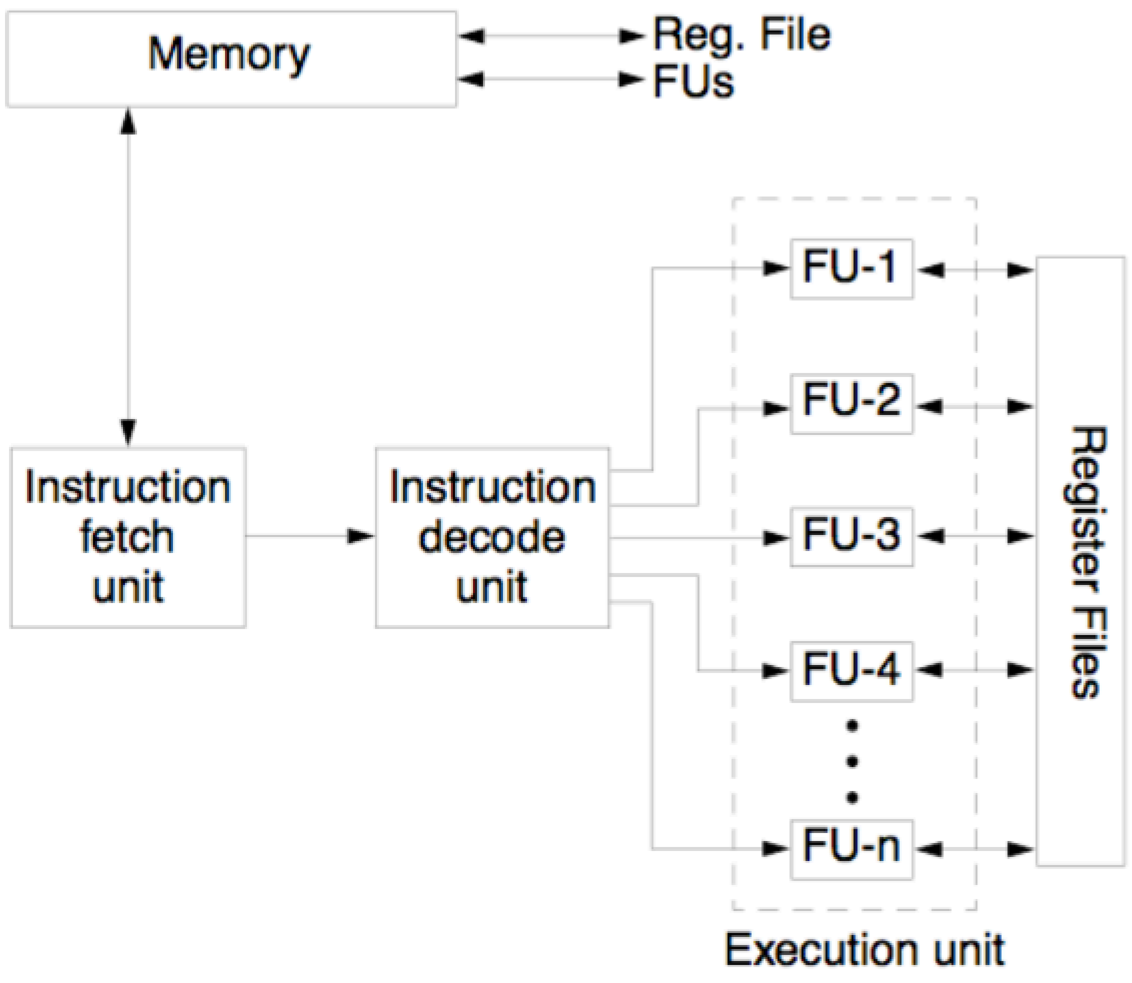
Superscalar and Very Long Instruction Words

You have long instruction scheduled in parallel by the compiler. Efficient in systems that are very regular. Fixed number of instructions (4-16). The idea is to have parallel instructions. Similar to ILP with CPI > 1. THe HW is simpler as the job is done by the compiler(You don't need rescheduling at HW level and don't want instruction reorder).
Some limitations:

- Cannot support instruction with different latency
- Cannot support different control flows
- Compiler decide the parallelism
  You try to separate the concept of computation and instructions. Set of operations and not an agglomerate of instructions. Distinct operations coming from the program and instructions generated by the compiler. Compiler decide the instructions to put in each block. The operations should start at the same time. Usually compilers use the same block for the same type of instructions. The number of operations are proportional to the resources at your disposal. Base on assumption that if instructions are in the same clock cycle the dependencies are already resolved.


VLIW: Very Long Instruction Word
Int Op 1
Int Op 2
Mem Op 1
Mem Op 2
FP Op 1
FP Op 2
Two Integer Units, Single Cycle Latency
Two Load/Store Units, Three Cycle Latency
Two Floating-Point Units, Four Cycle Latency

A VLIW Machine Configuration



VLIW Compiler Responsibilities

The compiler:
Schedules to maximize parallel execution Exploit ILP and LLP (Loop Level Parallelism)
It is necessary to map the instructions over the machine functional units
This mapping must account for time constraints and dependencies among the tasks
Guarantees intra-instruction parallelism
Schedules to avoid data hazards (no interlocks)
Typically separates operations with explicit NOPs
The goal is to minimize the total execution time for the program
You can determine the parallelism between instructions
If you don't have enough space to manage the dependencies you insert empty instructions(SW level).
Two strategies to support ILP:

- Dynamic Scheduling: Depend on the hardware to locate parallelism

- Static Scheduling: Rely on software for identifying potential parallelism
  Hardware intensive approaches dominate desktop and server markets

Static Scheduling

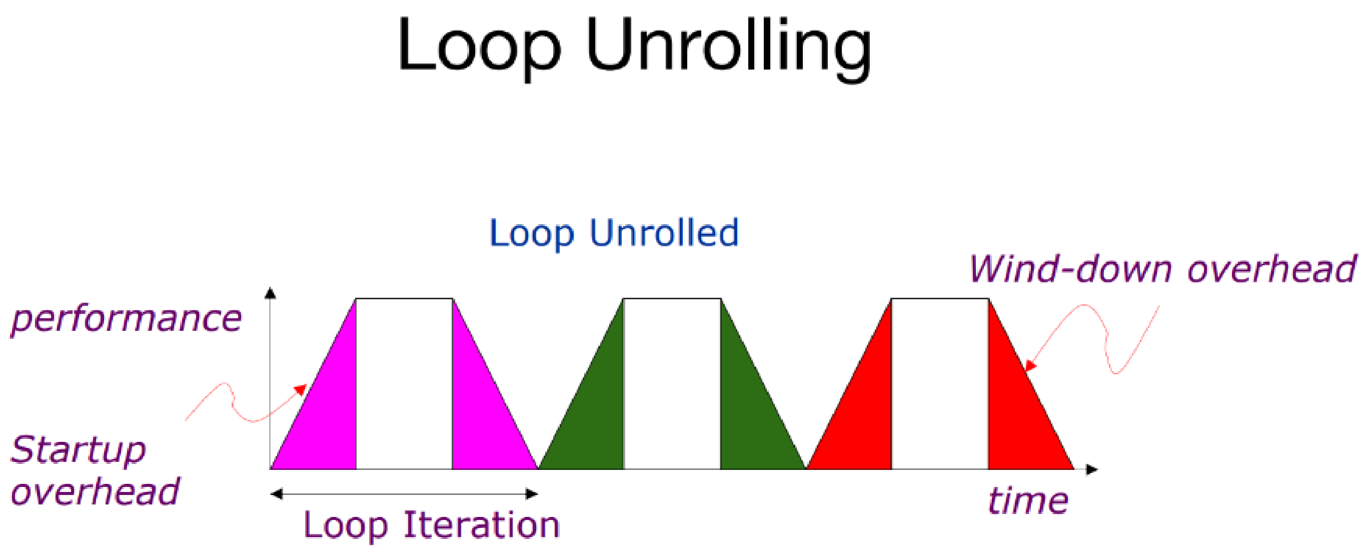
The idea is to use the processor/pipeline as much as possible. There are two ways: maximise the number of operations executed in each cycle and having non stalling FUs. The amount of parallelism in a basic block is very limited as instructions in the same block probably are correlated between them and you could have a controller. To rehave something that is efficient is to parallelize a singular block. One idea is to use trace scheduling(build a possible trace inside your control flow, then you execute different traces at the same time). The basic bloc is a compiler concept where there aren't jump instructions(all instructions inside are executed). With a static compiler we want to detect and resolve as much instructions as possible. To do so you can reorder the operations in your code. The idea is not stall postponing the execution. You expect that the code is dependencies free(as you cannot/don't want to detect them dynamically).
Pros

- Simple HW
- Easy to extend the FUs
- Good compilers can effectively detect parallelism
  Cons
- Huge number of registers to keep active the FUs
- Needed to store operands and results
- Large data transport capacity between
- FUs and register files
- Register files and Memory
- High bandwidth between i-cache and fetch unit
- Large code size
- Binary compatibility
  Knowing branch probabilities
- Profiling requires a significant extra step in build process
  Scheduling for statically unpredictable branches
- Optimal schedule varies with branch pat
  The compiler must be aware of the HW but the code produced is limited. You need the portability of the branches otherwise you cannot guarantee the dependencies free. You may have unpredictable branches that you want to analyse statically.

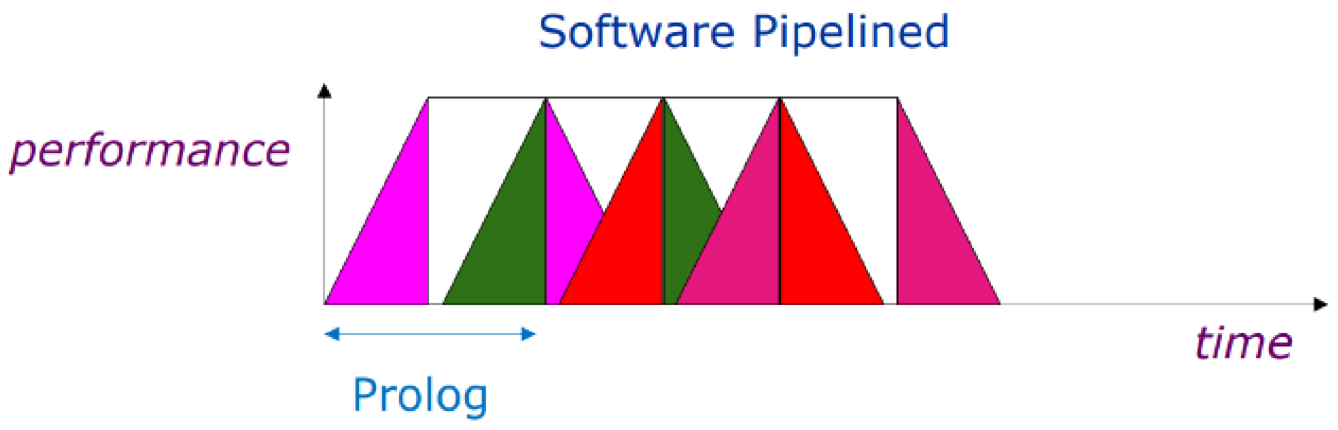
Static Scheduling: methods

- Simple code motion: concept based on the need of a specific register and time to be executed. You can identifies the dependencies and move operations, but the number could be limited

- Loop unrolling & loop peeling: optimizations at loop level and if the iterations haven't dependencies you could increase the instructions executed at the same time.



- Software pipeline: you pay the cost of starting and completing an instruction only once and not at the start of the iterations



- Global code scheduling (across basic block)
- Trace scheduling
- Superblock scheduling
- Hyperblock scheduling
- Speculative Trace scheduling

What if there are no loops?

The control instruct reduce the size of the block. If the data loop is made in multimedia solutions a No loop solutions are used for non standard solutions. You want to create a larger path, a trace(larger block without a loop, there are a certain set of loop as you know that you will execute a certain sequence of block). Now we analyze the instructions from the beginning to the end of the trace. If you know the number of iterations you can create a trace from a loop function concatenating the series of block in the loop for the number of time needed. Usually you work on the more frequent traces, considering a trace a block without a loop.

Trace Scheduling: finding common path and scheduling traces in that path independently
Scheduling in a trace rely on basic code motion but now has a global taste across more that one basic block by appropriate use of renaming
Compensation codes are needed for side entry points: i.e. points except beginning
and slide exit points: i.e. points except ending
Blocks on non common path may now have added overhead, so there must be a high probability of taking common path according to profile (may not be clear for
some programs)
Problems: compensation codes are difficult to generate specially for entry points

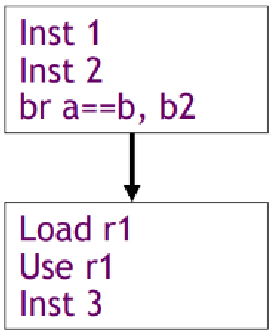
Code Motion in Trace Scheduling

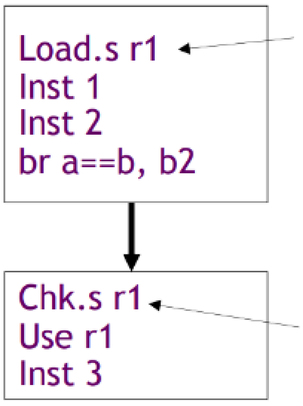
In addition to need of compensation codes there are restrictions on movement of a code in a trace:

- The dataflow of the program must not change
- The exception behavior must be preserved
  Dataflow can be guaranteed to be correct by maintaining two dependencies:
- Data dependency
- Control dependency
  There are two solutions to eliminate control dependency:
- By use of predicate instructions (Hyperblock scheduling) and removing the branch.
- By use of speculative instructions (Speculative Scheduling) and speculatively move an instruction before the branch.
  You have to execute the same set of instruction and the behaviour of the exception has to be precise. One of the principal aspect of Computer Architecture is that an architecture has to behave in the same way for the same inputs.
  We can use two solutions:
- Predicate instruction and removing the branch as the behaviour is in the instruction
- Speculative instructions where you flush the result if the evaluation is different than expected or you try to predict the branch result

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don’t cause exceptions





Speculative load never causes exception, but sets “poison” bit on destination register

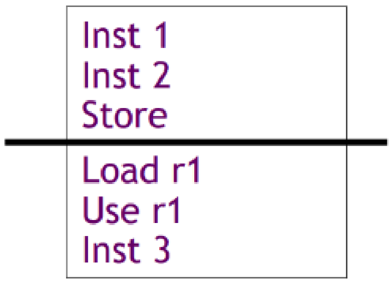
Check for exception in original home block jumps to fixup code if exception detected

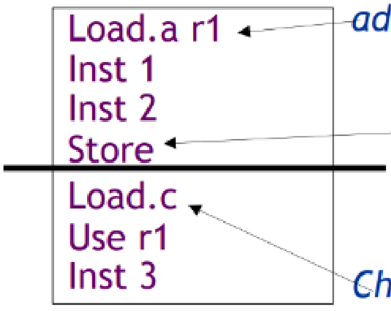
Can’t move load above branch because might cause spurious exception

Particularly useful for scheduling long latency loads early

Problem: Possible memory hazards limit code scheduling

Solution: Hardware to check pointer hazards





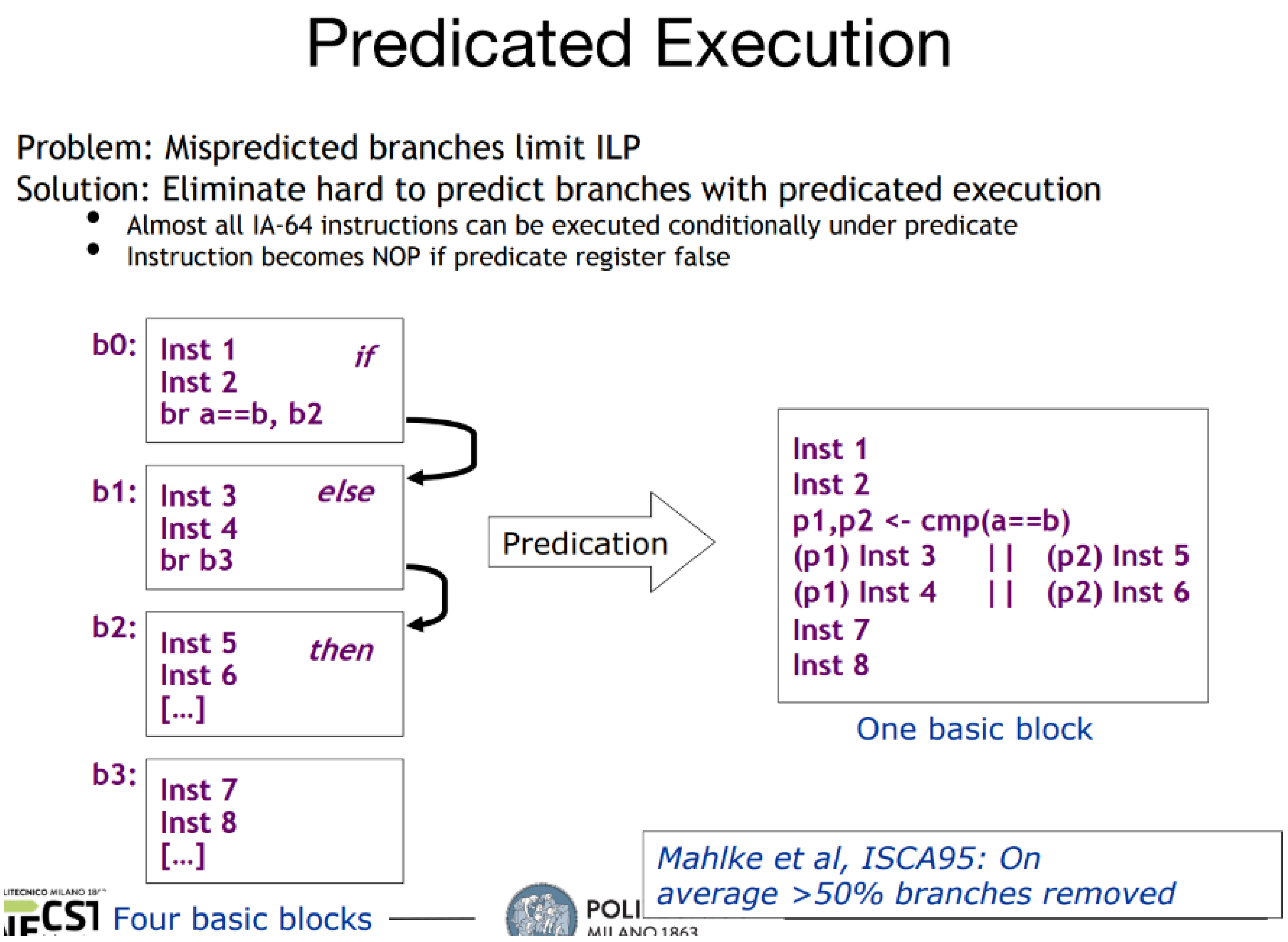
Data speculative load adds address to address check table

Store invalidates any matching loads in address check table

Check if load invalid (or missing), jump to fixup code if so

Can’t move load above store because store might be to same address

Requires associative hardware in address check table



Predicated Execution

Problem: Mispredicted branches limit ILP
Solution: Eliminate hard to predict branches with predicated execution

- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false

b0:
Inst 1
Inst 2
br a==b, b2
if
b1:
Inst 3
Inst 4
br b3
else
b2:
Inst 5
Inst 6
[...]
then
b3:
Inst 7
Inst 8
[...]
Four basic blocks
Predication
Inst 1
Inst 2
p1,p2 <- cmp(a==b)
(p1) Inst 3 || (p2) Inst 5
(p1) Inst 4 || (p2) Inst 6
Inst 7
Inst 8
One basic block

Mahlke et al, ISCA95: On average >50% branches removed

Can remove up to 50% of the branches of the program. Basically when you are really executing the operation the value is already evaluated and so the operation becomes a NOP.