Multicore Compilation Strategies and Challenges

BY- AASTHA VERMA
JITENDRA SINGH SOLANKI
SRINIVAS

Introduction

- Computers are hitting a performance limit.
- High performance computers are using a lot of energy.
- No of gates per chips and frequency are increasing.
- Performance is based on throughput and efficiency, performing computation in parallel to complete a larger volume of work in a shorter period of time.

The performance problem

- Transistors continue to shrink.
- More and more transistors fit on a chip.
- Chips run faster and faster and use lot of energy .
- Resulting into producing a lot of heat.

Performance Problem Solution : Multicore

- Two or more processors(multicores) on a chip
- Simple , slower , cooler processors .
- To increase performance and frequency multicore comes in picture.
- Processors can work on independent parts of the same task.
- Users and software will organize tasks to maximize Parallelism.

Multicores have an important advantage over current MPSoC (multiprocessor systems-on-chip) Systems—programmability.

Examples –

- Texas Instruments (TI) TMS320C6474 that has three eight-wide C64x very long instruction word (VLIW) cores,
- the Sun UltraSparc T1 that has eight cores,
- the Sony/Toshiba/IBM Cell processor that consists of nine cores,
- the NVIDIA GeForce 8800 GTX that contains 16 streaming multiprocessors,
- and the Cisco CRS-1 Metro router that utilizes 192 Tensilica processors.

PARALLELISM

- To increase frequency we are using parallel processing.
- Multiple cores we use multithreaded applications
- Single threaded applications can be parallelized.

INSTRUCTION-LEVEL PARALLELISM

- Multiple independent assembly instructions are executed in the processor at the same cycle . E.g.

```
load R1 -> R2                    add   R3 -> R3, "1"
add R3 -> R3, "1"                add   R4 -> R3, R2
add R4 -> R4, R2                 store [R4] -> R0
```

- Superscalar, general-purpose processors exploit ILP

- VLIW architectures exploit,

  - ILP in signal processing applications.

  -Independent instructions are packed into a single large instruction word and issued in parallel each cycle.

- The TI C6x is the most well-known family of VLIW DSPs that can issue eight operations each cycle.

DATA-LEVEL PARALLELISM

- Same instruction is performed on different pieces of data in parallel.
  - In some situations ,a single execution thread controls operations on all pieces of data.
  - In others, different threads control the operation, but they execute the same code.
- Forms of DLP -
  - SIMD (single-instruction , multiple data processors),
  - Vector computation models.

Examples -Intel SSE and SSE2, AMD 3DNow, ARM NEON, and Motorola AltiVec

LOOP-LEVEL PARALLELISM

- Independent iterations of the same loop are executed in parallel on different processors .

- Loops with independent iterations (DOALL loops) statically identified at compile time.

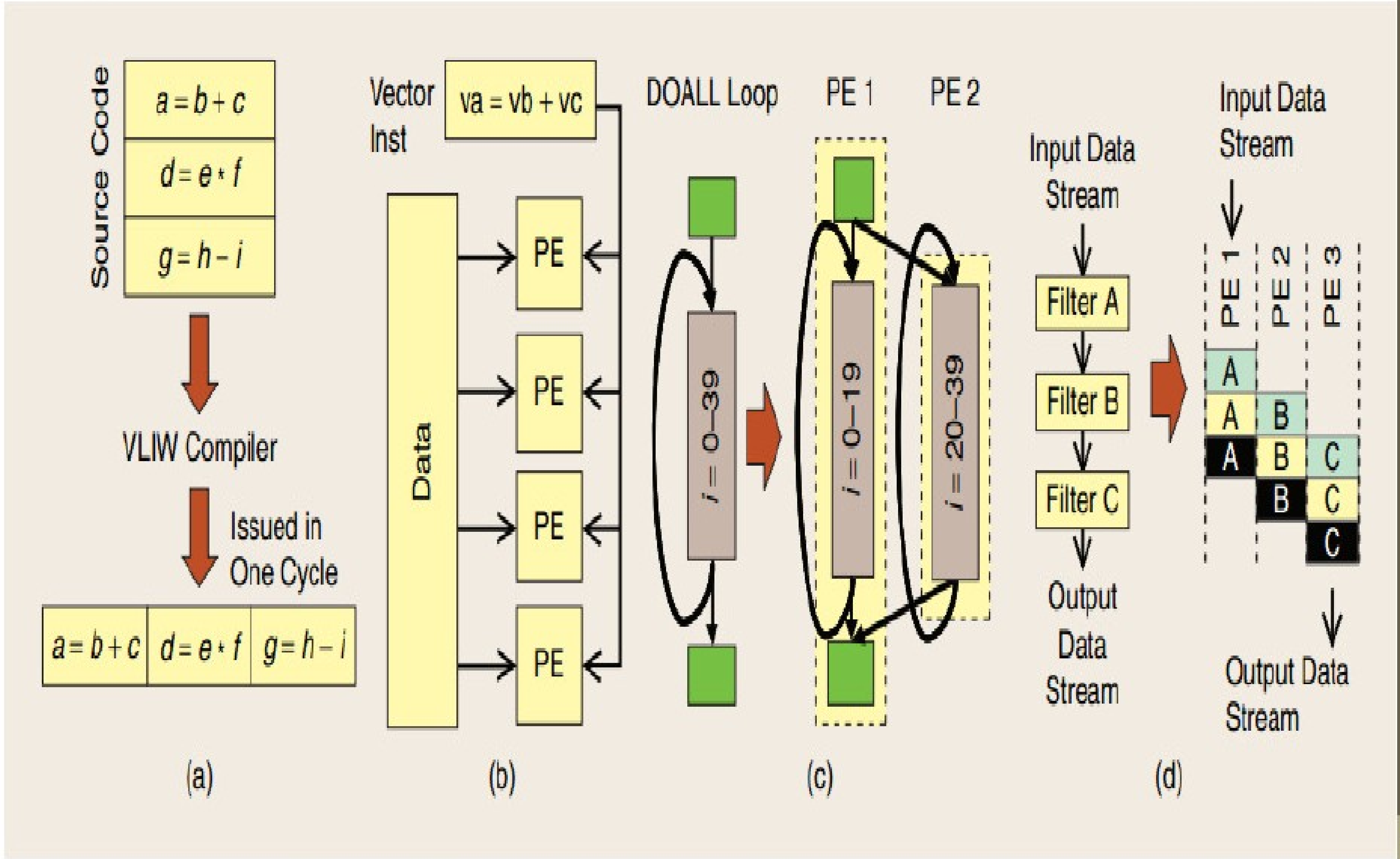
- compiler can exchange loop with outermost loop and maximize the exploited parallelism.

- parallel loops using parallel programming constructs as OpenMP application programming interface.

  application executed until reaches a parallel loop.

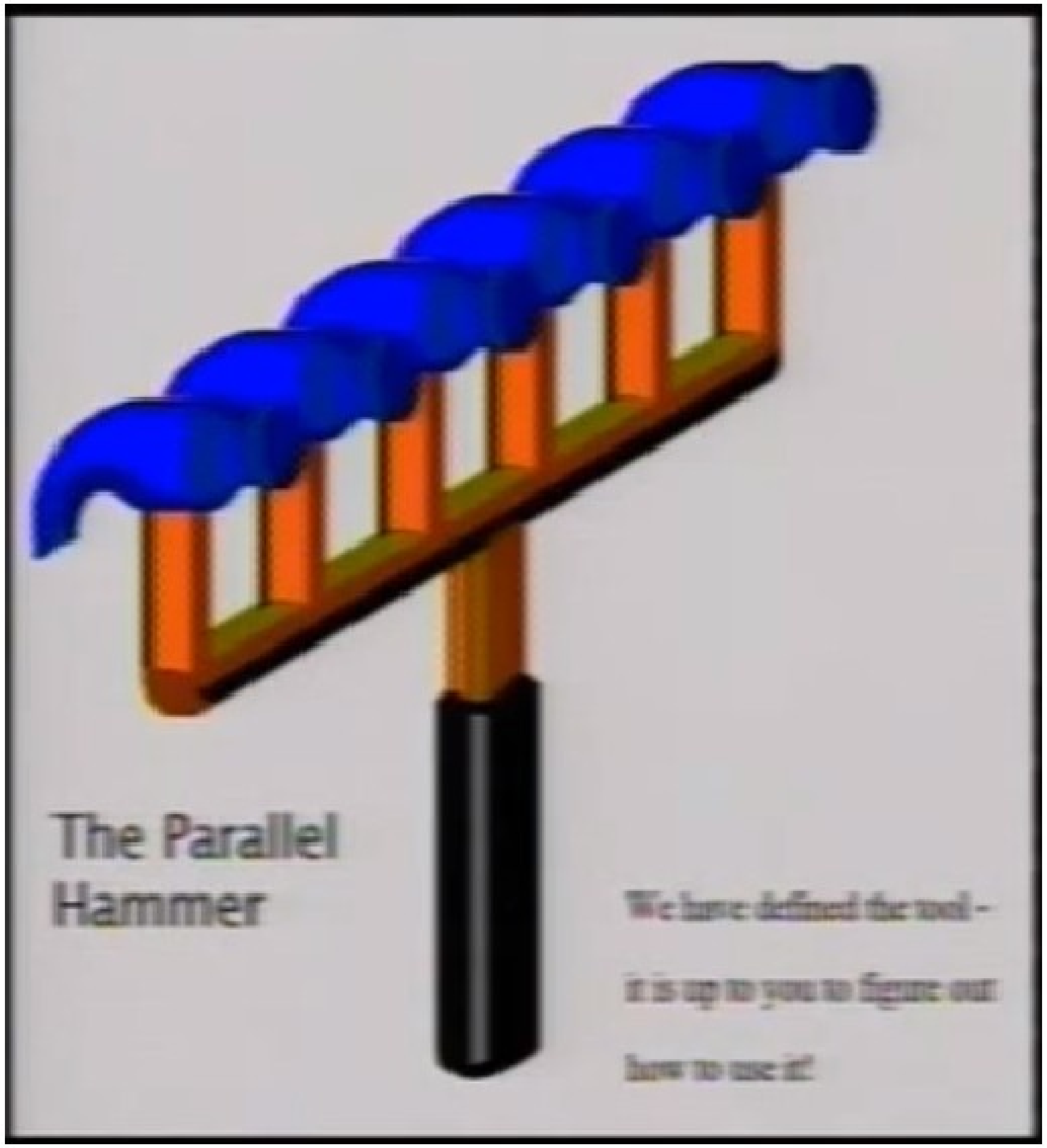
PIPELINE PARALLELISM--STREAM PROGRAMMING

- The application is decomposed into a series of stages.
- Each stage performs -
  - partial processing on a set of data,
  - forwards it to the next stage for processing, and
  - starts working on the next set of data.
- Here ,pipeline stages can run in parallel while working on different chunks of data.
- Stream languages enable the explicit specification of producer-consumer parallelism
- Examples of stream languages -StreamIt, Brook, CUDA, SPUR, Cg, Baker, and Spidle.



[FIG1] Various forms of exploiting parallelism: (a) ILP, (b) DLP, (c) LLP, and (d) the streaming model called pipeline parallelism.

• *PARALLELISM SOLVES THE PERFORMANCE PROBLEM ???*



ROLE OF THE COMPILER

- Signal processing applications have been traditionally developed in the form of sequential algorithms.

- Later evolved to more complex domain-specific vectorized Computations

- Coarse-grain parallelism :

Responsibilities can be broken down into two major categories based on when the action occurs: static, which occurs offline, and dynamic, which occurs online.

- Task through a combination of compiler analyses and transformations to understand the detailed memory access behavior

STATIC COMPILATION

- IDENTIFICATION AND EXTRACTION

Automatically parallelizing code poses two basic problems:

  - finding an exploitable region of code through analysis and
  - transforming the code into a parallel form.

MEMORY DEPENDENCE ANALYSIS

- Memory dependencies influences the performance of parallelized code.
- Careful scheduling, optimizing compiler can minimize the effect of data dependencies.
- Array dependence analysis
  - enumerates array dependences between loop iterations.
  - operates by encoding the constraints of the offset, or offsets in multiD arrays into linear constraint problem.
- Pointer analysis determines whether two pointers can alias.
- Shape analysis techniques use separation logic to determine the properties of recursive data structures .

AUTOMATIC VECTORIZATION

- Vector hardware performs the same operation on many inputs simultaneously.

- Analyses detect vectorization opportunities through array dependence analysis.

- Code can be vectorized if,

dependence distance < size of the hardware-specific vector

- Cray Fortran, the first automatic vectorizing compiler, automatically vectorize Fortran DO loops
    - without cross-thread dependencies,
    - and without control-flow statements.

- XL compiler –

- XL's vectorization targets loops with -
  - induction ("counting" in a regular way),
  - deduction (computed by sum, min, max, and product),
  - loop-scoped, and
  - loop-invariant accesses.

DOALL AND DOACROSS PARALLELIZATION

- *DOALL parallelization*
  - threaded multicore equivalent of vectorization techniques.
  - loop can execute in several threads in parallel,
    - has no cross iteration dependencies ,and
    - iteration space is statically divisible
  - highly efficient since no cross-thread communication while the loop is iterating.

DOACROSS parallelization -

- Adjacent iterations execute in alternating threads.
- one thread completes the loop's critical path, execution of the loop's next iteration begins on the next core.
- DOACROSS loops are universally applicable.

DECOUPLED SOFTWARE PIPELINING

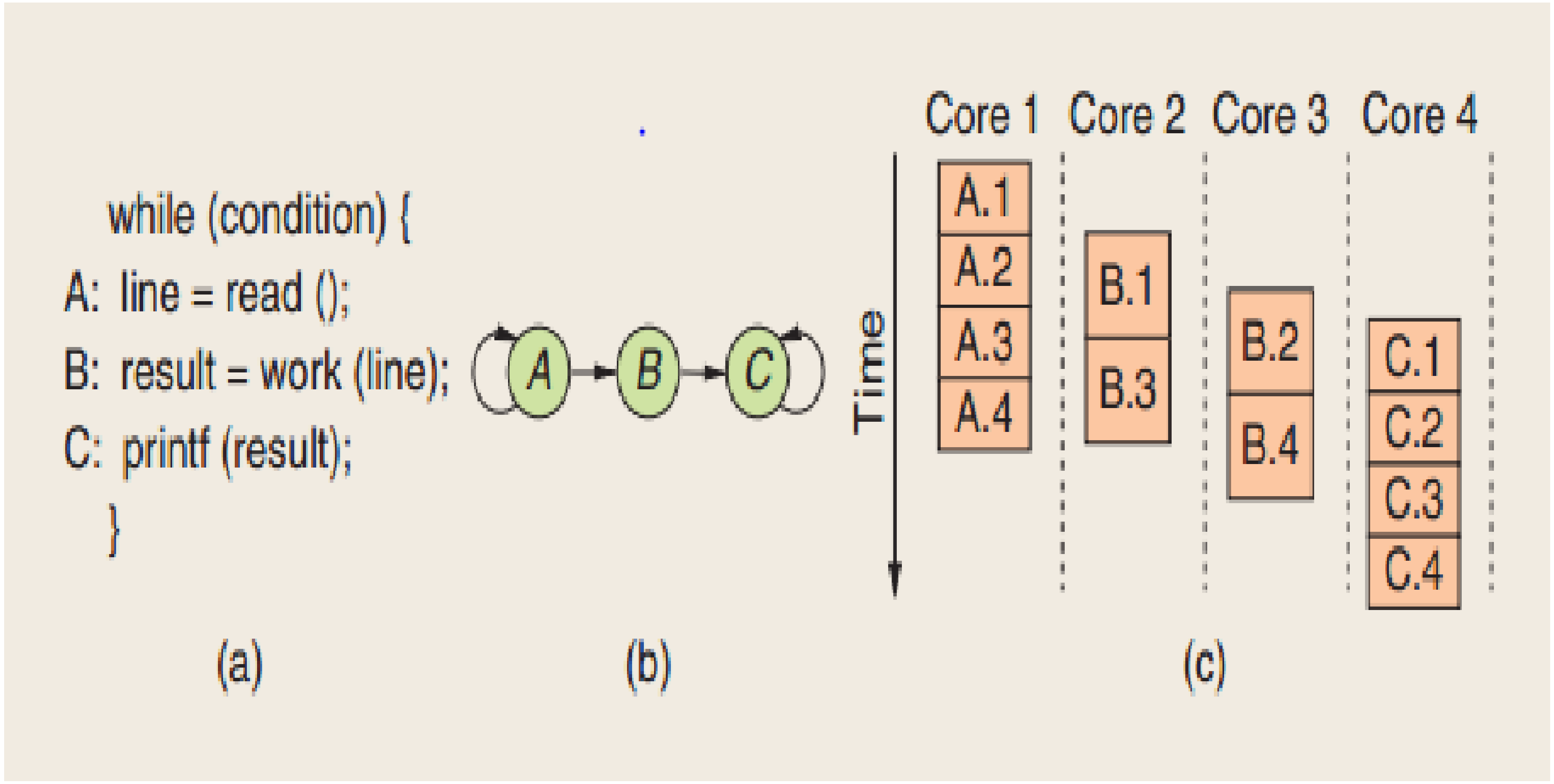
- Partitions each loop iteration such that communication happens from earlier stages to later stages.

- Communication between stages buffered $\longrightarrow$ latency tolerant

  - Doubling latency does not affect DOALL's performance, have no cross-thread communication

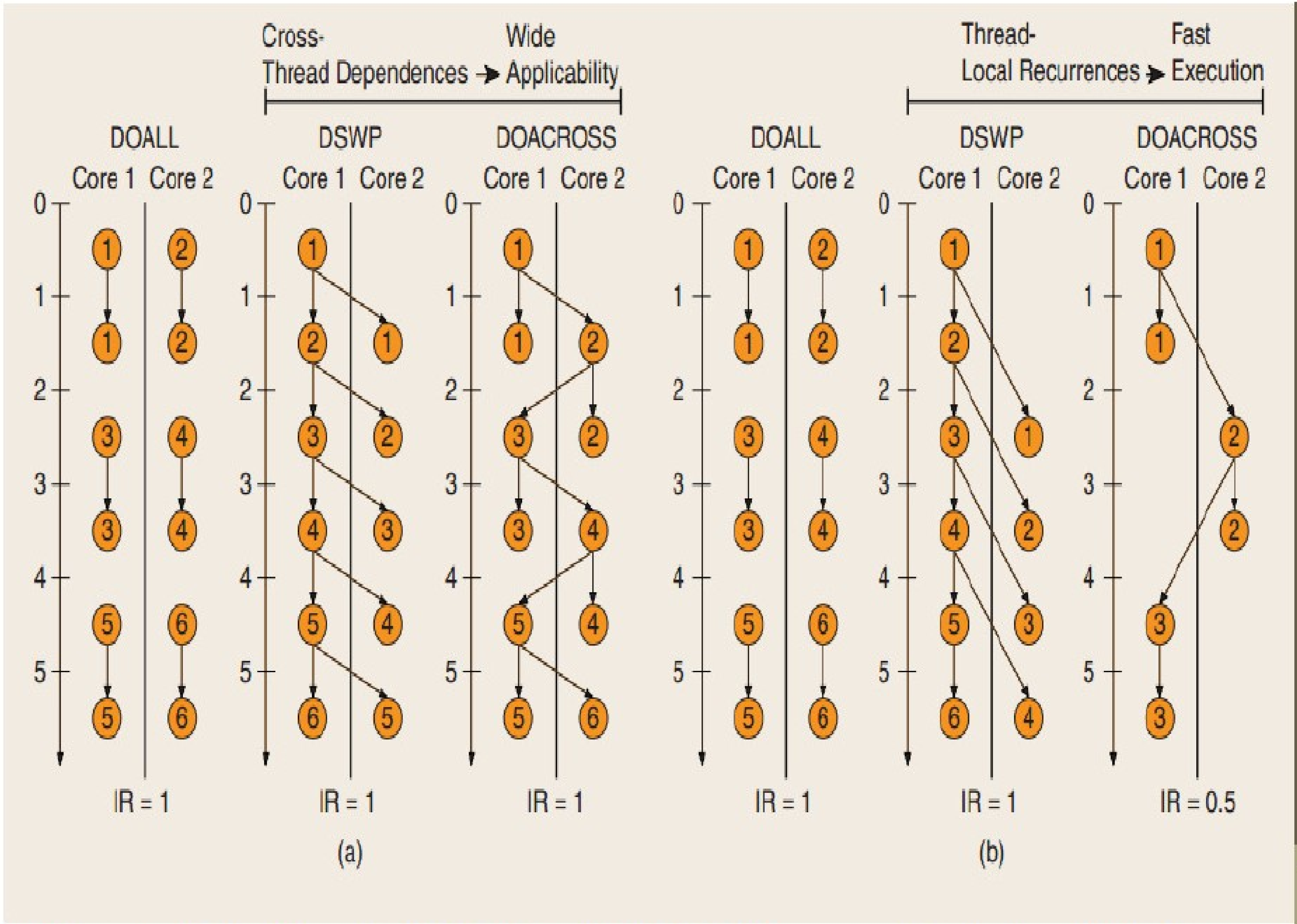
- loop parallelized for four threads,

  - first thread executing read function and communicating results to the second and third threads,
  - the second and third threads execute the work function of alternating iterations in parallel
  - the fourth thread maintains original order of printing.



```
   while (condition) {
A:  line = read ();
B:  result = work (line);
C:  printf (result);
   }
```

[FIG3] Example compiler parallelization using decoupled software pipelining to a traditionally sequential while loop. (a) Example code. (b) Static stage dependences. (c) Potential task execution.



[FIG2] **Execution schedules of loops using DOALL, DSWP, and DOACROSS. Solid lines represent intra-iteration, and dashed lines represent loop-carried (critical path) dependences. Initiation rate (IR) is the number of iterations started per cycle. (a) Communication latency = 1. (b) Communication latency = 2.**

MEMORY AND COMMUNICATIONS OPTIMIZATIONS

- Inefficient use of the memory system can cause cache stalls to dominate program execution.

- Loop tiling or loop blocking –

  Compiler reorders iterations of a loop to achieve better memory locality and thus better caching behavior.

e.g., if a loop updates in row-column order and an entire row cannot fit in the cache at once, then the data loaded into the cache gets evicted before being reused. Instead, loop tiling will break the iteration space into a series of small enough squares to fit in the cache .

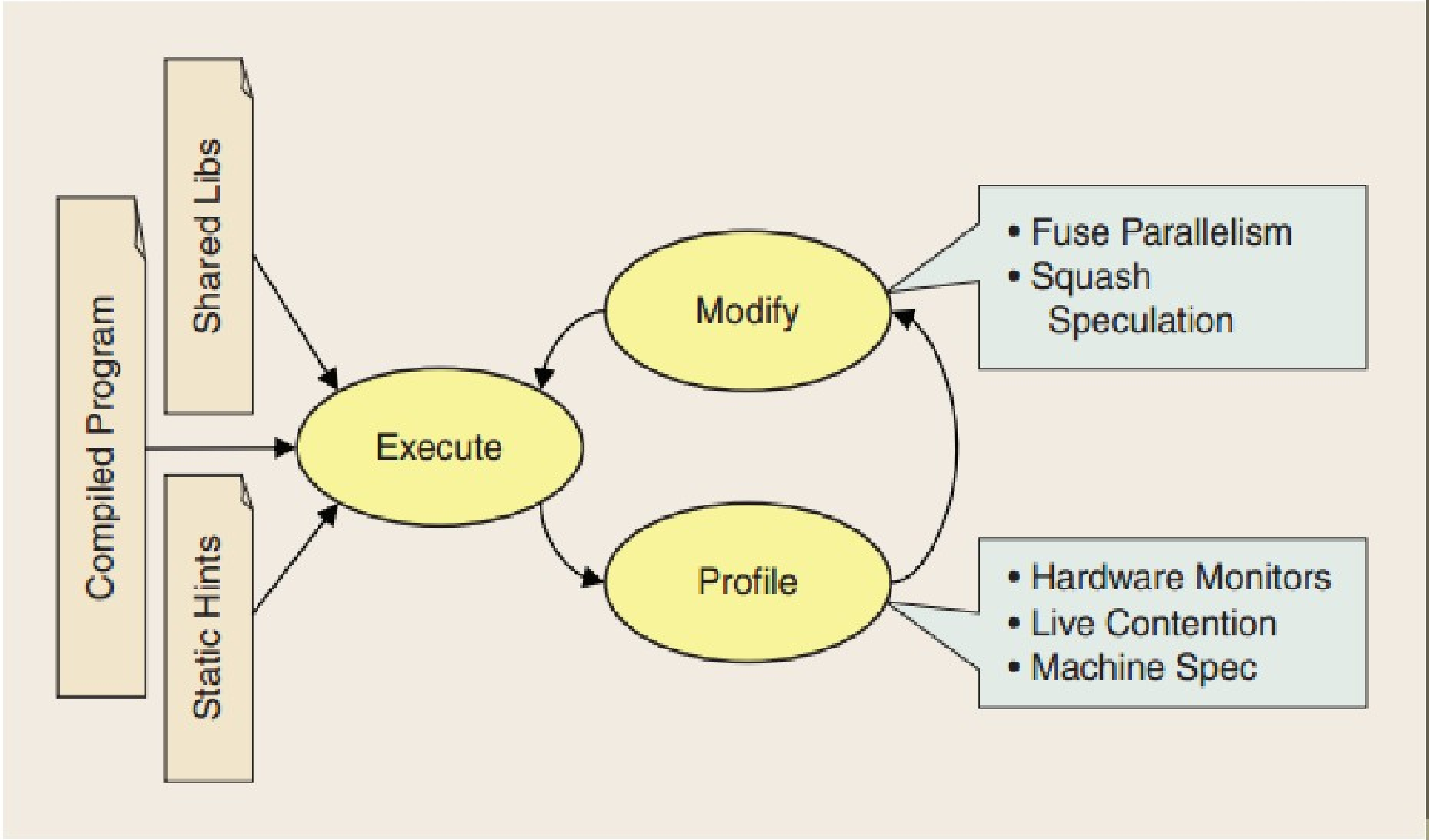
$\uparrow$ cache hit rate and performance

RUN-TIME MANAGEMENT AND OPTIMIZATION

- THE DIFFICULTY AND COMPLEXITY OF PROGRAMMING FOR MULTICORES INEVITABLY PUSHES PROGRAMMERS TO RELY MORE HEAVILY ON TOOLS, SUCH AS COMPILERS AND RUN-TIME OPTIMIZERS.

- Hardware performs various optimization and resulting code may then become tightly tied to the underlying system, limiting the true “portability” of the program.

- Performance can suffer when unexpected run-time events (such as cache misses) or resource contention occur.



[FIG4] A run-time adaptation engine continuously profiles and modifies a program as it runs. Profile information can come from the hardware, the operating system, the compiler (via statically inserted hints), or the application itself.

REFERENCES

http://www.cs.virginia.edu/kim/docs/ieeespm09.pdf

Videos –

http://www.youtube.com/watch?v=PqM10gv-qLM

THANK YOU