Speculative Parallelization

- Technology’s only constant is CHANGE

Devarshi Ghoshal

Sreesudhan

Agenda

- Moore's law
- What is speculation ?
- What is parallelization ?
- Amdahl's law
- Communication between parallely executing regions
- Existing speculative systems
- Formal programming model for speculative parallelization
- Comparison of results

Moore's law

- The number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every 18 - 24 months
- Performance of processors also doubled every 18 – 24 months

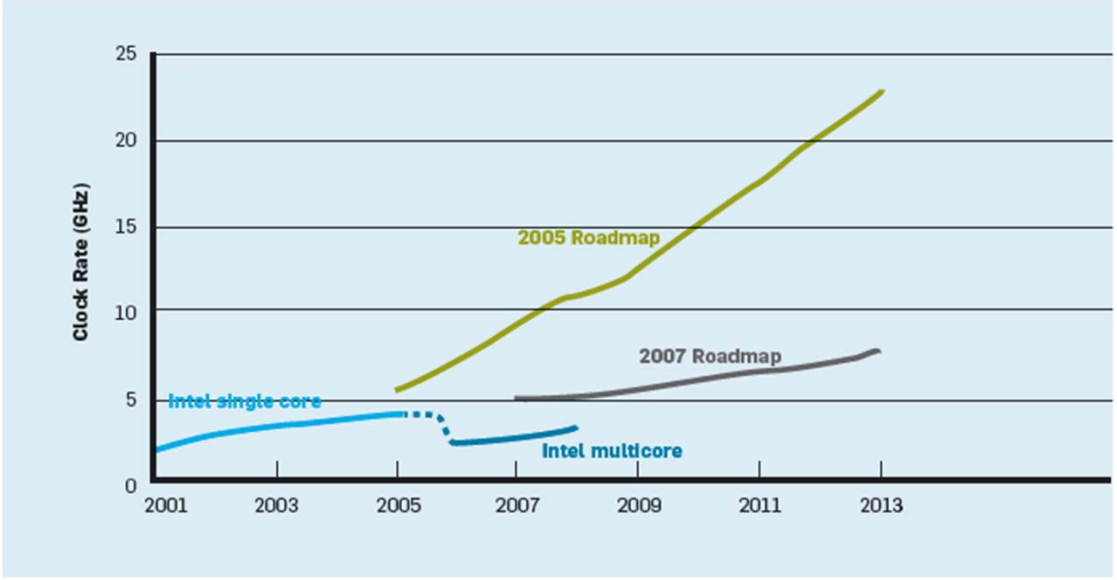
Single core processor

- Increase in clock frequency
- Increase in Instructions per cycle (IPC)
- Optimization techniques
  - Multiple instruction issue
  - Deep pipelines
  - Out of order execution
  - Prefetching
  - Speculative execution

Speculative Execution

- Basic concept of Speculative Execution
  - The execution of code the results of which may not be required
- Statements and Definitions in a Program
  - Segments which must be run and mandatory
    - Does not benefit from speculative execution
  - Segments which do not need to be run because they are irrelevant
    - Can be discarded without execution
  - Segments which cannot be proven to be in either of the two groups
    - Target of speculative execution as its members can be run concurrently with mandatory computations

Projection Vs Reality



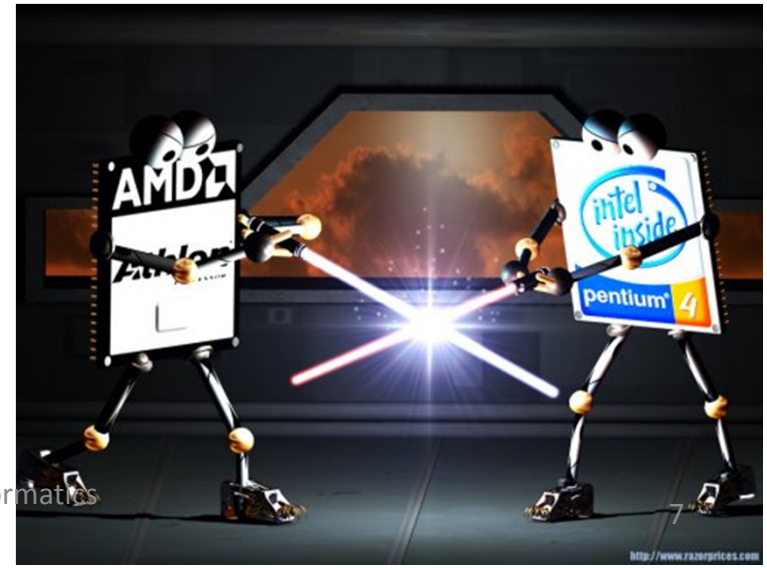
Microprocessor clock rates of Intel products Vs projects from the international roadmap for semiconductors in 2005 and 2007

Former Intel CEO's famous quote in 2005

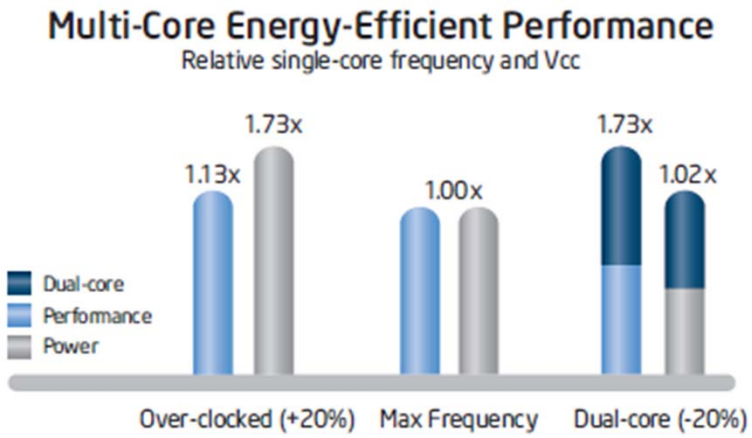
"The future of GRIN is multi core"

"If you want your computer or your entertainment device to do more for you, then you want more processing power and more communications capability. And dual-core and multi-core does that"

- Craig Barrett



Power and Frequency relation for dual-core processor



Here we add a second core on the under-clocked example. This results in a dual-core processor that at 20 percent reduced clock frequency effectively delivers 73 percent more performance while using approximately the same power as a single-core processor at maximum frequency.

Parallelism

- Single core processor
  - mutex, semaphore, wait – signal or broadcast (Multi threading - Process abstraction)
  - pipes, shared memory, sockets (Multi processing - Resource abstraction)
- Multi core processor
  - OpenMP (Shared memory)
- Multi processors
  - OpenMPI, MPICH (Distributed memory)

Speculative parallelization

- Hybrid model that speculatively executes regions of code parallely on available resources
- Utilizes available resources completely to improve performance
- Reduces execution time of application (most of the time)
- May take more time than normal in worst case

Existing Speculative Parallelizing Systems

- FastTrack
- FastForward
- Software BOP

Amdahl's law

- Used to find Speedup for some enhancement

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

- Fraction(enhanced) - The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement
- Speedup(enhanced) - The improvement gained by the enhanced execution mode; that is, how much faster the task would run if the enhanced mode were used for the entire program

Fast Track

- Creates dual track regions which involves code that can be run speculatively
- Runs unoptimized code parallely (against sequential version) on multiple processors
- Checks correctness after sequential version is executed
- Proceeds with speculative version if results are correct / sequential version otherwise

Semantics for loop

```
while (...) {
  ...
  if (FastTrack ()){
   /* unsafely */
   /* optimized */
   fast_fortuitous();
  }
  else {
   /* safe code */
   safe_sequential();
  }
  EndDualTrack();
  ...
}
```

Unsafe loop optimization using fast track. Iterations of fast_fortuitous() will execute sequentially. Iterations of safe_sequential() will execute in parallel with one another, checking the correctness of the fast iterations.

Semantics for functions

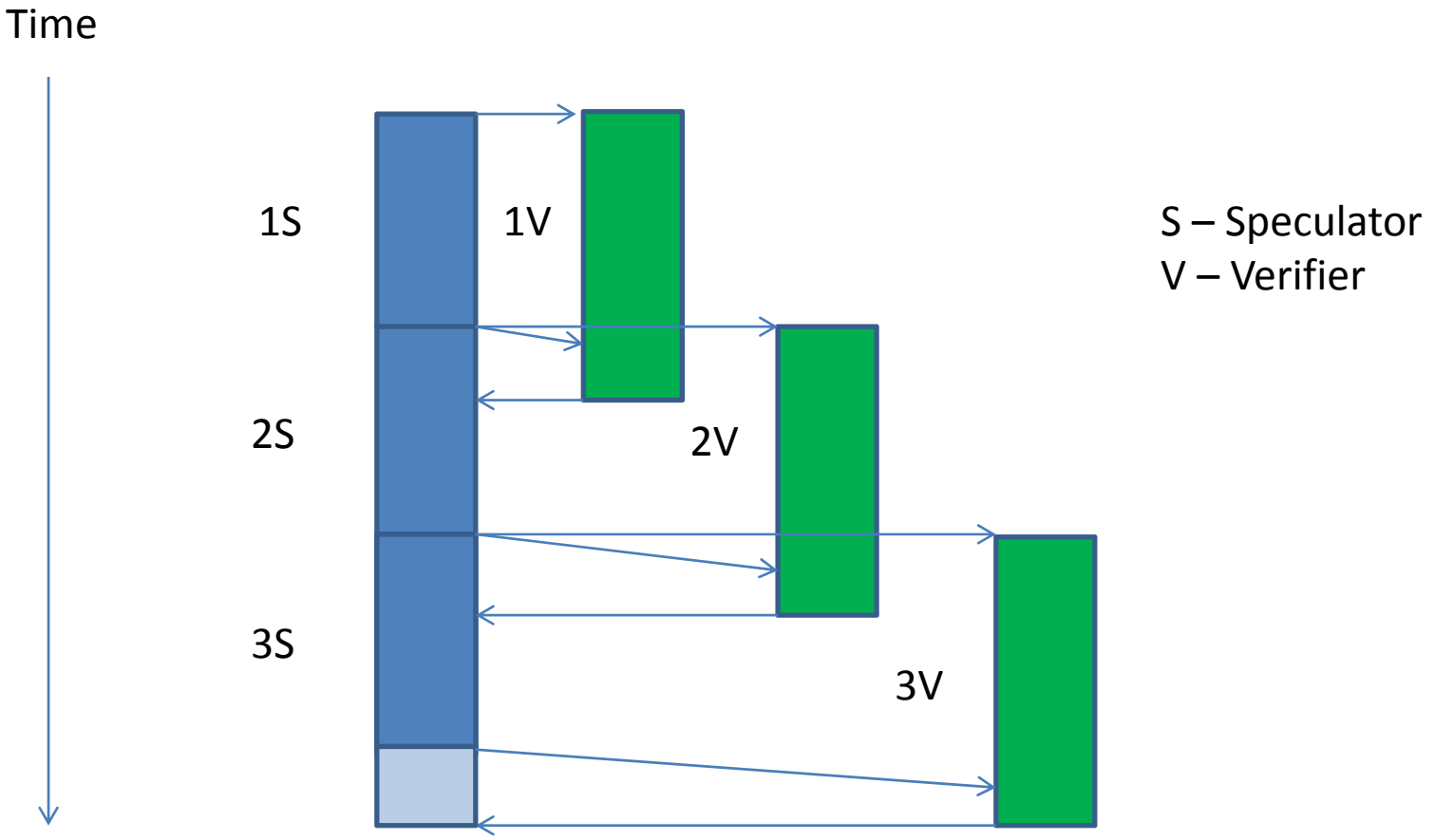
```
...
if (FastTrack ())
   /* optimized */
   fast_step_1();
else
   /* safe code */
   step_1();
...
if (FastTrack ())
   /* optimized */
   fast_step_2();
else
   /* safe code */
   step_2();
```

Unsafe function optimization using fast track. Routines fast_step_2 and step_2 can start as soon as fast_step_1 completes. They are likely to run in parallel with step_1.

System design

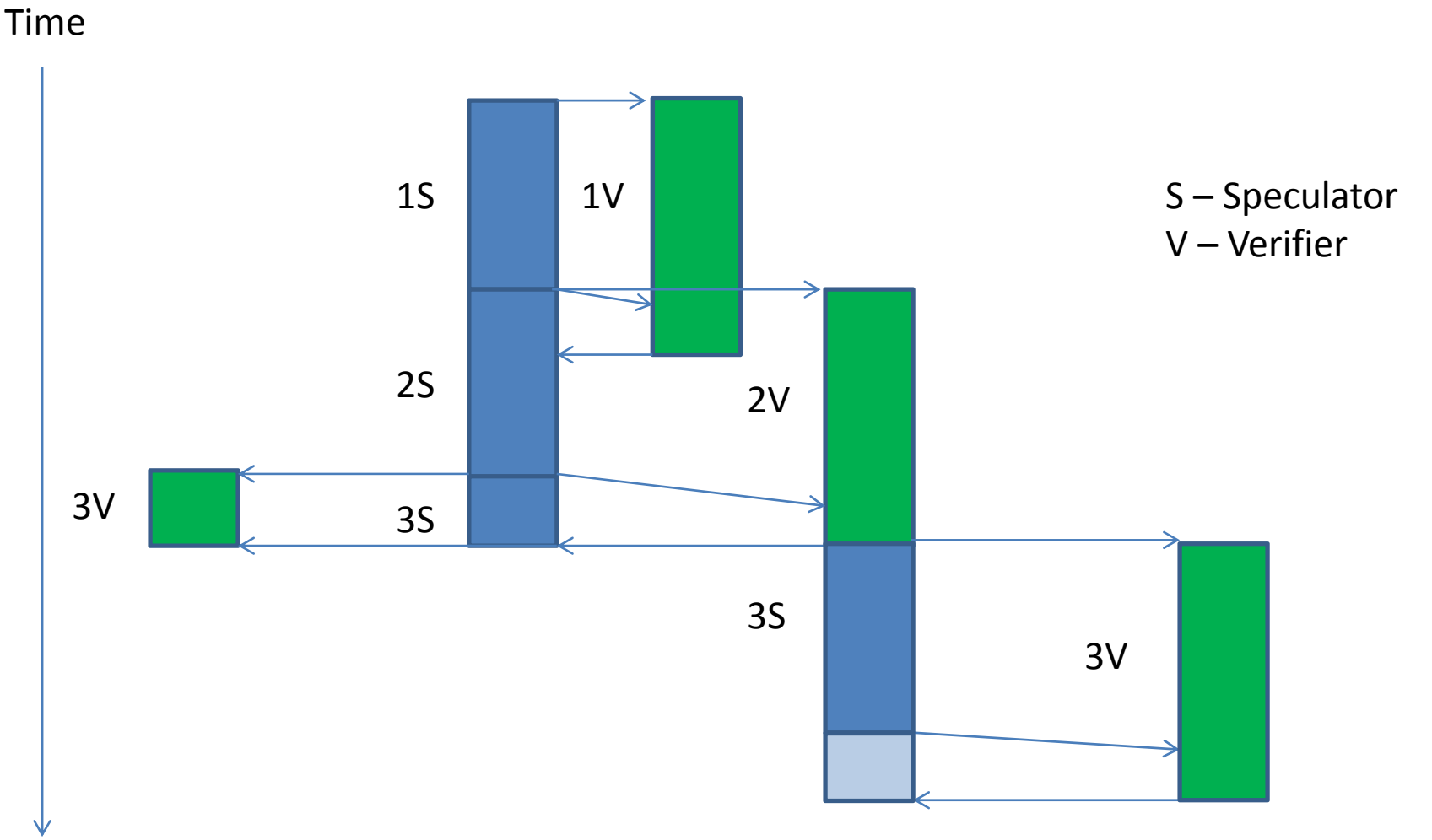
- Compiler support
  - Records changes made by both dual track regions
  - Compiler's inherent support for stack variables
  - Copy on write + access map for global & heap
- Run time support
  - Transfer pages of modified data using shared pipe
  - Compare memory state at the end of dual track region

Code flow



Case 1: All speculations are correct

Code flow



Case 2: Result of 2S is wrong

System limitations

- Waste of system resources including electric power
- Cannot create dual track if that piece of code involves interrupts, network interaction
- Limitation on system memory
- Special care for all Program termination points inside speculative region

Fast Track – Contd

- Unsafe program optimization techniques
  - Memoization
    - Store results of previous execution and use it for speculation
  - Manual program tuning
    - Programmer identifies speculative regions and creates dual track regions

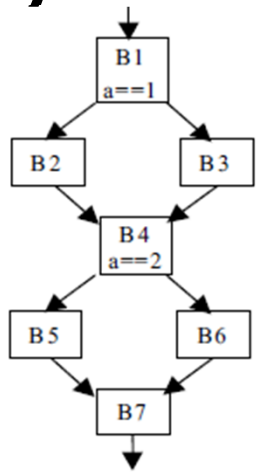
Fast Forward

- Compiler + speculative multi threaded support
- Compiler figures out hot spot for branches which are regions of code that are executed with high probability
- Multiple basic blocks are transformed to form one basic block containing all hot spots
- New basic block is optimized during compilation

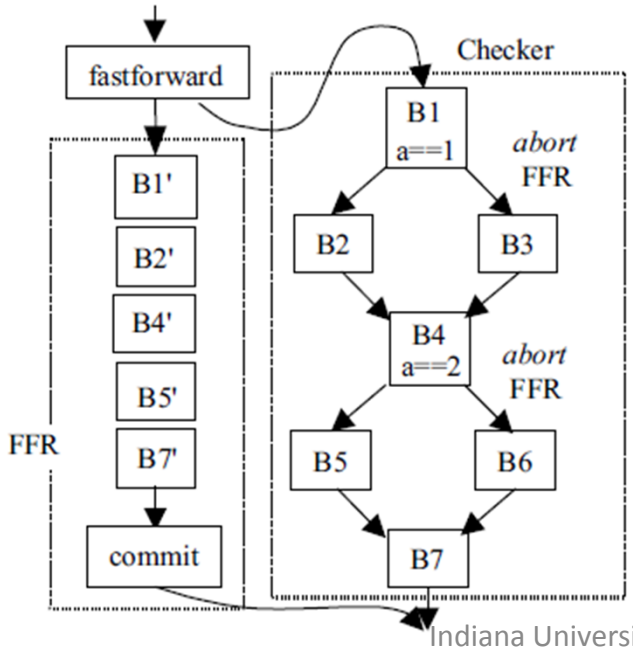
System design

```
B1
if (a !=1)
    B2
else
    B3  /* cold */
B4
if (a!=2)
    B5
else
    B6  /* cold */
B7
```

(a)

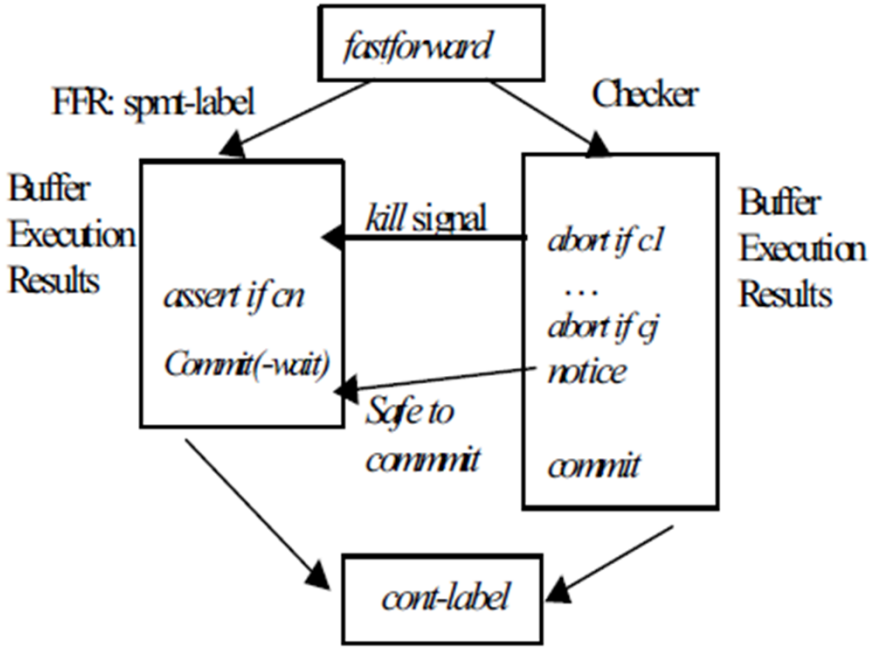


(b)



(a) Source program
(b) Control flow graph
(c) FFR region

Fast forward region and checker



FFR region terminates itself if assert becomes true / gets abort from checker

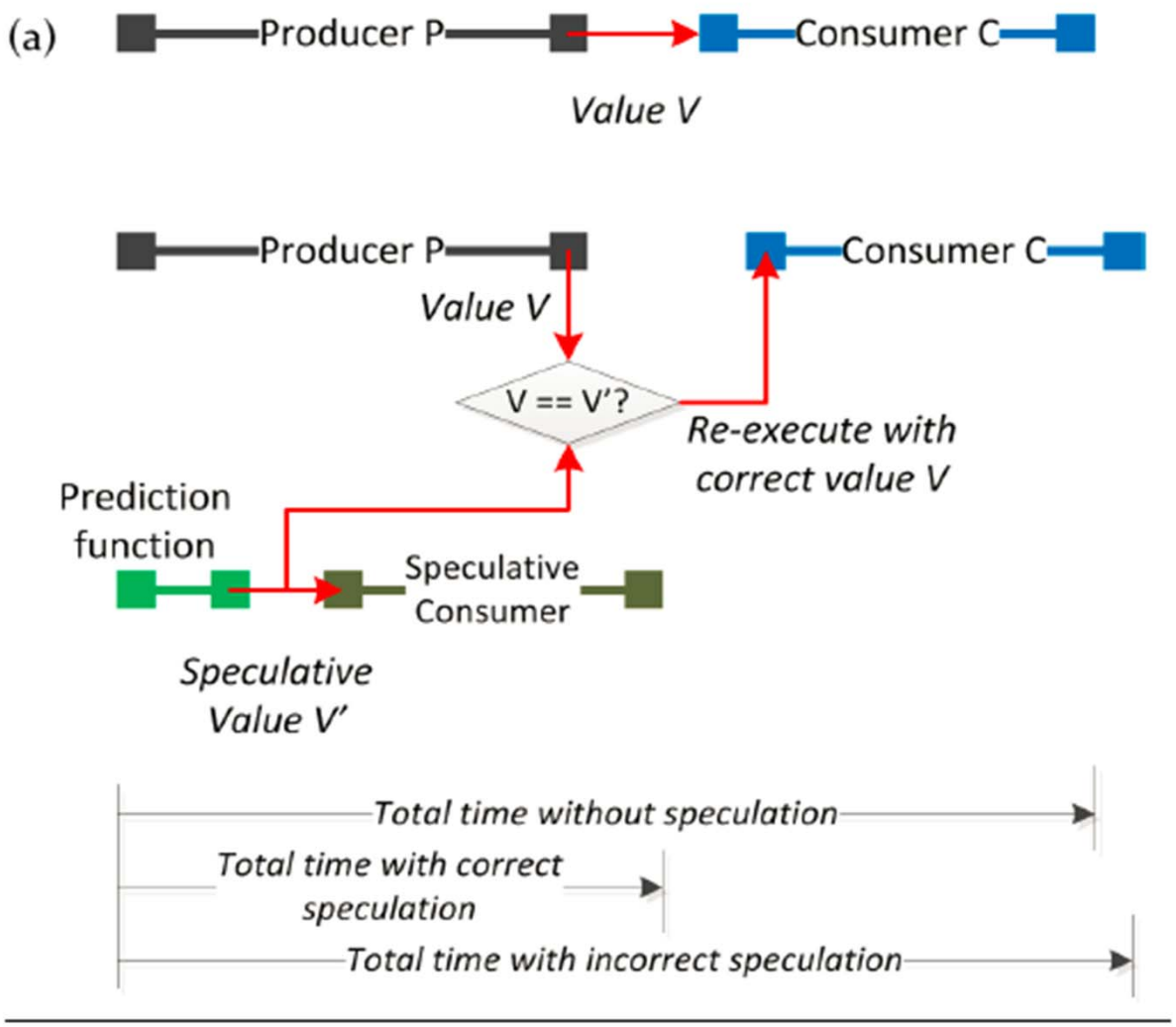
System limitations

- FFR has to wait until checker executes all abort instructions
- FFR and checker threads has to terminate as soon as possible to avoid unnecessary computation

Formal Language for Speculative Parallel Algorithms

- Inherently Sequential Algorithms
  - Lexical Analysis
  - Huffman Decoding
- Value Speculation
- Two new language constructs:
  - Speculative composition
  - Speculative iteration

Value Speculation



Syntax and Semantics of Speculate

- Fold expressions
  - Model a simple form of iteration where each iteration depends on the value computed in the previous iteration
- Speculative application
  - (spec p g c) : a producer p, a predictor g, a consumer c
- Speculative fold
  - (specfold f g l u) : represents the loop body, g is a prediction function in the range l..u, f is the fold expression
  - Executes all iterations of the loop in parallel using the predicted value and takes corrective action when prediction fails

Safe Speculation without Rollback

- Correctness Criterion
  - Final Equivalence
  - Dependence Equivalence
- Safety Criterion
  - Data races
  - Heap updates in case of misprediction
- Termination Guarantees
  - Speculation-validation step waits for both producer and predictor to complete execution
  - If validation step detects misprediction, it attempts to cancel the speculative consumer

Speculative Library

```
1 public class Speculation {
2   public static void Apply<T>(
3     Func(T) producer,
4     Func(T) predictor,
5     void Action(T) consumer)
6
7   public enum ValidationMode { Seq, Par };
8
9   public static void Iterate<T>(
10    int low, int high,
11    Func(int, T, T) loopBody ,
12    Func(int, T) predictor,
13    ValidationType val /* optional */)
14
15  public static void Iterate<T, U>(
16    int low, int high,
17    Func(U) initializer,
18    Func(int, U, T) loopBody,
19    Func(int, T) predictor,
20    Action(int, U) finalizer,
21    ValidationMode val)
22 }
```

Speculative Parallelization of Loops

- Main Thread- maintains the non-speculative state of the computation
- Multiple Parallel Threads- execute parts of the computation using speculatively-read operand values from non-speculative state
- State Separation
- Copy or Discard Mechanism

Speculative Parallel Execution Model

- The shared memory space is divided into three partitions:
  - D: Non-speculative State
  - P: Parallel or Speculative State
  - C: Coordinating State
- Misspeculation- Detection and Recovery
  - Version Numbers for Variables in D State Memory—C state of the main thread
  - Mapping Table for Variables in P State Memory—C state of a parallel thread

Loop Sections

- Dividing the Loop Iteration into three sections:
  - The prologue
  - The speculative body
  - The epilogue
- Main Thread non-speculatively executes the prologues and epilogues
- The Parallel Threads are created to speculatively execute the bodies of the iterations on separate cores

Software Behavior Oriented Parallelization (BOP)

- Programmable software speculation
  - Program parallelized based on “partial” information about program behavior
  - User or analysis tool marks “possibly” parallel regions
  - Runtime system executes these regions speculatively
- Critical-path minimization
- Value-based correctness checking
- No change to the underlying hardware or operating system

BOP Overview

- Issues addressed
  - Unknown data access and control flow make applications difficult to parallelize
  - Input-dependent behavior where both the degree and the granularity of parallelism are not predictable
- Design
  - Possibly Parallel Regions (PPR)
    - Marking the start and end of the region with matching markers: BeginPPR(p) and EndPPR(p)
  - Protects the entire address space by dividing it into possible shared and privatizable subsets

Possibly Parallel Regions (PPR)

- At a start marker, BOP forks a process that jumps to the matching end marker and speculatively executes from there
- At any point t, the next PPR instance starts from the first start marker operation BeginPPR(p) after t and then ends at the first end marker operation EndPPR(p) after the BeginPPR(p).
- Example:
  - A program, from the start t_0, executes the markers six times from t_1 to t_6 as follows:

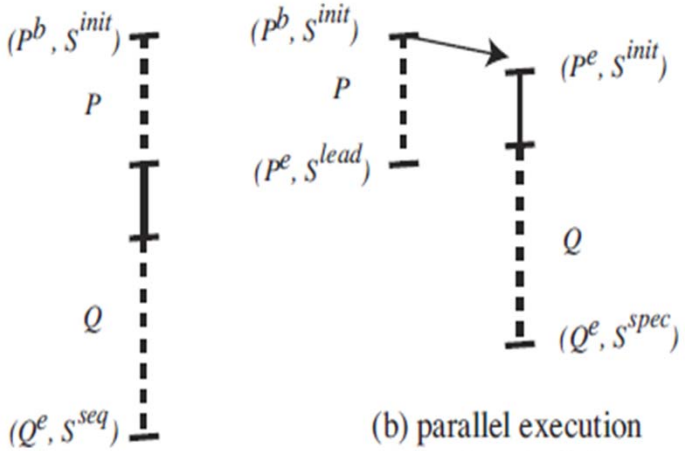
t_0	t_1	t_2	t_3	t_4	t_5	t_6
	m^b_P	m^b_P	m^e_P	m^b_Q	m^e_P	m^e_Q

  - Two dynamic PPR instances are from t_1 to t_3 and from t_4 to t_6, which will be run in parallel. The other fragments of the execution will be run sequentially, although the part from t_3 to t_4 is also speculative
- PPR markers can be inserted anywhere in a program and executed in any order at run-time
  - The system tolerates incorrect marking of parallelism
- The markers are programmable hints
  - The quality of hints affects the parallelism but not the correctness nor the worst-case performance

Parallel Ensemble

- The execution starts as the “lead” process
  - continues to execute the program non-speculatively until the program exits
- Uses concurrent executions to hide the speculation overhead off the critical path
  - determines the worst-case performance where all speculation fails and the program runs sequentially.
- At a (pre-specified) speculation depth k, up to k processes are used to execute the next k PPR instances.
  - for a machine with p available processors, the speculation depth is set to p−1 to make the full use of the CPU resource

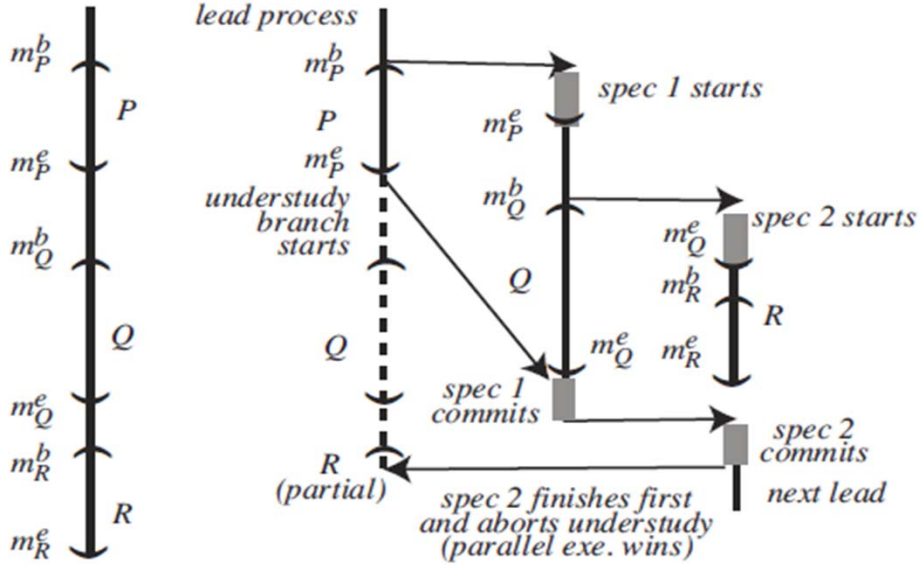
System Design



(a) sequential execution

(b) parallel execution (*lead* on the left, *spec* on the right)

States of Sequential and Parallel Execution



(a) Sequential execution of PPR instances *P*, *Q*, and *R* and their start and end markers.

(b) A successful parallel execution, with *lead* on the left, *spec* 1 and 2 on the right. Speculation starts by jumping from the start to the end marker. It commits when reaching another end marker.

Execution Flow

Table 1. *BOP* actions for unexpected behavior

behavior	prog. exit or error	unexpected *PPR* markers
lead	exit	continue
understudy	exit	continue
spec(s)	abort speculation	continue

State Isolation

- Thread-based systems
  - Weak isolation
  - The updates of one thread are visible to other threads
- BOP
  - Strong isolation
  - The intermediate results of the lead process are not made visible to speculation processes until the lead process finishes the first PPR
  - Strong isolation comes naturally with process-based protection

Data Validation

Table 2. Three types of data protection

type	shared data D_{shared}	checked data $D_{checked}$	(likely) private data $D_{private}$
protection	Not written by *lead* and read by *spec*	Value at *BeginPPR* is the same at *EndPPR* in *lead*. Concurrent read/write allowed.	no read before 1st write in *spec*. Concurrent read/write allowed.
granularity	page/element	element	element
needed support	compiler, profiler, run-time	compiler, profiler, run-time	compiler (run-time)
overhead on critical path	1 fault per mod. page copy-on-write	copy-on-write	copy-on-write

- Three disjoint address spaces
  - Shared
  - Checked
  - Private
- Three types of data protection
  - Page-based protection of shared data
  - Value-based checking
  - Likely private data

Limitations of BOP

- Not efficient- extra CPU, memory and energy usage)
- Speculative region cannot invoke general forms of I/O and other operations with unrecoverable side-effects

Comparative Study

	Fast Track	Fast Forward	BOP
Speculation Type	Process-based	Thread-based	Process-based
Overheads	High	Low	High
Speedup	2 – 4 times	1.2 times (average)	1.2 – 2.1 times
Speculation Depth	Multiple	Two	Multiple

References

- L. Rauchwerger and D. Padua, “The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization,” in Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI), 1995, pp. 218–232, http://doi.acm.org/10.1145/207110.207148

- L.-L. Chen and Y. Wu, “Aggressive compiler optimization and parallelization with thread-level speculation,” in International Conference on Parallel Processing (ICPP), 2003, http://doi.ieeecomputersociety.org/10.1109/ICPP.2003.1240629

- K. Kelsey, T. Bai, C. Ding, and C. Zhang, “Fast Track: A software system for speculative program optimization,” in Proceedings of the International Symposium on Code Generation and Optimization, 2009, pp. 157–168, http://doi.ieeecomputersociety.org/10.1109/CGO.2009.18

- M. F. Spear, K. Kelsey, T. Bai, L. Daless, M. L. Scott, C. Ding, and P. Wu, “Fastpath speculative parallelization,” in Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2009, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.157.4762

- C. Tian, M. Feng, V. Nagarajan, and R. Gupta, “Speculative parallelization of sequential loops on multicores,” International Journal of Parallel Programming, vol. 37, no. 5, pp. 508–535, Oct. 2009, http://dx.doi.org/10.1007/s10766-009-0111-z

- C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang, “Software behavior oriented parallelization,” in Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2007, pp. 223–234, http://doi.acm.org/10.1145/1250734.1250760

- P. Prabhu, G. Ramalingam, and K. Vaswani, “Safe programmable speculative parallelism,” in Proceedings of the ACM SIGPLAN 2010 International Conference on Programming Language Design and Implementation, 2010, http://doi.acm.org/10.1145/1806596.1806603

- D. Bruening, S. Devabhaktuni, and S. Amarasinghe, “Softspec: Software-based speculative parallelism,” in Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization, 2000, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.9885

Thank you