Optimistic Parallelization
of Communicating Sequential Processes

David F. Bacon
University of California, Berkeley
and IBM T.J. Watson Research Center

and

Robert E. Strom
IBM T.J. Watson Research Center

Third ACM SIGPLAN Symposium on
Principles and Practices of Parallel Programming
PPoPP 1991
Goal

Parallelization of

- distributed programs

- across run-time dependencies

- with predictable behavior
Example Problem

```c
/* S1 */ OK = Update(Item, Value);
/* S2 */ if OK
    Write(File,"Did it");
```
Outline

• Motivation

• Parallelizing a Single Process

• Parallelizing Communicating Processes

• Example Applications

• Conclusions
Motivation: Latency

Latency Remains Constant

- Round-trip to New York = 31 ms
- Round-trip across a 4 ft machine = 8 ns
Superscalar and VLIW architectures:

• already using optimistic techniques
  
  – Hardware: RS/6000.
  
  – Software: Trace Scheduling Compilers.

• approaching limits of static scheduling:
  
  – 1/7 instructions conditional branch.
  
  – Studies claim speedup $\leq (5, 10)$

“There is widespread opinion that we are fast approaching the physical limit in speeds for computers”
More Motivation: Distribution

Distributed programs often can’t be analyzed statically:

- dynamic binding.

- different machine types, operating systems.

- security issues.
Model and Assumptions

Our Distributed Computing Model: CSP

- single-threaded processes
- only communication is via message-passing

Mechanisms Assumed:

- checkpointing state
- rollback to a checkpointed state

Note: checkpointing can use message logging.
Parallelizing a Single Process

```c
/* S1 */ OK = Update(Item, Value);
/* S2 */ if OK
    Write(File, "Did it");
```

If OK = False, there is a VALUE FAULT.
Dependencies (Side Effects)

Flow Dependence

- \( \text{readset}(S_2) \cap \text{writeset}(S_1) \neq \emptyset \)
- constitutes a value fault
- abort \( S_2 \) and re-execute
 Dependencies (Side Effects)

Anti-Dependence

- \text{writeset}(S_2) \cap \text{readset}(S_1) \neq \emptyset

- could abort $S_1$ and $S_2$

- solved by copying potentially shared state
Dependencies (Side Effects)

Output Dependence

- \( \text{writeset}(S_1) \cap \text{writeset}(S_2) \neq \emptyset \)

- same solutions as for anti-dependencies
Parallelizing Multiple Processes

Client          Database          Fileserver

C1               C2               C3

R1               R2               R3
• Propagation of dependencies

• Detection and undo of failed parallelization

• Detection of successful parallelization
Failed parallelizations generate a cycle in the “happens before” relation.

Cycle must include the “backward arrow”.
Solution: Dependency Tracking
Dependency Tracking (Time Fault)

A time fault exists at end of client thread:

\[ \{x_1\} \rightarrow \{x_1\} \]
• Time fault results in an abort message.
• All threads roll back to the point before aborted commit guard was acquired.
Applications

\[ y = \text{fast-}f(x); \]
if not slow-check(x,y) {
    \[ y = \text{slow-}f(x); \]
}

e tc;

\[ \downarrow \]

\[ y = \text{fast-}f(x); \]

\[ \text{etc} \quad \| \quad \text{if not slow-check(x,y) } \{
    \text{ABORT etc;}
    \[ y = \text{slow-}f(x); \]
    \text{etc;}
\} \]
Remote Write

send Update(Item, Value);
if receive(status) = BAD {
    exit();
}

etc;

↓

send Update(Item, Value);

etc      ||      if receive(status) = BAD {
             ABORT etc;
             exit();
         }

*
Caching

\[ y = \text{local-read}(x); \]
\[ y2 = \text{remote-read}(x); \]
if y \neq y2 {
    y = y2;
}

etc;

\[ \downarrow \]

\[ y = \text{local-read}(x); \]

\[ \text{etc} \quad \text{||} 
\]
\[ y2 = \text{remote-read}(x); \]
if y \neq y2 {
    ABORT etc;
    y = y2;
    etc;
}

24
Other Examples

- Optimistic Recovery

- Time Warp Distributed Simulation

- Bisection method for Eigenvalues in an interval

- Detection of data structure corruption concurrent with usage
Single Process Parallelization Research

Hardware (Knight, MIT):

- cache detects conflicts

- special cache for uncommitted values

- roll back by reading main memory

Software (Katz, Stanford):

- compiler instruments potentially conflicting references

- instrumented code saves and restores values

*
Implementation
Conclusions

Methods capable of handling run-time dependencies will soon be needed:

• limits on basic block sizes.

• ever-increasing relative latencies.

Optimistic Parallelization:

• is applicable to a wide variety of problems.

• is efficient enough to extract substantial coarse-grained parallelism.

• provides a uniform framework for a large class of optimistic transformations.
Failed Parallelization

*
Hairy Example: Successful

*
Hairy Example: Failed

*