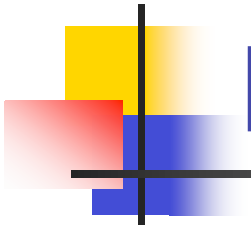
Parallelization of Utility Programs Based on Behavior Phase Analysis

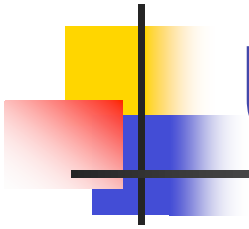
Xipeng Shen Chen Ding

Department of Computer Science
University of Rochester



Motivation

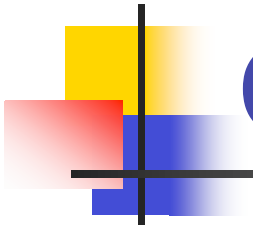
- Multi-core is coming to personal computers
- Many programs, especially those run on past personal computers, are sequential programs
- Automatic parallelization is the path of least resistance



Utility Programs

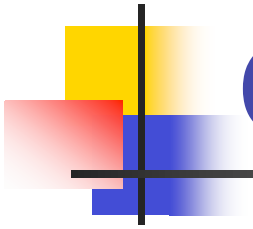
- A class of dynamic programs which take a group of requests and serve them one by one
- Examples
  - Compilers, interpreters, compressions, transcoding utilities, ...
    - GNU C compiler (Gcc)

      The compilation of a function is a phase



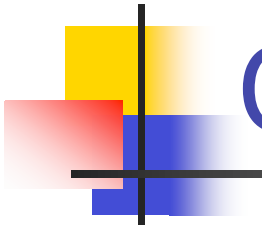
Challenges

- Dynamic data (access)
  - Dynamically allocated data structures
  - One or more levels of indirections
- Complex control flow
  - Input-dependent execution paths
  - Many (recursive) function calls
- More difficult to analyze and parallelize than scientific programs



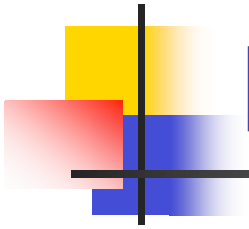
Opportunities

- Different phase instances operate on different data, thus have few data dependences between them
- Recently we found a way to detect the phase boundaries
- Can we automatically parallelize those programs at the phase level?



Overview

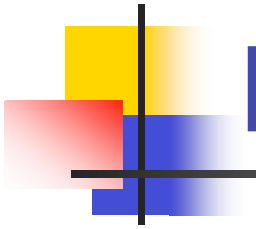
- Objective: to preliminarily check the feasibility of parallelizing utility programs at phase level without special hardware support
- Technology
  - Phase detection
  - Dependence detection
  - Program transformation
- Evaluation
- Summary



Behavior Phase Detection

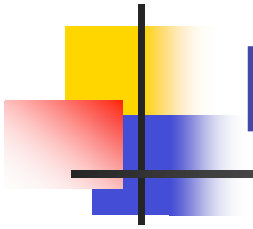
- Key idea: active profiling
  - Use regular input to trigger repetitive behavior
  - Filtering dynamic basic block trace based on frequency and recurring distance
  - Use real input to verify phase boundaries

*Refer to “Shen et. al., TR 848, CS, U of Rochester, 2004”



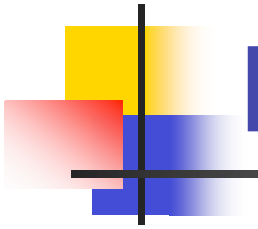
Phase-based Parallelization

- Process-based parallelization
  - Separate address space
- Each process executes one or a group of phase instances



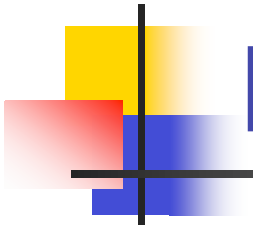
Phase-Dependence Detection

- Trace memory accesses in profiling runs
- Detect different kinds of dependences
  - anti- and output dependences can be ignored because of separate address space
  - Classify flow dependences into removable and non-removable types



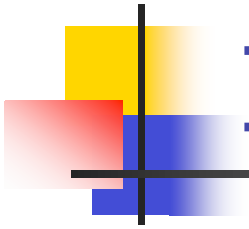
Flow Dependence

- Removable flow dependence
  - Memory reuses
  - Implicit initialization
  - Byte operations



Memory Reuses

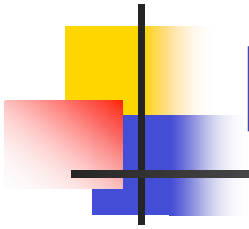
Two objects are allocated to the same memory location in different part of the execution.



Implicit Initialization

```
NODE* xlevel(NODE* expr){
    if (++xltrace<TDEPTH){
      ...
    }
    - - xltrace;
}
```

*code fragments from SPEC2K/LI

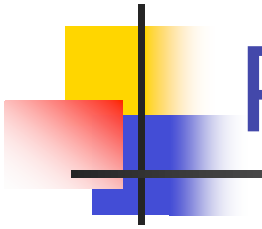


Byte Operation

```
char * buf;
...
buf[i] = 0; // byte operation
```

```
lda s4, -28416(gp) //  load array base address
addq s4, s0, s4 // shift to the target array element
ldq u v0, 0(s4) //  load a quadword from the current element
mskbl v0, s4, v0 // set the target byte to 0 by masking
stq u v0, 0(s4) // store the new quadword to the array
```

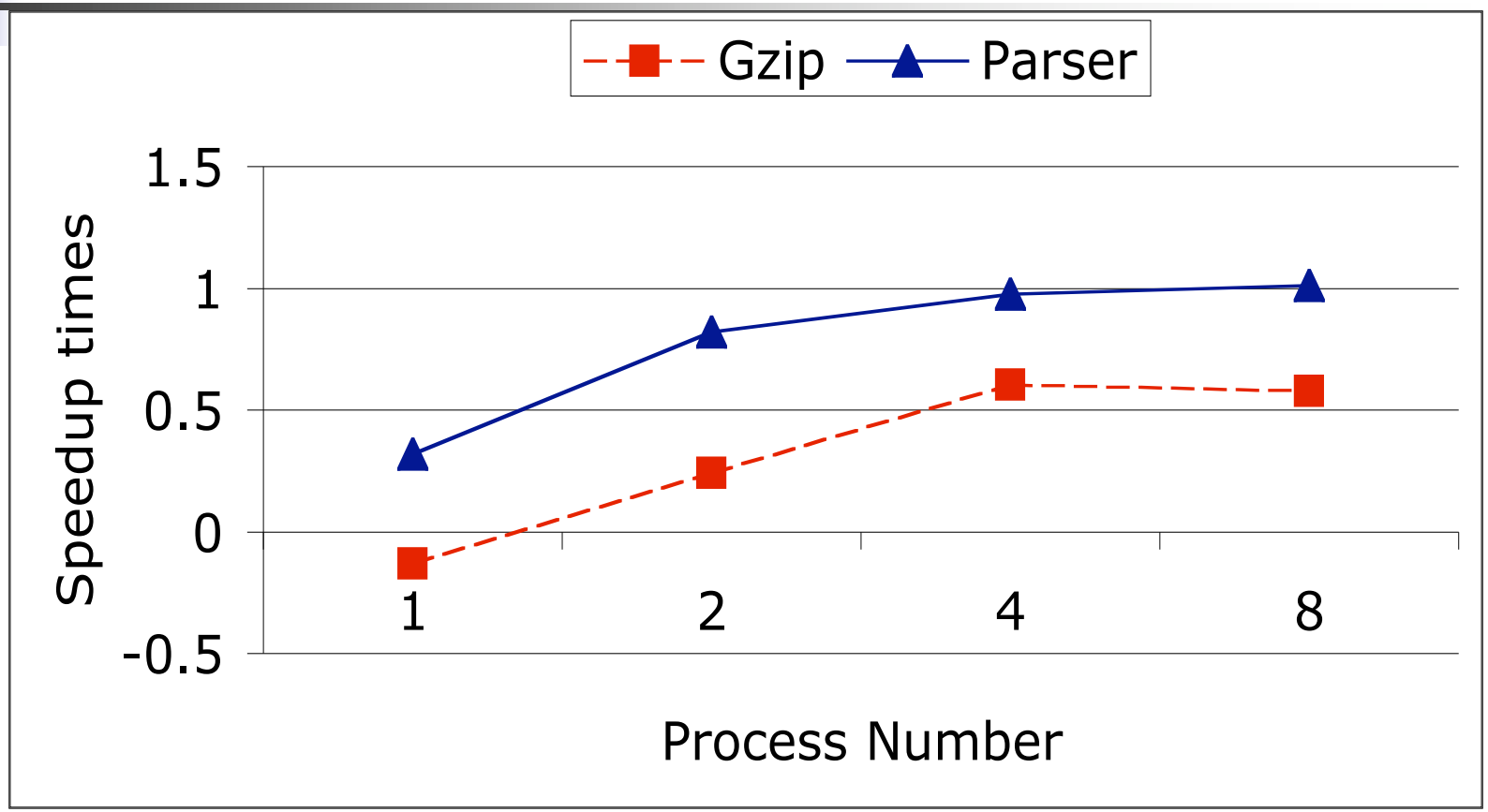
*code fragments from SPEC2K/Parser



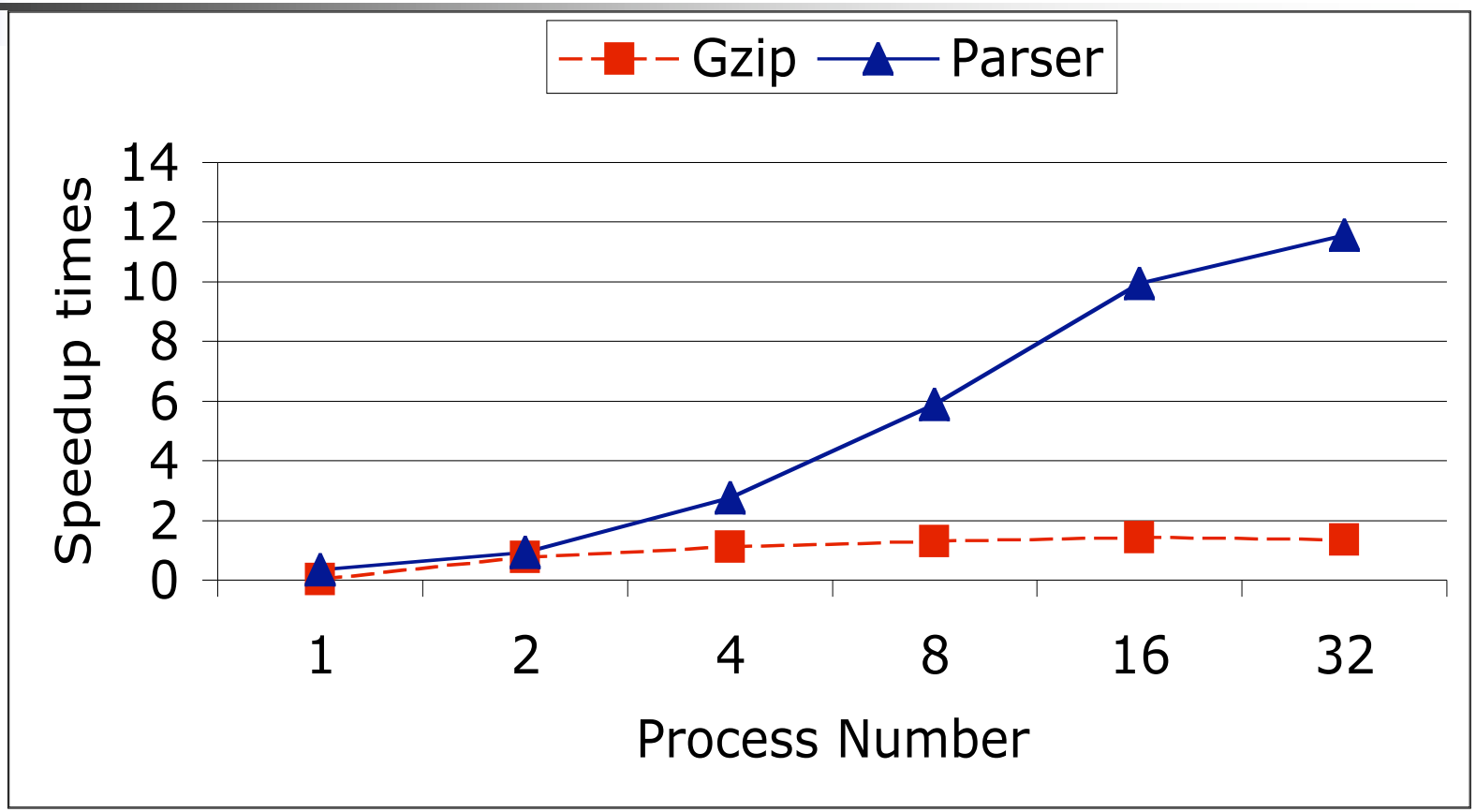
Program Transformation

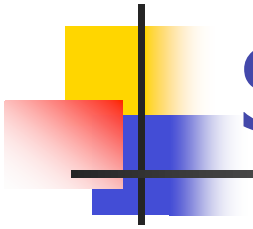
- We parallelize programs by hand at phase boundaries based on the information provided by the automatic tool
- A fully automatic tool would include automatic parallelization with run-time support to guarantee correctness and rollback when necessary
  - Currently being studied

Evaluation (4-CPU Xeon 2GHz)



Evaluation (16-CPU Sunfire Sparc V9 1.2 GHz)





Summary

- A preliminary exploration on the coarse-grain parallelization of utility programs based on behavior phases
- Fully automatic system remains our future work

The End

Thanks!