Techniques for Efficient Placement of Synchronisation Primitives

Alexandru Nicolau, Guangqiang Li, Arun Kejariwal

University of California, Irvine and Yahoo! Inc

PPoPP '9

Introduction

- Exploitation of hardware parallelism is key to making concurrent software available.
- One mechanism is to rely on auto parallelization by using *post* and *wait* primitives.
- Placing synchronisation primitives at their natural locations may not allow the best usage of Thread Level Parallelism.
- There are two principle areas to explore:
  1. Placing a *post* request as early as possible.
  2. Placing a *wait* request as late as possible.
- Since manually placing primitives is too expensive an option, automation becomes useful.
- This paper presents compile time techniques for efficient placement of synchronisation primitives.

Why Synchronize?

- Not all program loops are naturally of $\mathrm{DO\text{-}ALL}$ nature.
- Preserving correctness is important while exploiting parallelism.
- Placing wait statements as late as possible allows efficient parallelization.
- The paper aims to *percolate* post primitives as early as possible and the wait primitives as late as possible.
- There is a useful trade off that comes into the picture as far as parallelizing code consisting of dependencies.

Terminology

- Let N denote all nodes n_0,n_1, n_2... in a CFG.
- $\mathrm{PRED(n)}$ denotes predecessors, $\mathrm{SUCC(n)}$ denotes successors.
- Incoming edges are I_j, Outgoing edges are E_j at node n_j.
- No write conflicts are allowed within a particular node.
- The DAG of conditionals is assumed to be rooted.
- For a conditional x, $s_p(x)$ denotes the operations above it, $s_t(x)$ denotes the true branch, and $s_f(x)$, the false branch.
- Given a loop carried dependency between u and v, $u \rightarrow v$ denotes a dependency with u as the source and v as the sink.

The Techniques

- We have three primary questions to answer:
  1. Which operations are percolated in which directions?
  2. How is an operation percolated?
  3. How far up or down should an operation be percolated?
- The set of natural *post* and *wait* positions would be the operations we wish to percolate.
- Transformations are applied iteratively on adjacent CFG nodes, so that one transformation may expose further chances for code motion.
- Operations are classified as *Move-Op-Up* and *Move-Op-Down* based on which direction an operation at a node is percolated.

The Techniques: continued

- Moving up or down is subject to 3 assumptions:
  1. No conflict exists between the parent and child node.
  2. The operation does not kill a live node from where it is being moved.
  3. There is no write to a shared memory location unique to either the parent or the child node.
- Interestingly, both Move-Up and Move-Down together are needed to guarantee best placement.
- Conditional operations can be moved up or down subject to certain copy operations made on the added CFG nodes.
- A *Unify-Up* operation may be performed to move an operation from multiple copies of the same node to a common predecessor node.
- A corresponding *Unify-Down* transform also copies operations from predecessors to a common successor.
- The *Delete* transformation removes empty nodes.

Results

- Kernels have been extracted from SPEC CPU2000,CPU2006 and the Linux Kernel.
- The experimental setup used an Intel Pentium Processor,with 1025020 KB memory.
- Code was compiled using the Intel C++ compiler on an Ubuntu Platform.
- The maximum speed up achieved was 60.34 percent, when compared to vanilla synchronisation.

Summary and Future Work

- The paper presents a robust technique to place synchronisation primitives.
- Transformations on the CFG, coupled with operations to move instructions up and down iteratively result in well-parallelized code.
- Extending the analysis to consider write operations to global variables by assessing which threads are likely to alter its state can be one area of future exploration.
- Allowing for asynchronous code motion in languages like JavaScript to analyse the effect on global variables can be one more area to explore.
- Extending the analysis to allow for dynamic class loading in a language like Java can be another useful area to explore.

Thank You!