A Lightweight Approach to Compiling and Scheduling Highly Dynamic Parallel Programs

Ettore Speziale, Michele Tartara

Dipartimento di Elettronica ed Informazione (DEI), Politecnico di Milano, Milano, Italy

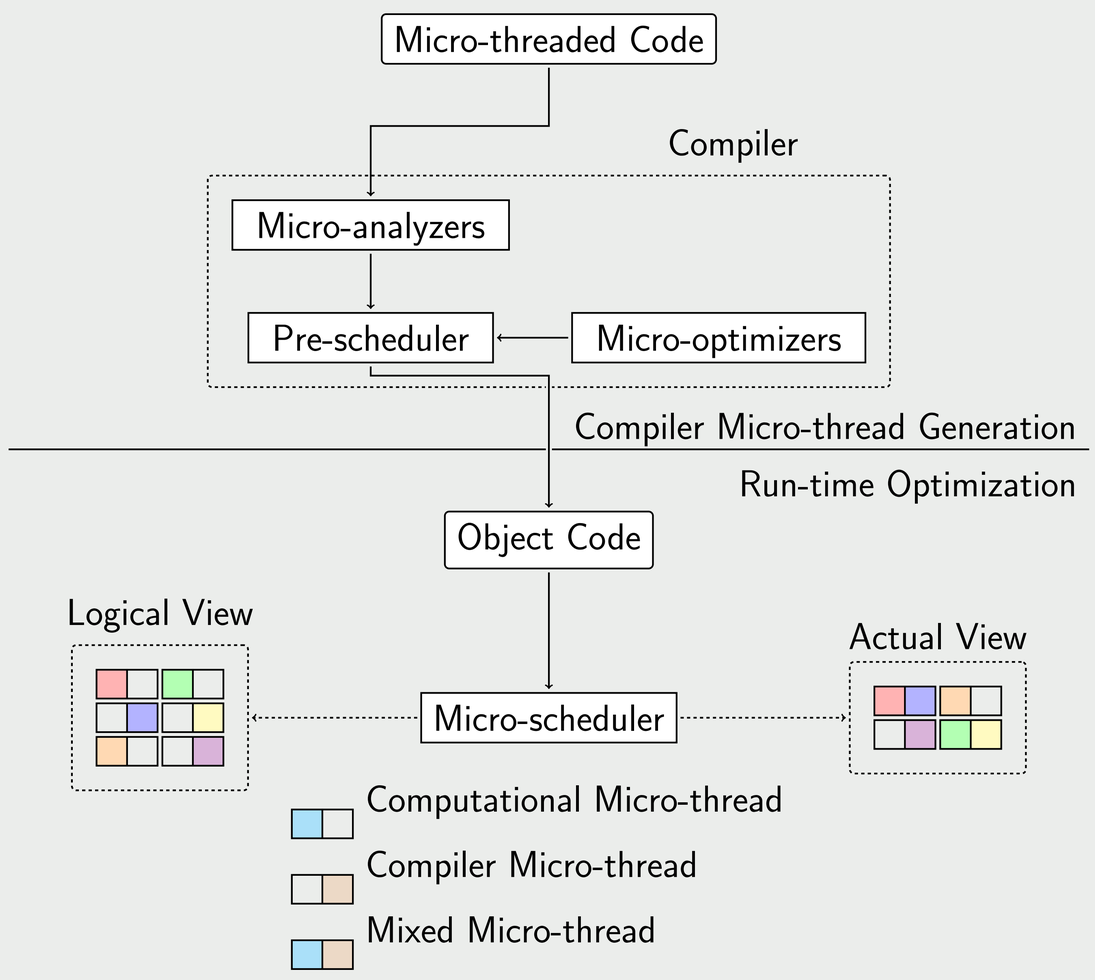
Problem Statement

- Optimizing highly dynamic parallel programs requires runtime info
  - program behavior depends on input data
  - wide spectrum of hardware configurations
  - the performance of statically compiled programs is not predictable
- Runtime libraries are useful – sometimes required – to manage parallel programs and to improve their performance
  - some information needed at compile-time are available at run-time
  - exploited by runtime libraries to make up for compiler deficiencies
  - increased use of computational resources at run-time
- A compiler-based approach is more appealing

Current Solutions

- Code Specialization
  - multiple version of code are statically generated at compile-time
  - the fittest one is selected at run-time
  - *pros:* run-time cost only given by choice between alternatives
  - *cons:* increasing alternatives improves effectiveness, but also code-size
- JIT Compilation
  - ad-hoc machine code compiled at run-time
  - *pros:* potentially optimal code is generated
  - *cons:* a full compiler is required; the actual code quality is limited by run-time constraints
- Self-modifying Code
  - modifies code on the fly, while it is being executed
  - *pros:* improves the performance of code without requiring a compiler
  - *cons:* makes code harder to write and to maintain

Proposed Approach

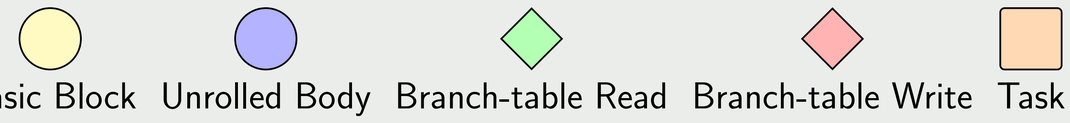


- At compile-time
  - determine which optimizations will be applicable only at run-time
  - select those with the highest expected profitability
  - inject the code that will apply the optimization at run-time
  - pre-scheduler prevents optimizing/optimizable code conflicts
- At run-time
  - optimizable code mixed with optimizing code
  - micro-scheduler triggers the execution of optimizing code
  - optimizing code shares the same execution-flow as optimizable code

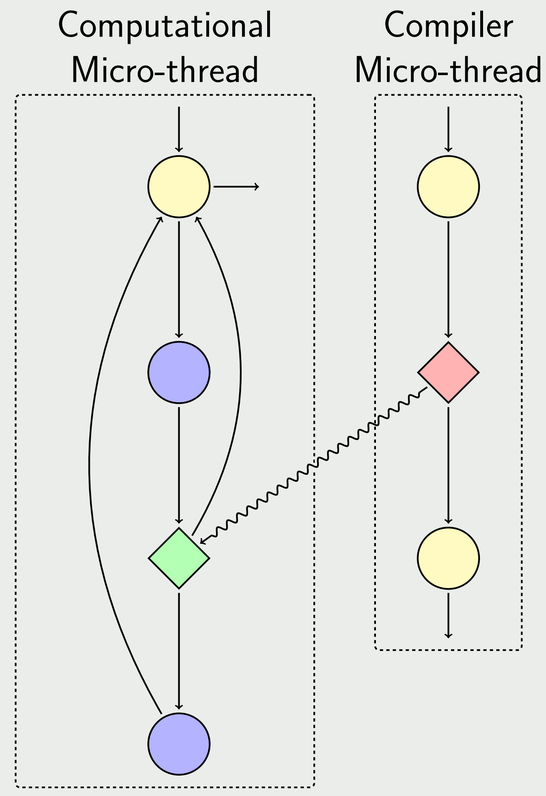
Challenges

- Computing expected profitability of optimizations
- Choosing optimization mechanism
  - branch tables
  - compiler-generated self-modifying code

Example: Legend

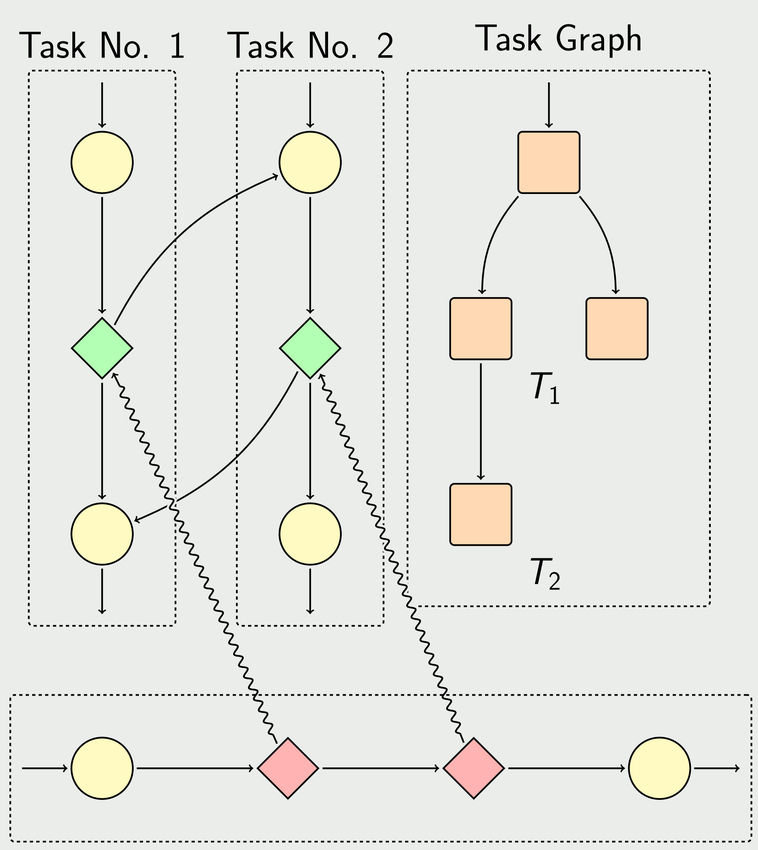


Example: Loop Unrolling



- Loop body unrolled multiple times
  - loop bodies separated by branch-table reads
  - actual unrolling factor controlled by optimizers through branch tables
  - micro-programmed architectures can benefit from the reduced number of branches by pursuing more aggressive micro-code schedules

Example: Task Fusion



- Dynamic task graphs require scheduler invocation after each task
- A branch-table read at the end of each task
  - allows jumping to the next ready task without invoking the scheduler
- Optimizers set the branch-table according to the actual task-graph
  - scheduler code shares the same execution-flow as task code

Conclusion and Future Work

- Dynamic techniques are essential to manage highly dynamic parallelism
  - static approaches lacks fundamental information
  - JIT compilation requires too much resources
- Our proposed approach
  - bridges the gap between static and dynamic techniques
  - splits optimization effort between compile-time and run-time
  - prepares optimizer-aware code at compile-time, lowering run-time cost
  - finalizes the optimization at run-time, exploiting then-available info

http://www.dei.polimi.it <name>.<surname>@mail.polimi.it