Compiler techniques for in-core Thread-Level Speculation

UNIVERSITY OF CAMBRIDGE
Department of Computer Science and Technology

Utpal Bora[1] Akshay Bhosale[1] Márton Erdős[1] Alexandra W. Chadwick[1] Yuxin Guo[1] Bob Lytton[2]
Giacomo Gabrielli[2] Timothy M. Jones[1]

[1]Computer Laboratory, University of Cambridge [2]Arm Ltd, Cambridge

Introduction

Automatic parallelization of loops is a challenging problem.

- Traditional multi-threading schemes incur substantial overheads (setup, communication, data-movement etc.) → useful for very large loop bodies.
- ILP schemes use the out-of-order window and provide diminishing returns in loops with large iterations → useful for very small loop bodies.

→ **This leaves medium-sized loop bodies unparallelized.**

Our research group is exploring a hardware-software codesign approach combining the benefits of Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP) into a modern Thread-Level Speculation (TLS) scheme. This poster covers compiler techniques targeting parallelism in medium granularity loops.

Background

We leverage three hint instructions to encode parallelism based on **Tapir** [2].

- A `detach` is the point at which a speculative thread can be forked.
- A `reattach` is the point of joining with the speculative thread. There can only be one `reattach` per `detach`.
- A `sync` marks the end of the speculative region, and all outstanding speculative threads are squashed. There can be more than one `sync` per `detach`, one for each loop exit.

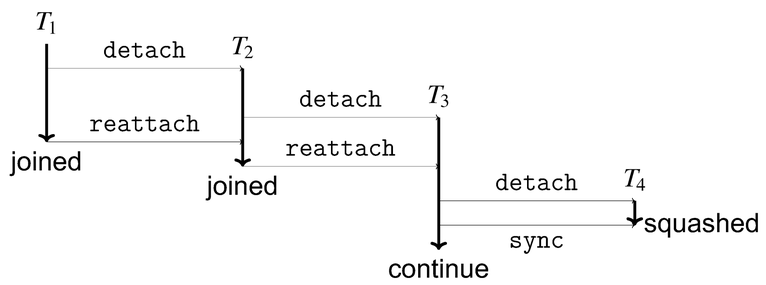

Figure 1. Fork-join execution of the threads. T_1 forks a new thread T_2 on `detach` hint. Threads T_1 and T_2 joins on `reattach` and speculative thread T_4 is squashed when T_3 encounters a `sync`.

A (speculatively parallel) **body** consists of the instructions, on all paths, between a `detach` and its corresponding `reattach`.

Highlights

Implemented an LLVM based automatic-parallelizing compiler for speculative execution.

- **Tapir** based parallel IR with `detach`, `reattach`, and `sync` instructions
- These are encoded as hint instructions
- **Serial Elision** property holds (hints can be replaced with **NO-OPs**)
- **Custom post-dominator** for program point equivalence checking
- **16% geomean speedup** on SPEC CPU 2006

Baseline Compiler

In the baseline scheme, we insert `detach` and `reattach` hints on program points that are executed exactly once in each iteration of the loop. The live registers at `detach` and `reattach` should be identical. Memory conflicts are handled in the microarchitecture and store-to-load forwarding is supported. However, in the event of misspeculation the speculative thread is squashed.

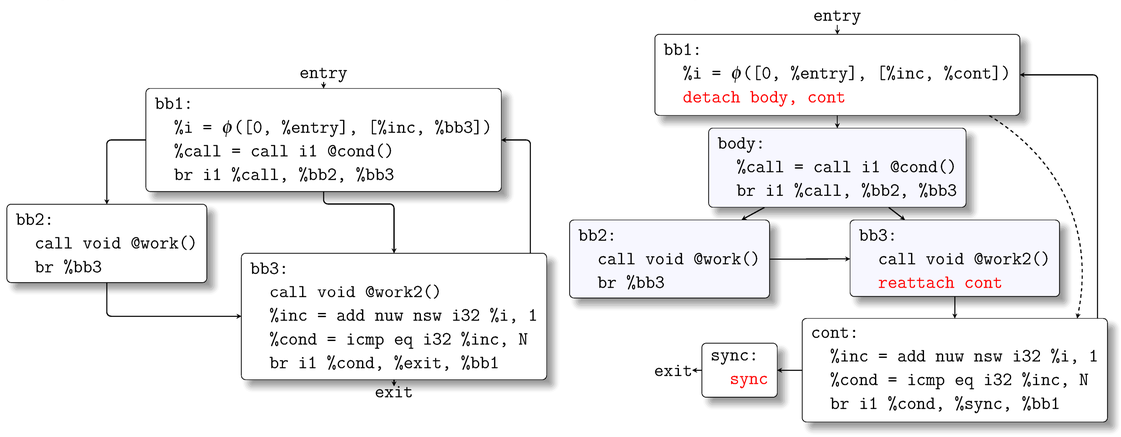

Figure 2. Loop on the right is transformed with `detach`, `reattach`, and `sync` instructions. In the baseline scheme, hints are executed exactly once in each iteration of the loop. There are no through-register dependencies originating in the **body**.

Optimisation 1: Path Specific Hints

In contrast to DOALL and DOACROSS loops, real-world loops often involve complex control flow, where only a portion of the loop can be parallelized. In this approach, we extract parallelism from such loops by placing hints on specific paths.

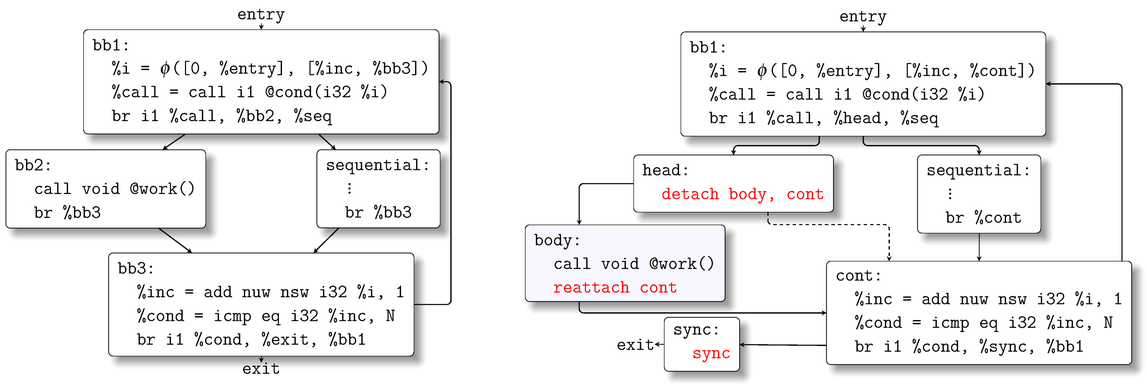

Figure 3. Loop on the right is transformed exposing parallelism on one path. The `detach` should be control-flow equivalent to the `reattach`, i.e. `detach` dominates `reattach` and `reattach` post-dominates `detach`, excluding the loop exits.

Optimisation 2: Reduction Transformation

Reductions in loops inhibit speculative parallelization if not handled explicitly. This transformation parallelizes reduction patterns in loops by demoting through-register reductions to through-memory, allowing the microarchitecture to resolve them as part of conflict checking.

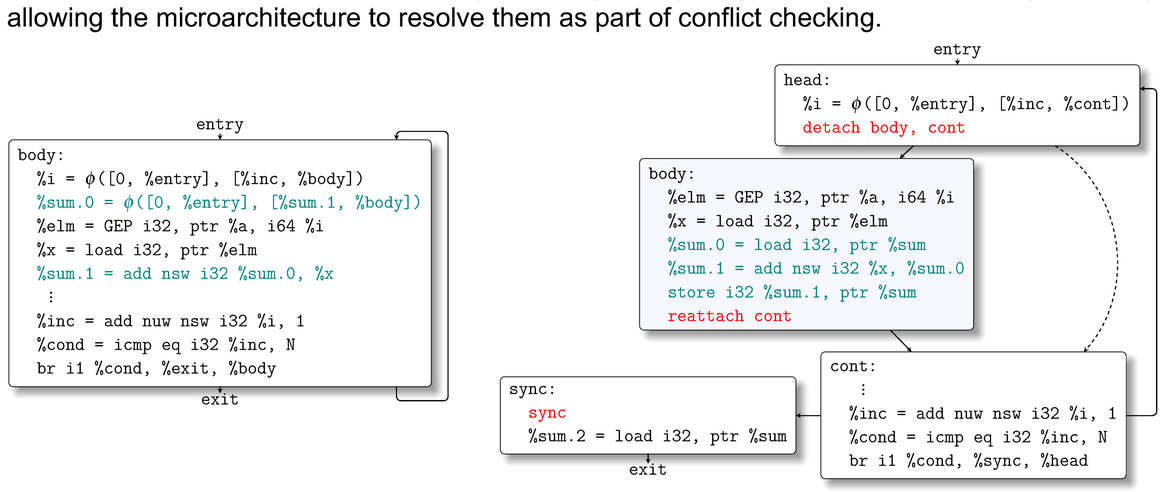

Figure 4. Loop on the left has through-register reduction pattern which is demoted to memory for speculative parallelization as shown on the right.

Optimisation 3: Write-After-Write (WAW) Transformation

Through-register control-dependent WAW conflicts inhibit parallelization. However, memory conflict tracking in the microarchitecture must already handle such patterns. We demote through-register WAW conflits to through-memory exposing parallelism.

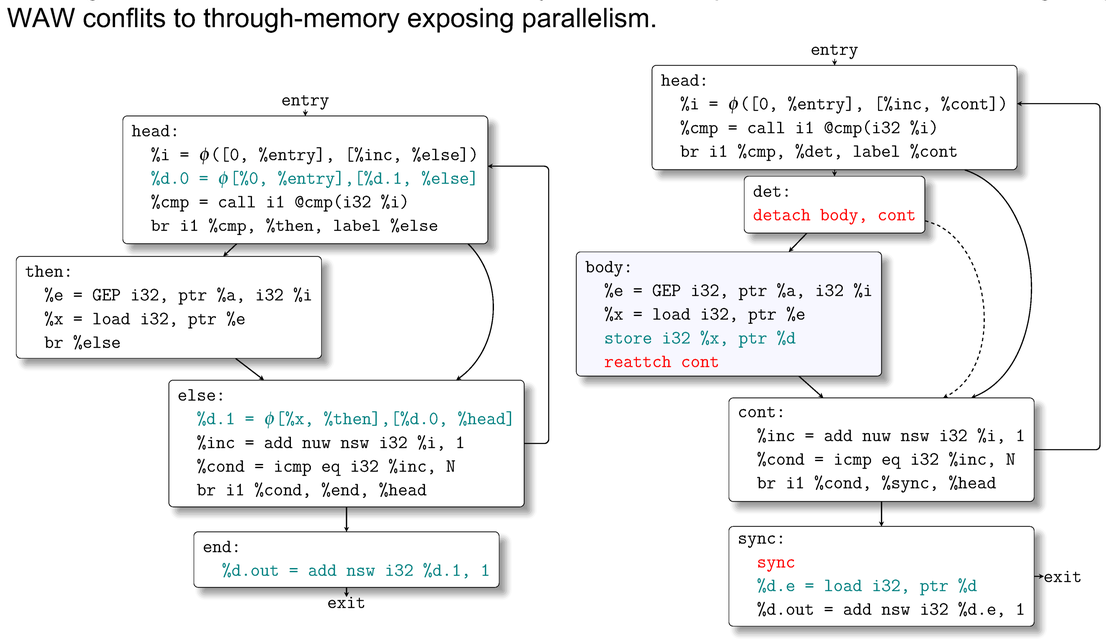

Figure 5. Loop on the left with WAW conflict can be speculatively parallelized as shown on the right.

Implementation

We have implemented these speculative auto-parallelization techniques in OpenCilk LLVM version 16.0 [1]. Our enabling transformations are added at the late stage in the O3 pipeline, and hints are inserted just before lowering. We currently support both pragma-based selective parallelization and full-fury aggressive loop parallelization.

Results

We evaluated the performance of our hardware-assisted TLS scheme on the SPEC CPU 2006 benchmark suite using a gem5-based architectural simulator. We achieve a **16% geomean speedup** over **-O3** optimized binaries.

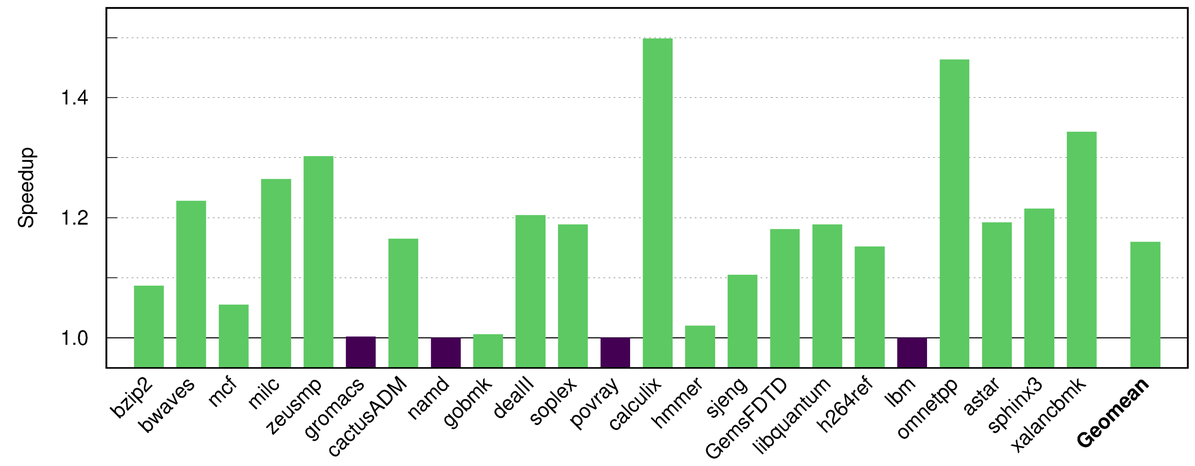

Figure 6. Speedup on SPEC CPU 2006 benchmarks. Purple indicates no speedup.

Acknowledgements

This work was supported by EPSRC (grant EP/W00576X/1) and Arm.

References

[1] Tao B. Schardl and I-Ting Angelina Lee. Opencilk: A modular and extensible software infrastructure for fast task-parallel code. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, pages 189—203, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700156. doi: 10.1145/3572848.3577509. URL https://doi.org/10.1145/3572848.3577509.

[2] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 249—265, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344937. doi: 10.1145/3018743.3018758. URL https://doi.org/10.1145/3018743.3018758.