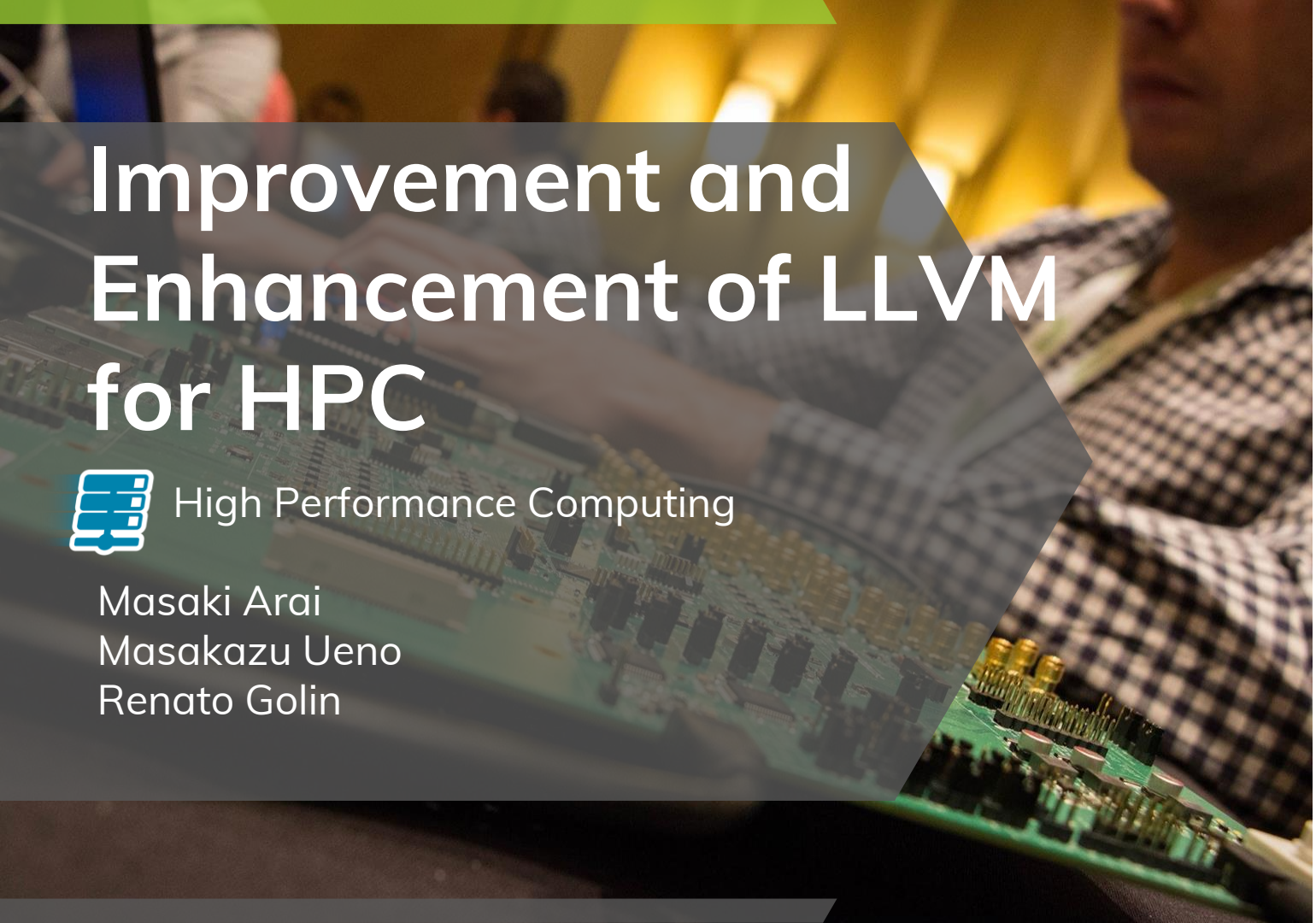


Improvement and Enhancement of LLVM for HPC

High Performance Computing

Masaki Arai
Masakazu Ueno
Renato Golin

Linaro connect
Bangkok 2019

Outline

- Overview of LLVM for HPC
- Eliminating redundant branches for HPC applications
- Activating software pipelining for AArch64
- Improving register allocation for HPC applications
- Vectorization/SIMDization
- Future work

Overview of LLVM for HPC

- We are improving and enhancing LLVM for HPC applications
- Many optimizations and patches are currently being introduced into LLVM for AArch64
- Significant optimization flow for HPC applications is
    1. Pre-processing code for HPC kernels
    2. Distributing and/or fusing loops considering hardware resources
    3. Vectorization/SIMDization considering Scalable Vector Extension(SVE)
    4. Software pipelining considering SVE
    5. Allocating registers for HPC kernels
- Today’s presentation will explain our activities except item 2

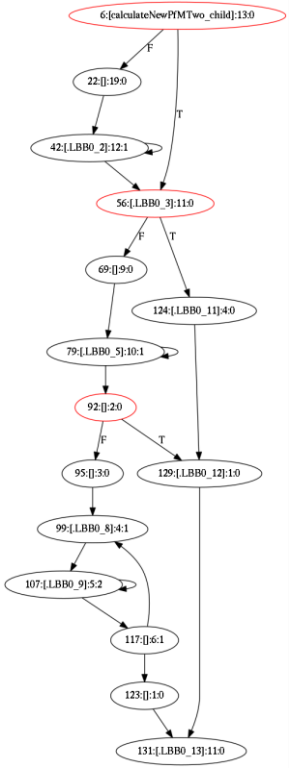
Eliminating redundant branches for HPC applications

fiber-miniapp/mVMC-mini benchmark

```
void calculateNewPfMTwo_child(...) {
  ...
  for (msi = 0; msi < nsize; msi++) {
    ...
  }
  ...
  for (msi = 0; msi < nsize; msi++) {
    ...
  }
  for (msi = 0; msi < nsize; msi++) {
    ...
    for (msj = 0; msj < nsize; msj++) {
    }
  }
  ...
  return;
}
```

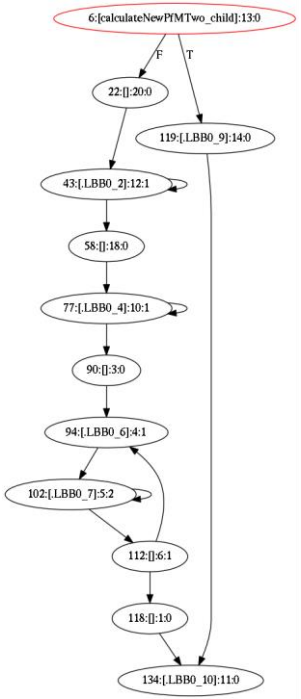
nsize ≦ 0 ?

LLVM 7.0.1



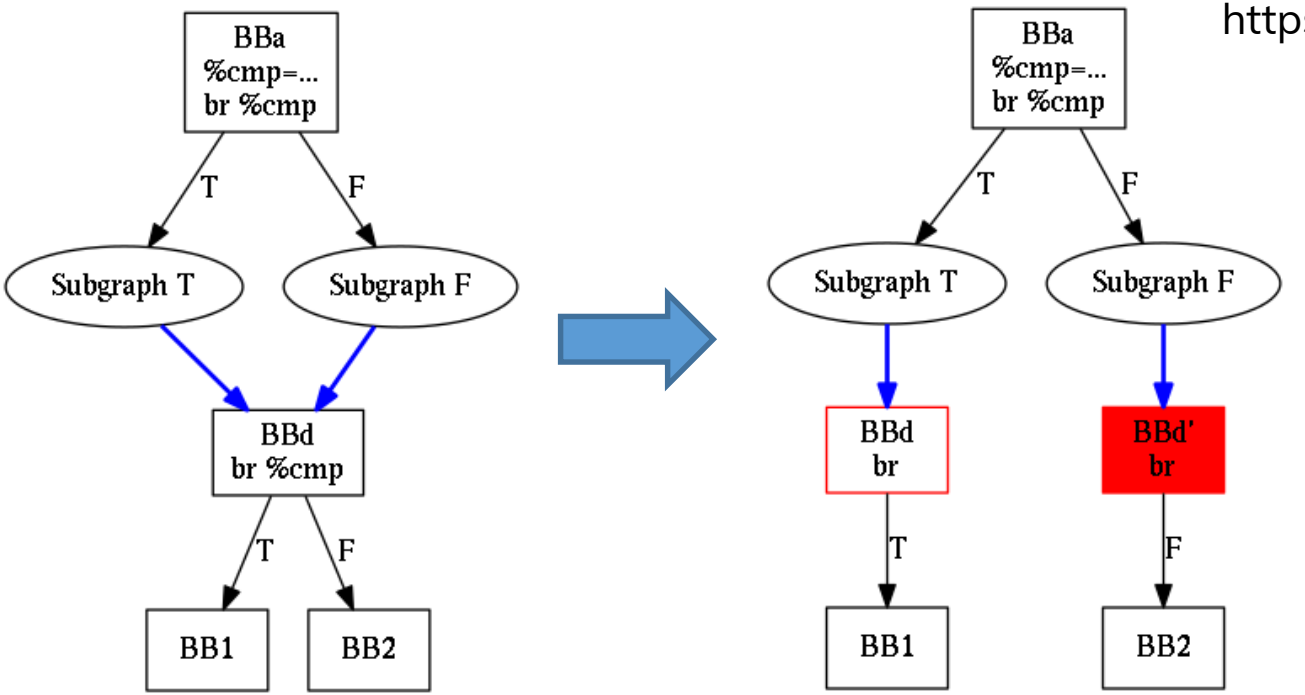
Redundant branches inhibit optimizations for kernel loops.

LLVM 7.0.1
+ our improvement



Eliminating redundant branches for HPC applications

This patch is under review:
https://reviews.llvm.org/D57953



After global value numbering, this optimization can be performed.

Activating software pipelining for AArch64

- Software pipelining is one of significant optimizations for HPC kernel loops
- **MachinePipeliner** was introduced from LLVM 4.0
  - It implements Swing Modulo Scheduling algorithm
  - Its target architecture is currently Hexagon family CPU only
- We are porting **MachinePipeliner** to AArch64 by the following works:
  - Extending **SMSchedule** class for multiple CPU scheduling models of LLVM
  - Adding code specific to AArch64 for recognizing loop induction variables and conditional branches
- This patch is not ready for LLVM upstream yet
  - There is no target machine model for this optimization
  - We are preparing the target machine model for **FUJITSU A64FX**(Armv8.2-A + SVE)

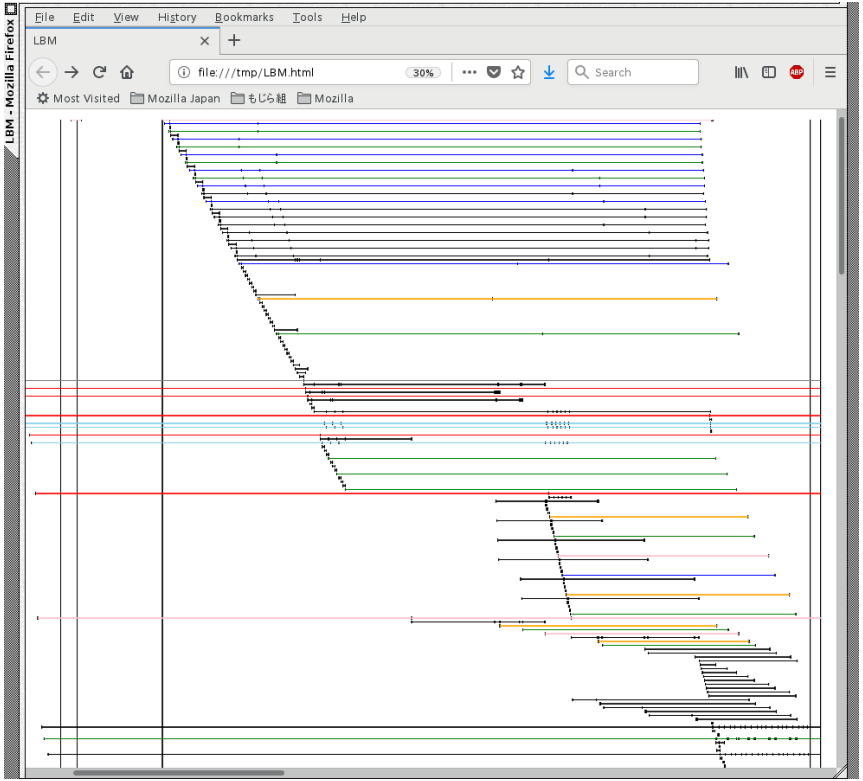
Activating software pipelining for AArch64

- There is some discussion about the enhancement and improvement of **MachinePipeliner**
- Problems and proposals from the round table discussion in LLVM Dev Meeting on 2018/10/17-18
  - There are some problems with schedule node ordering and copy instructions after modulo scheduling
  - Backtracking might be useful for some case
  - This optimization may be better applied to non-SSA form
- One of the proposed changes is:

  A new software pipeliner pass based on non-SSA form
  https://reviews.llvm.org/D55106
- We will keep up with future development direction

Improving register allocation for HPC applications

- We reported problems of **Greedy Register Allocator** of LLVM last year
  - It doesn't care much about HPC kernel loops
  - We think that there are some points to be improved
  - It is not easy to show "improvement" because the structure is complex
- We made a tool to display the internal state of **Greedy Register Allocator** step by step



SPEC CPU2017/619.lbm_s benchmark

Improving register allocation for HPC applications

- HPC kernel loops generally involve a large number of definition and use of floating point variables
- Compilers generate a lot of spill code for HPC kernel loops due to lack of hardware registers
- LLVM currently generates more spill code for HPC kernel loops than GCC
- We modified **Greedy Register Allocator** to **split** virtual registers that unlikely to be allocated aggressively

Compiler	# of load (# of spill-in)	# of store (# of spill-out)
GCC 7.3 -O2	78 (15)	62 (20)
GCC 8.2 -O2	98 (27)	75 (33)
LLVM 7.0.1 -O2	98 (34)	77 (36)
LLVM 7.0.1(modified) -O2	93 (29)	73 (32)

SPEC CPU2017 619.lbm_s/LBM_performStreamCollideTRT Target CPU: thunderx2t99

Vectorization/SIMDization

- Vectorization is a significant optimization that affects the performance of AArch64+SVE
- Vectorization is realized by various functions of the compiler
  - Using special function or library function
  - Using directives like OpenMP
  - Using Polly based on integrated loop optimization
  - Using **VPlan** which estimates effects by the cost model
  - By instruction pattern matching
  - By target machine specific code generation

- One of our activities is to add an instruction pattern to **LoopUtils**
  - Need to improve the cost estimation when conditional branches are in the loop for SVE

```
for (int i = 0; i < LEN; i++) {
    if (a[i] > (float)0.)
      c[i] = -c[i] + d[i] * e[i];
    else
      b[i] = -b[i] + d[i] * e[i];
    a[i] = b[i] + c[i] * d[i];
}
```

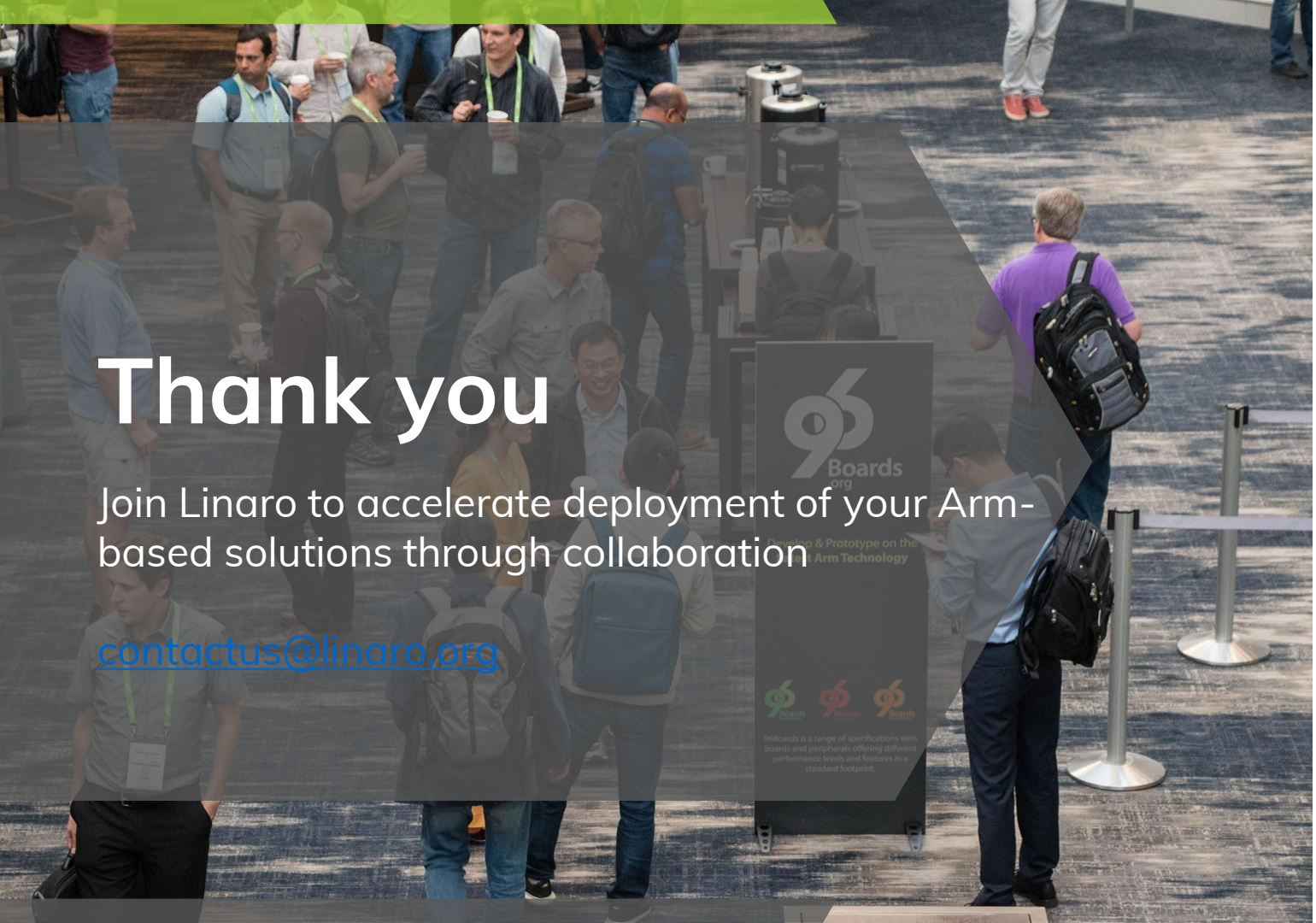
⇒ Clang/LLVM does not vectorize, for now

(Vectorization does not always improve performance since redundant instructions are executed)

After SVE's mask-instr is implemented, performance will improve 2 times faster by SVE conditional instruction execution(w/o SWPL)

Future work

- Proposing our patches to LLVM upstream
- Developing optimization flow for HPC applications
    1. Pre-processing code for HPC kernels
    2. **Distributing and/or fusing loops considering hardware resources**
    3. Vectorization/SIMDization considering Scalable Vector Extension(SVE)
    4. Software pipelining considering SVE
    5. Allocating registers for HPC kernels
- Evaluating the quality of compilers continually



Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org

Linaro connect
Bangkok 2019