Handling Multi-Versioning in LLVM: Code Tracking and Cloning

Alexandra Jimborean   Vincent Loechner   Philippe Clauss

INRIA - CAMUS Team
Université de Strasbourg

London - September, 2011

Outline

Why do we need multi-versioning?

Multi-versioning

- Sampling – Instrumentation
- Adaptive computing – Runtime version selection
- Dynamic optimization – Speculative parallelism

Multiple versions in different representations

- Each version in the most suitable IR
- Low-level IR for acquiring low-level information
- Higher level IR for performing code transformations
- Handled by a runtime system

Why do we need multi-versioning?

Multi-versioning

- Sampling – Instrumentation
- Adaptive computing – Runtime version selection
- Dynamic optimization – Speculative parallelism

Multiple versions in different representations

- Each version in the most suitable IR
- Low-level IR for acquiring low-level information
- Higher level IR for performing code transformations
- Handled by a runtime system

Outline

Related work

Tracking code through the optimization phase

- Extend debugging info and create bi-directional maps [*Brooks et al.*]
- Debug dynamically optimized code [*Kumar et al.*]

Interactive Compilation Interface

- Providing access to the internal functionalities of the compilers
- Generic cloning, instrumentation, control of individual optimization passes
- Multi-versioning available only at function level

http://ctuning.org/ici

LLVM features

Embedding high-level information in the IR

- Support for preserving the high-level information
- Annotate the code using metadata
    - No influence on the optimization passes, unless designed for this

Cloning utilities

- Copies of instructions, basic blocks or functions
- No correlation between original and cloned values
- Reserved only for some very specific situations

Outline

From C/C++ to LLVM IR with metadata

Code tracking in C/C++ source code

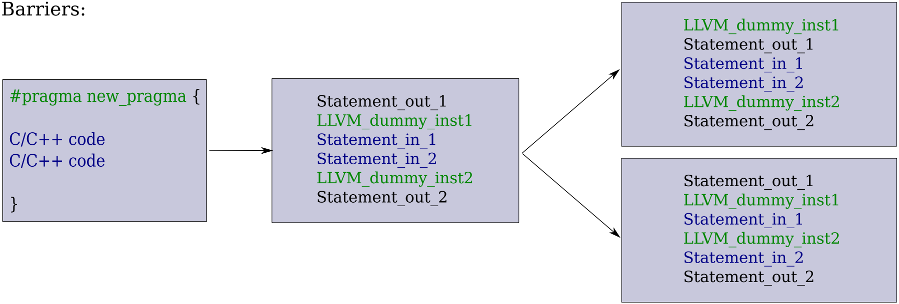
- Source code : pragma
  - Define new pragma to delimit the code regions of interest

```
#pragma multi-version
{
   for(int i=0; i<N; i++)
      a[i] = 2 * i;
}
```

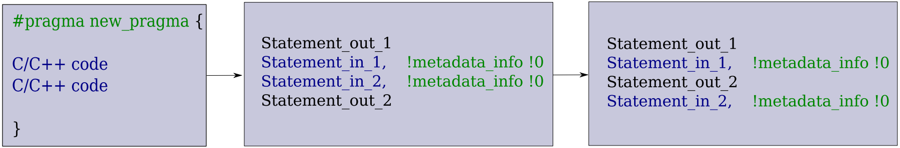
- Focus on loop nests

Extending the IR vs using annotations

Barriers:



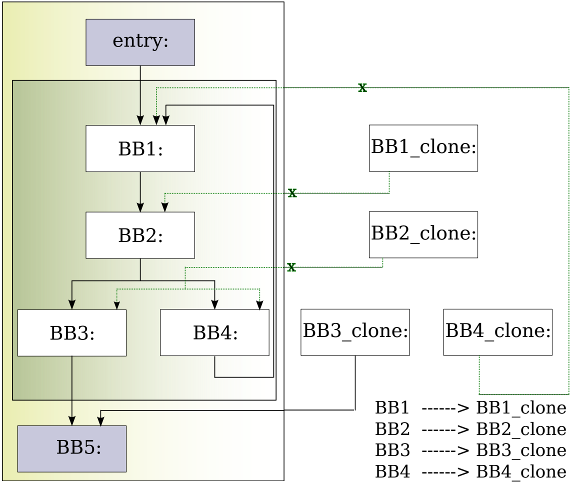
Metadata:



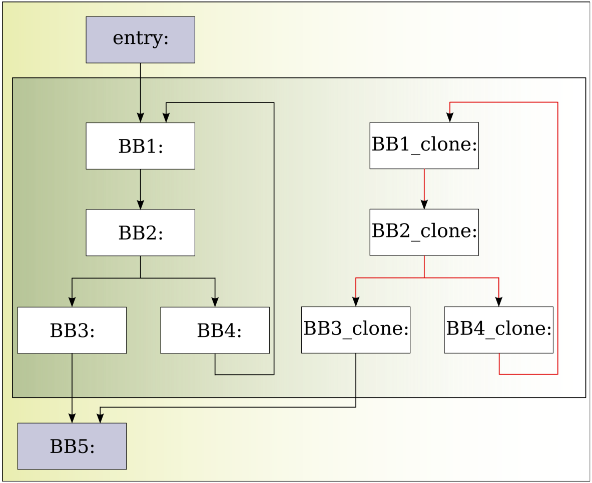
Identify the region after applying optimizations

- Loop nest structure is significantly changed
  - Loop fusion, splitting, interchange etc.
- Metadata information may not be preserved
- Identify instructions that carry metadata information and consider the whole enclosing loop nest
  - Additional code might be included
  - All instructions marked for multiversioning are enclosed

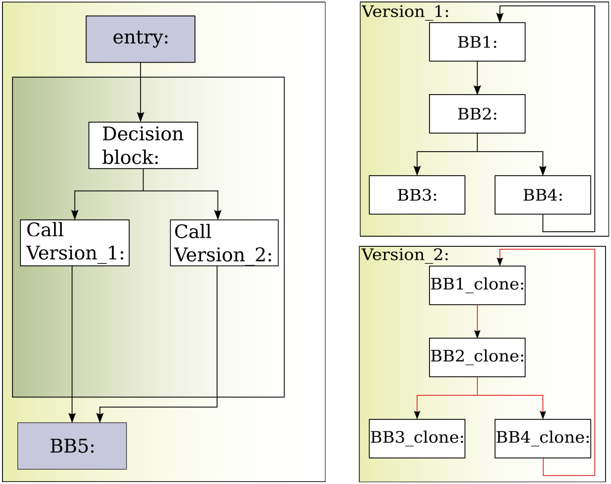
A. Cloning


entry:
BB1:
BB2:
BB3:
BB4:
BB5:
BB1_clone:
BB2_clone:
BB3_clone:
BB4_clone:
x
x
x
BB1 ------> BB1_clone
BB2 ------> BB2_clone
BB3 ------> BB3_clone
BB4 ------> BB4_clone

B. Rebuild control-flow-graph between clones



C. Extract versions in separate functions



Each version compiled independently into the most suitable IR

Challenges: Dominate all uses

Instruction does not dominate all uses!

```
%tmp = add i32 %a, %b
%aux_clone = add i32 %c, %tmp
```

Clone, replace uses in clones, reinsert, reconstruct the loop structure

```
%tmp = add i32 %a, %b
%aux = add i32 %c, %tmp

%tmp_clone = add i32 %a_clone, %b_clone
%aux_clone = add i32 %c_clone, %tmp_clone
```

Outline

Interaction between high- and low-level IRs

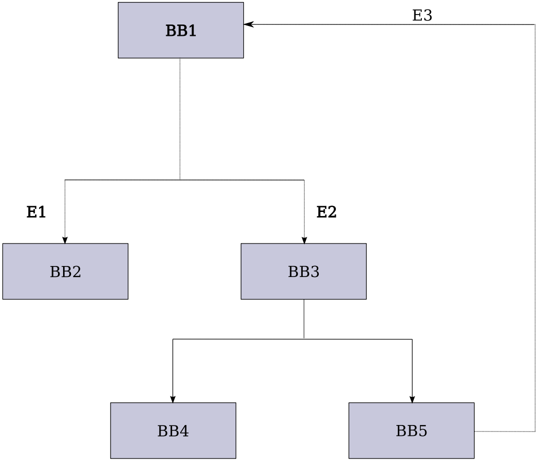
- Communication between code versions in distinct representations
- Control flow cannot enter or exit lower level representations
  - Inline assembly is expected to ‘fall through’ to the following code
- Handle the control flow graph in the low-level IR
- Minimally influence the behavior of the original code

Handling jumps between LLVM IR and inline assembly

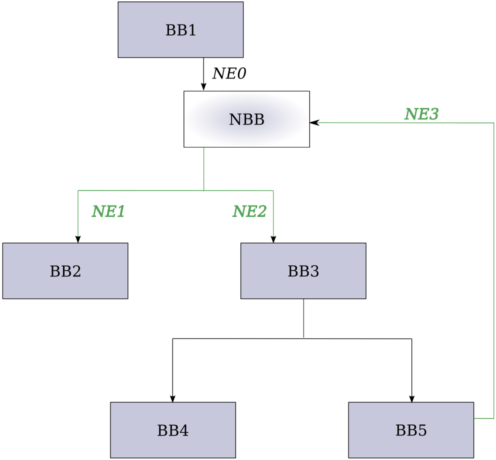
- Generic callbacks - patched by the runtime system
  - `mov $0x0,%rdi //address of the module`
  - `mov $0x0,%rsi //address of the function`
- Labels
  - Identify the address of the code to be patched
  - Target of the inline jumps
- Jumps

Macro	Hexadecimal form
asm_jge8 TARGET	.byte 0X7D .byte \TARGET \()-.-1
asm_jge32 TARGET	.byte 0X0F, 0X8D .long \TARGET \()-.-4

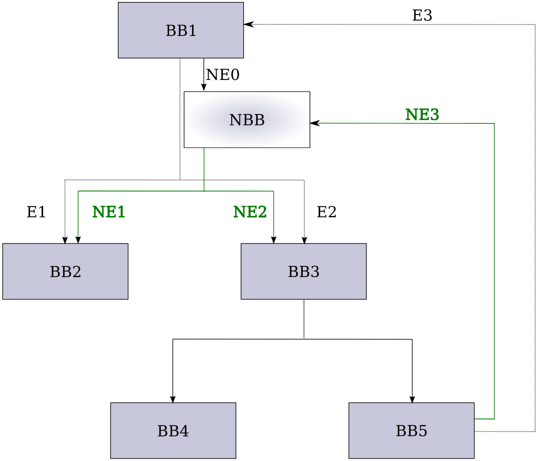
Control flow graph rewritten in inline code



Control flow graph rewritten in inline code



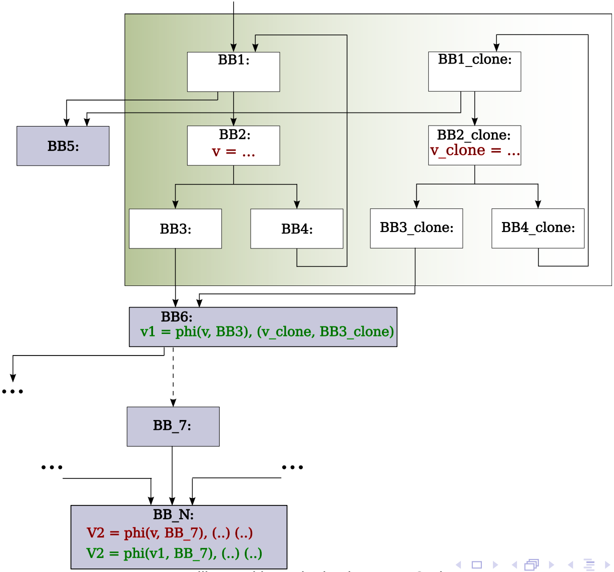
Control flow graph rewritten in inline code



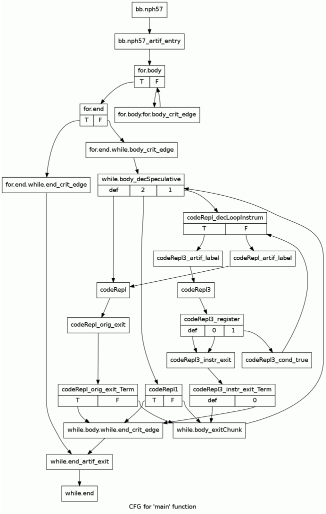
Challenges: Phi nodes

- Promote registers to memory
- opt -reg2mem prg.bc

Eliminate Phi nodes to hack into the CFG



Toy example



bb.nph57
bb.nph57_artif_entry
for.body
T
F
for.end
T
F
for.body.for.body_crit_edge
for.end.while.body_crit_edge
for.end.while.end_crit_edge
while.body_decSpeculative
def
2
1
codeRepl_decLoopInstrum
T
F
codeRepl3_artif_label
codeRepl_artif_label
codeRepl
codeRepl3
codeRepl_orig_exit
codeRepl3_register
def
0
1
codeRepl3_instr_exit
codeRepl3_cond_true
codeRepl_orig_exit_Term
T
F
codeRepl1
T
F
codeRepl3_instr_exit_Term
def
0
while.body.while.end_crit_edge
while.body_exitChunk
while.end_artif_exit
while.end
CFG for 'main' function

SPEC CPU 2006 bzip

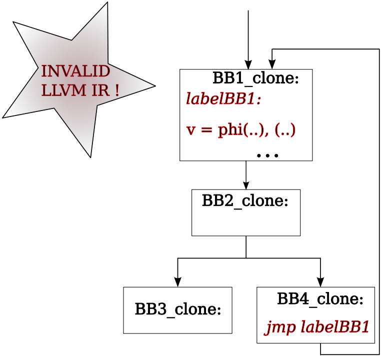
CFG of a simple loop from bzip2 SPEC CPU 2006

Promote registers to memory

- Loop indices must be either defined or used outside the loop, otherwise they are not sent as parameters when extracting the loops in new functions

Promote registers to memory

- *Inline assembly defining labels must come before the phi instructions*



Promote registers to memory

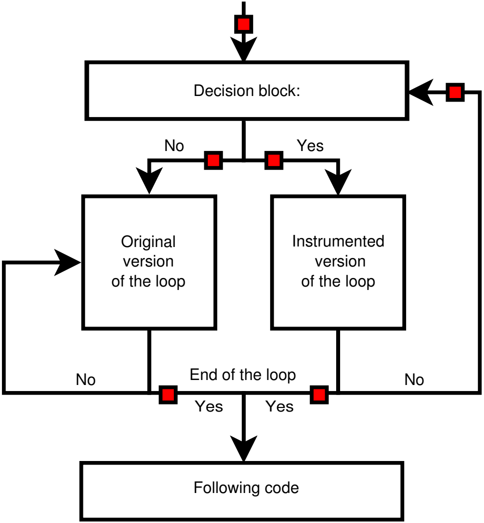
- More memory accesses
- Restricted optimizations
- Negative impact on performance

Challenges: Inline assembly

- Prevent optimizations from duplicating, reordering, deleting the inlined code
  - Create a new BasicBlock containing only the asm code
  - Connect it in the CFG using indirect branches
  - ~~Insert metadata to prevent optimizations~~
- Minimally influence the optimization passes to maintain performance

Outline

Loop Instrumentation by sampling



Challenges: Multiple exit loops

- Extract each loop in a new function
- Unique exit: returning point of the function

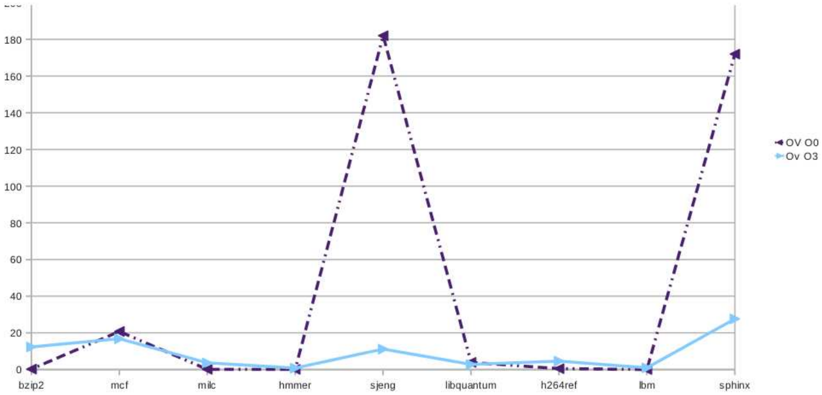
Challenges: Instrumentation instructions

- In x86_64 assembly: after register allocation
- In LLVM IR
  - Requires type conversions
  - Instrumenting all LLVM loads and stores –> negative impact on the performance

Results SPEC CPU 2006

Program	Runtime overhead (-O0)	Runtime overhead (-O3)	# linear m.a.	# instrumented m.a.	Percentage of linear m.a.
bzip2	0.24%	12.31%	608	1,053	57.73%
mcf	20.76%	17.23%	2,848,598	4,054,863	70.25%
milc	0.081%	3.61%	1,988,256,000	1,988,256,195	99.99%
hmmer	0.062%	0.76%	845	0	0%
sjeng	182%	11.13%	1,032,148,267	1,155,459,440	89.32%
libquantum	3.88%	2.76%	203,078	203,581	99.75%
h264ref	0.49%	4.59%	30,707,102	32,452,013	94.62%
lbm	0%	0.93%	358	0	0%
sphinx3	172%	27.62%	51,566,707	78,473,958	65.71%

Measurements on SPEC CPU 2006: -O0 vs -O3



Results

Pointer-Intensive benchmark suite

Program	Runtime overhead	# linear m.a.	# instrumented m.a.	Percentage of linear m.a.
anagram	-5.37%	134	159	84.27%
bc	183%	243,785	302,034	80.71%
ft	-8.46%	22	36	61.11%
ks	29.7%	29,524	42,298	69.79%

Outline

Open questions

- Promoting registers to memory (Phi node elimination)
- Maintain LLVM branches and jumps in inline assembly
- Type conversions

Perspectives

- Speculative code parallelization on the fly using multi-versioning
- Develop an easy-to-use API to extend the framework

Thank you.