DynamicAPInt

Infinite-Precision Arithmetic for LLVM

Arjun Pitchanathan *& Tobias Grosser*
University of Edinburgh, University of Cambridge

Use case: MLIR Presburger Library

Affine loop fusion, …

FPL: Fast Presburger Arithmetic through Transprecision

2024 EURO LLVM
Vienna Austria
DEVELOPERS' MEETING

Introduction

- Compute LB/UB/EQ of index-typed SSA values or (dynamic) dimension sizes of shaped values (tensor/memref).
- Compare two index-typed SSA values or dimension sizes.
- Op interface driven: `ValueBoundsOpInterface`
- Built on top of the MLIR Presburger library.
- Use cases (examples):
  - *Allocation Hoisting*: Compute an upper bound for a dynamic memory allocation size.
  - *Enable Vectorization*: Compute an upper bound of a dynamically-shaped tensor computation.
  - *Subset-based Programming / Bufferization / etc*: Prove that two slices/subviews into the same tensor/memref are equivalent/non-overlapping.

2

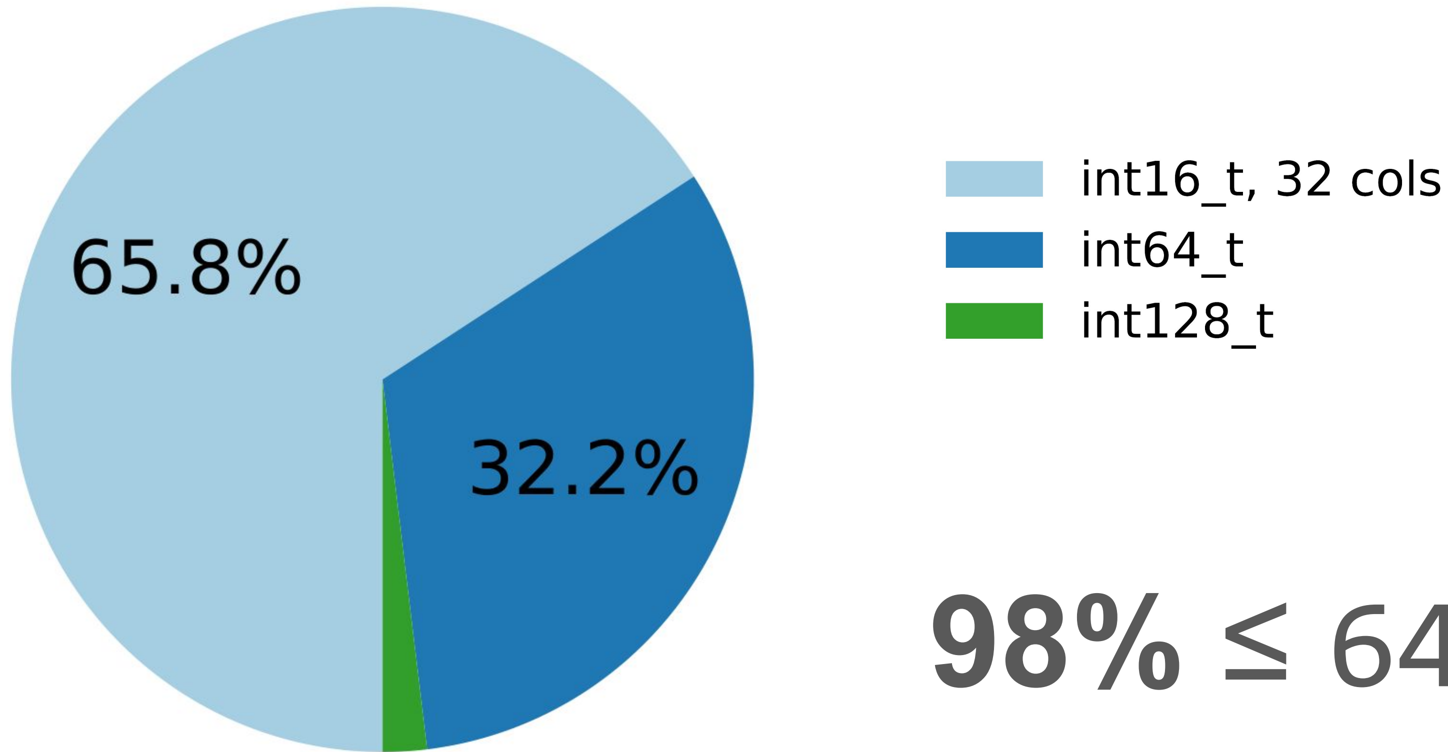
Computing Bounds of SSA Values in MLIR

Matthias Springer

Presburger library: a number cruncher

```
for (unsigned row = 0; row < nRow; ++row) {
  if (tableau[row][pivotCol] == 0)
    continue;
  tableau[row][0] *= tableau[pivotRow][0];
  for (unsigned j = 1; j < nCol; ++j)
    tableau[row][j] = tableau[row][j] * tableau[pivotRow][0] +
                      tableau[row][pivotCol] * tableau[pivotRow][j];
  tableau[row][pivotCol] *= tableau[pivotRow][pivotCol];
}
```

`int64_t` is usually enough in practice…



98% ≤ `64-bit`

…but we need to *know* when it isn't

Roofline:

```
DynamicAPInt &operator*=(DynamicAPInt &x, int64_t y) const {
  bool overflow = __builtin_mul_overflow(x.val, y, &x.val);
  if (!overflow)
    exit(42);
  return x;
}
```

Introducing `DynamicAPInt`

- Arbitrary precision supported for correctness…
- …but heavily optimized for 64-bit fast path.
- Defined as union of `int64_t` and `APInt`

DynamicAPInt operator*=

```
DynamicAPInt &operator*=(DynamicAPInt &x, int64_t y) const {

    bool overflow = __builtin_mul_overflow(x.val, y, &x.val);
    if (!overflow)
      return x;

    exit(42);


}
```

DynamicAPInt operator*=

```
DynamicAPInt &operator*=(DynamicAPInt &x, int64_t y) const {
  if (x.is64()) {
      int64_t result;
      bool overflow = __builtin_mul_overflow(x.val, y, &result);
      if (!overflow) {
        x.val = result;
        return x;
      }
  }
  return slowPath();
}
```

Microbenchmark

```
int64_t x = 1;
for (unsigned i = 0; i < 1000'000'000; ++i)
  if (__builtin_mul_overflow(x, 1, &x))
    exit(42);
```

Microbenchmark

```
int64_t x = 1;
for (unsigned i = 0; i < 1000'000'000; ++i)
  if (__builtin_mul_overflow(x, 1, &x))
    exit(42);
if (x != 1)
  exit(42);
```

Microbenchmark

```
int64_t one = getInput(); // 1
int64_t x = getInput(); // 1
for (unsigned i = 0; i < 1000'000'000; ++i)
  if (__builtin_mul_overflow(x, one, &x))
    exit(42);
if (x != 1)
  exit(42);
```

Microbenchmark

```
int64_t x = 1;
for (unsigned i = 0; i < 1000'000'000; ++i)
  if (__builtin_mul_overflow(x, 1, &x))
    exit(42);
```

Microbenchmark

```
void roofline() {
    int64_t x = 1;
    for (i = 0; i < 1000'000'000; ++i)
      if (__builtin_mul_overflow(x, 1, &x))
        exit(42);
}
void ours() {
    DynamicAPInt x = 1;
    for (i = 0; i < 1000'000'000; ++i)
      x *= 1;
}
```

Unrolling here…

…affects performance here!

Microbenchmark

- Unroll factors 2, 8, 9, 13, 17, 20, 24 are slow.
- `ours()` start addresses 0x453990, 0x4539d0, 0x453a10 are slow.

```
void ours() {
    DynamicAPInt x = 1;
    for (i = 0; i < 1000'000'000; ++i)
      x *= 1;
}
```

Microbenchmark

- Unroll factors 2, 8, 9, 13, 17, 20, 24 are slow.
- `ours()` start addresses 0x453**99**0, 0x453**9d**0, 0x453**a1**0 are slow.

```
void ours() {
    DynamicAPInt x = 1;
    for (i = 0; i < 1000'000'000; ++i)
      x *= 1;
}
```

Microbenchmark

Machine code layout

		0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
factorial_1	**0x9b00**	mov							mov					cmp			
	0x9b10	jb		push	mov					mov							nop
	0x9b20	mul			cmovo				dec			nop	nop	nop	nop	nop	nop
	0x9b30	nop	nop	nop	nop	nop	nop	nop	nop	cmp				ja		add	
	0x9b40			ret	cs nopw									nopl			
factorial_2	**0x9b50**	mov							mov					cmp			
	0x9b60	jb		push	mov					mov							nop
	0x9b70	mul			cmovo				dec			nop	nop	nop	nop	nop	nop
	0x9b80	nop	nop	nop	nop	nop	nop	nop	nop	cmp				ja		add	
	0x9b90			ret	cs nopw									nopl			

}

https://www.bazhenov.me/posts/2024-02-performance-roulette/

Annotating branches!

```
DynamicAPInt &operator*=(DynamicAPInt &x, int64_t y) const {
  if (x.is64()) {
      int64_t result;
      bool overflow = __builtin_mul_overflow(x.val, y, &result);
      if (!overflow) {
        x.val = result;
        return x;
      }
  }
  return slowPath();
}
```

Annotating branches!

```
DynamicAPInt &operator*=(DynamicAPInt &x, int64_t y) const {
  if (LLVM_LIKELY(x.is64())) {
      int64_t result;
      bool overflow = __builtin_mul_overflow(x.val, y, &result);
      if (!overflow) {
        x.val = result;
        return x;
      }
  }
  return slowPath();
```

Annotating branches?

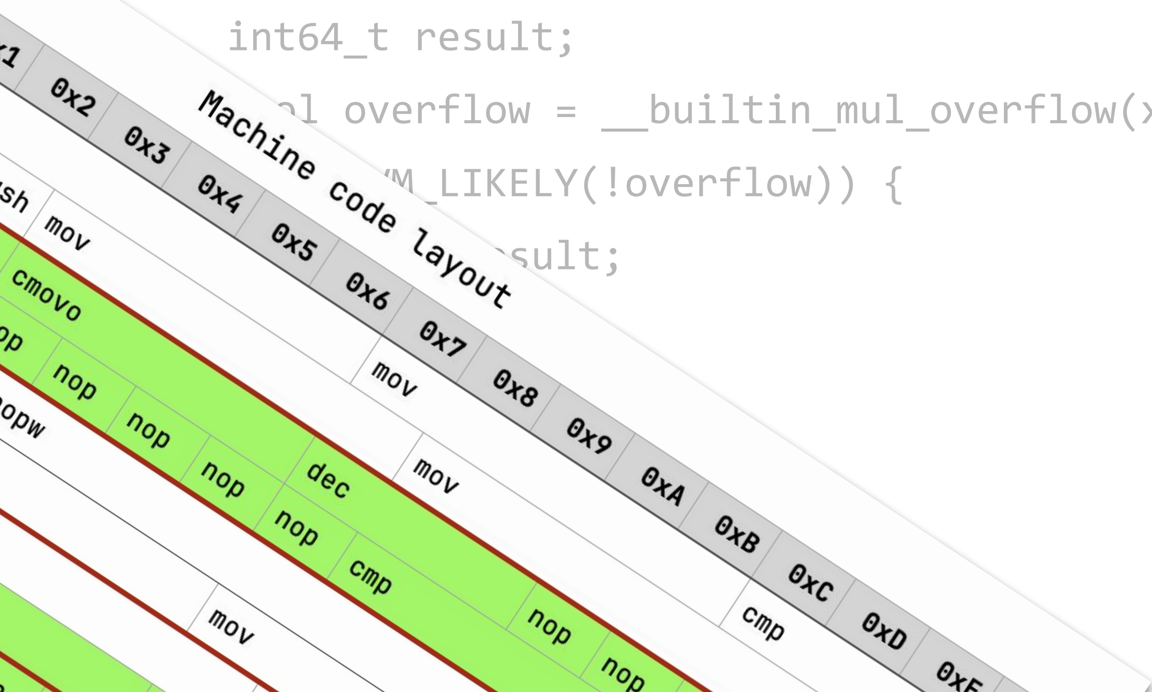
```
DynamicAPInt &operator*=(DynamicAPInt &x, int64_t y) const {
  if (LLVM_LIKELY(x.is64())) {
      int64_t result;
      bool overflow = __builtin_mul_overflow(x.val, y, &result);
      if (!overflow) {
        x.val = result;
        return x;
      }

      slowPath();
```

Machine code layout

0x4 0x5 0x6 0x7

Annotating branches?

```
DynamicAPInt &operator*=(DynamicAPInt &x, int64_t y) const {
  if (LLVM_LIKELY(x.is64())) {
      int64_t result;
        ol overflow = __builtin_mul_overflow(x.val, y, &result);
           M_LIKELY(!overflow)) {
                 sult;
```



Annotating branches!

```
DynamicAPInt &operator*=(DynamicAPInt &x, int64_t y) const {
  if (LLVM_LIKELY(x.is64())) {
      int64_t result;
      bool overflow = __builtin_mul_overflow(x.val, y, &result);
      if (LLVM_LIKELY(!overflow)) {
        x.val = result;
        return x;
      }
  }
  return slowPath();
}
```

`-align-all-functions=6` and `-align-all-nofallthru-blocks=6`

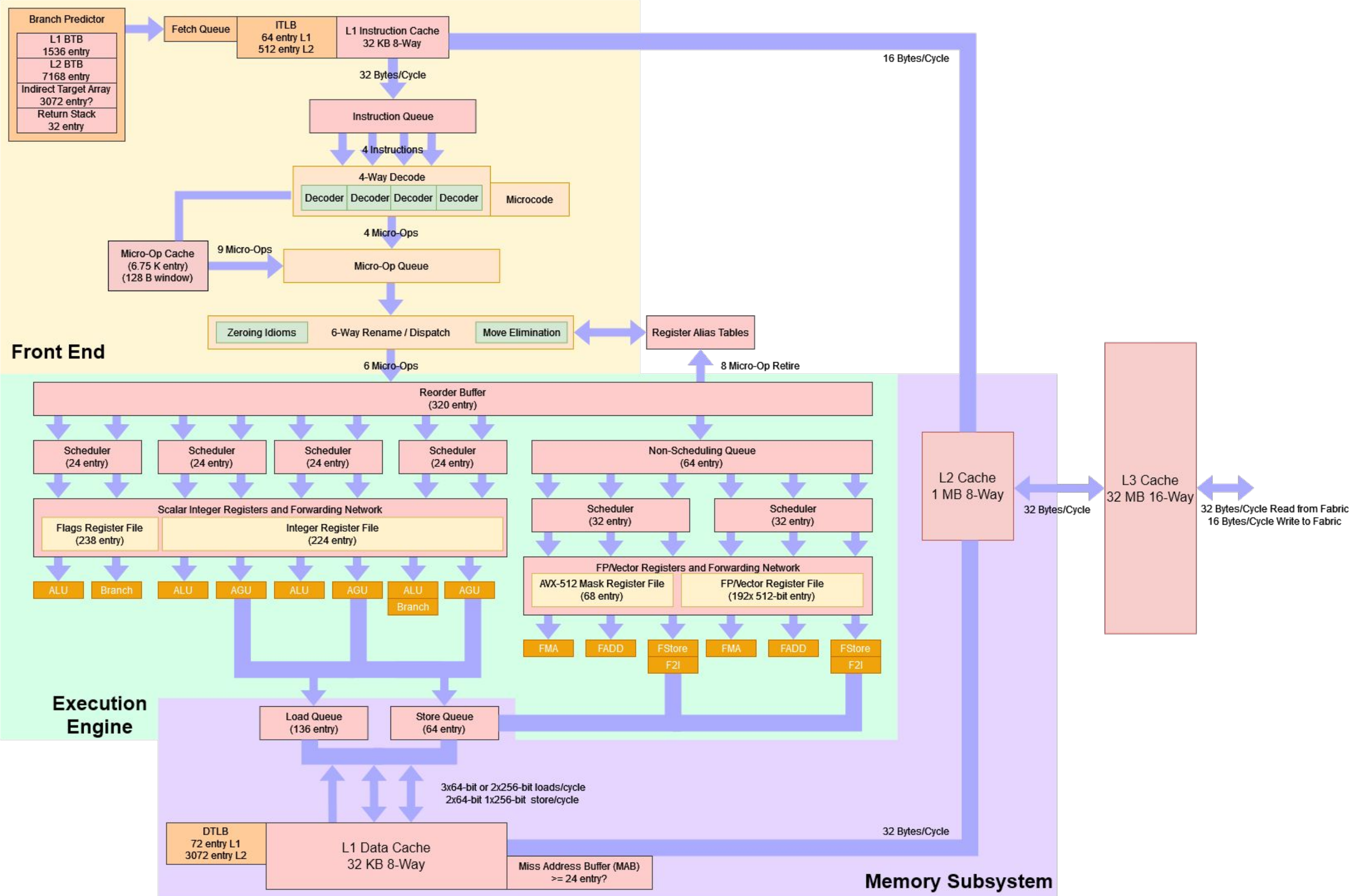
https://www.bazhenov.me/posts/2024-02-performance-roulette/

Inlining

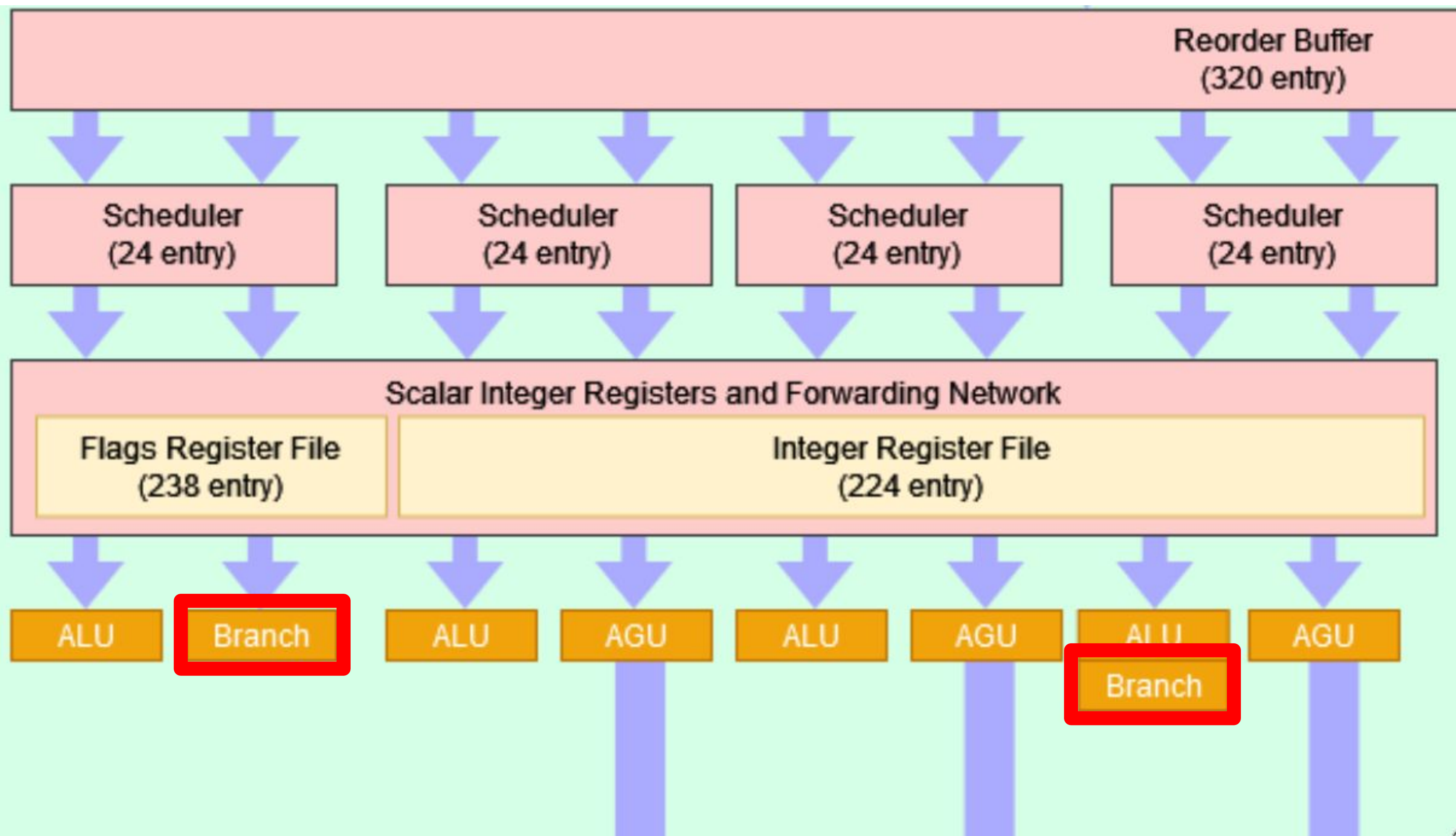
```
LLVM_ATTRIBUTE_ALWAYS_INLINE
DynamicAPInt &operator*=(DynamicAPInt &x, int64_t y) const {
  if (LLVM_LIKELY(x.is64())) {
      int64_t result;
      bool overflow = __builtin_mul_overflow(x.val, y, &result);
      if (LLVM_LIKELY(!overflow)) {
        x.val = result;
        return x;
      }
  }
  return slowPath();
}
```

Microbenchmark

```
for (i = 0; i < 1000'000'000; ++i) {
  if (LLVM_LIKELY(x.is64())) {
    int64_t result;
    bool overflow = __builtin_mul_overflow(x.val, y, &result);
    if (LLVM_LIKELY(!overflow)) {
      x.val = result;
      continue;
    }
  }
  slowPath(x);
}
```



https://chipsandcheese.com/p/amds-zen-4-part-1-frontend-and-execution-engine


Reorder Buffer
(320 entry)
Scheduler
(24 entry)
Scheduler
(24 entry)
Scheduler
(24 entry)
Scheduler
(24 entry)
Scalar Integer Registers and Forwarding Network
Flags Register File
(238 entry)
Integer Register File
(224 entry)
ALU
Branch
ALU
AGU
ALU
AGU
ALU
Branch
AGU

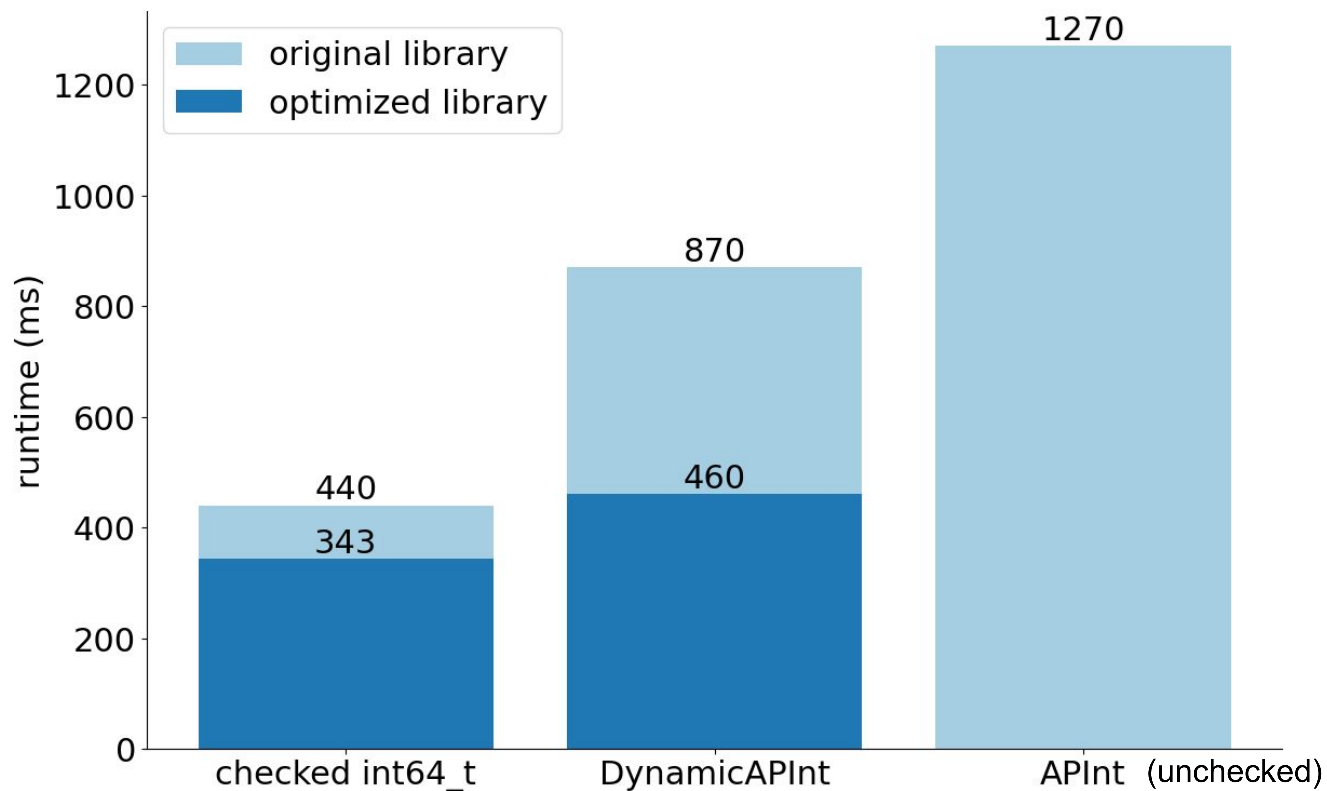
Microbenchmark

```
for (i = 0; i < 1000'000'000; ++i) {
  if (LLVM_LIKELY(x.is64())) {
    int64_t result;
    bool overflow = __builtin_mul_overflow(x.val, y, &result);
    if (!overflow) {
      x.val = result;
      continue;
    }
  }
  slowPath(x);
}
```

Microbenchmark no longer useful

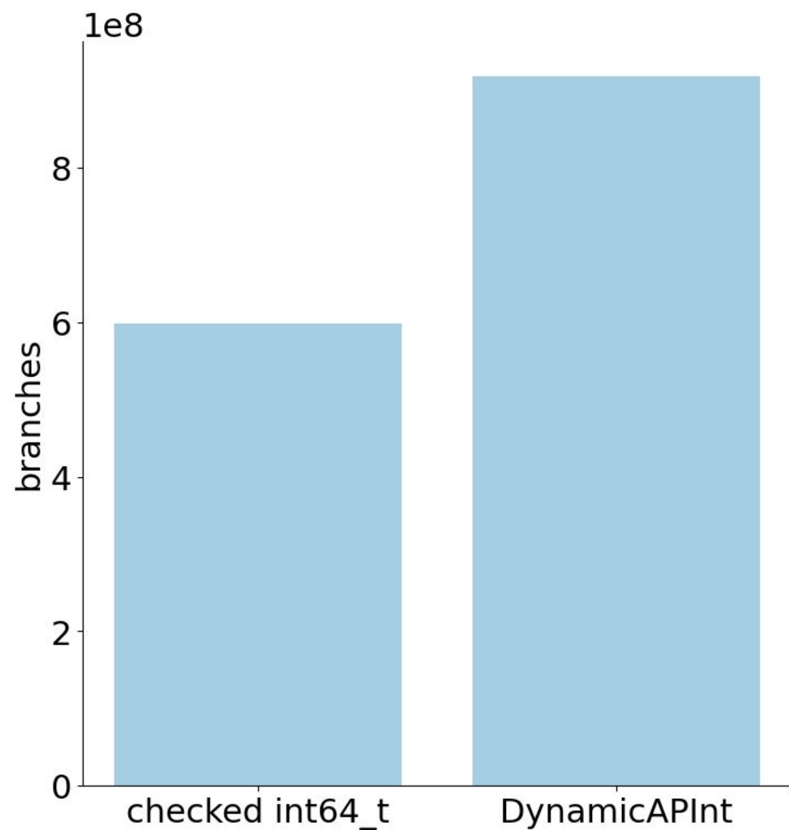
```
for (i = 0; i < 1000'000'000; ++i) {
  if (LLVM_LIKELY(x.is64())) {
    int64_t result;
    bool overflow = __builtin_mul_overflow(x.val, y, &result);
    if (!overflow) {
      x.val = result;
      continue;
    }
  }
  slowPath(x);
}
```

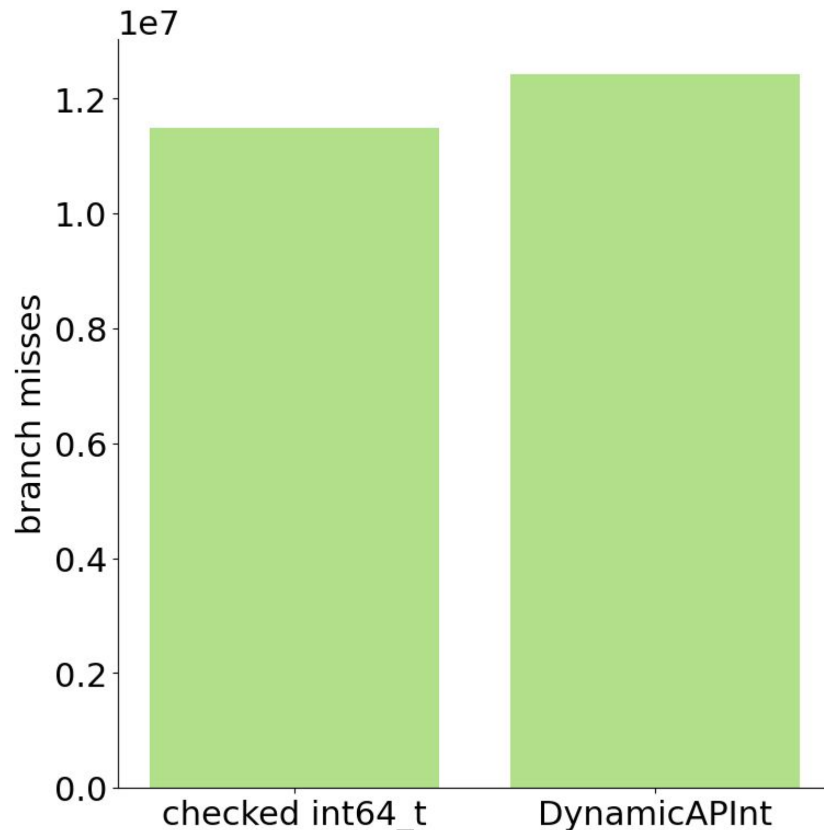
Speedups!



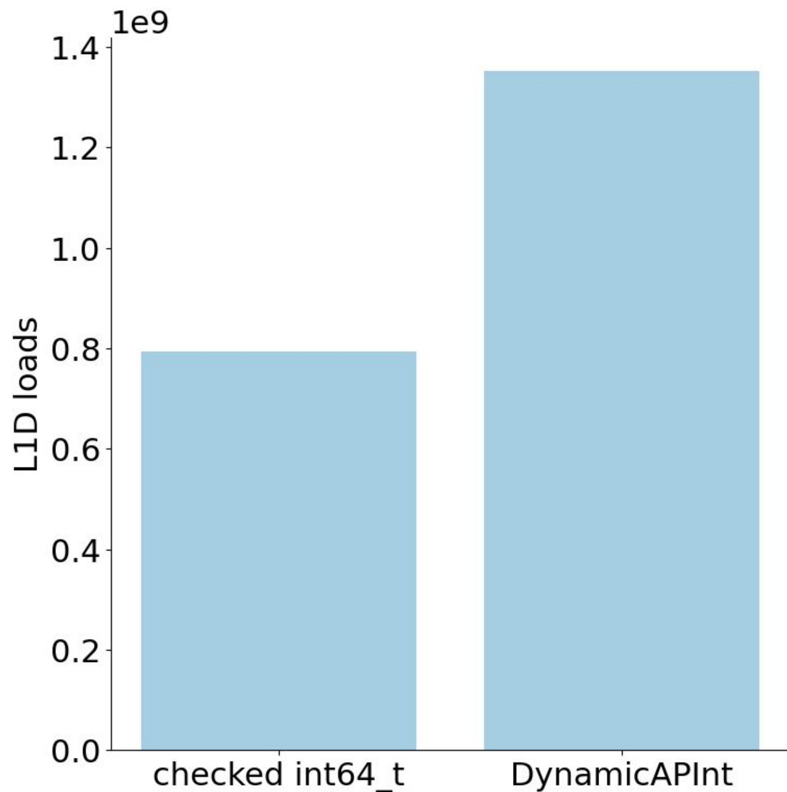
(unchecked)

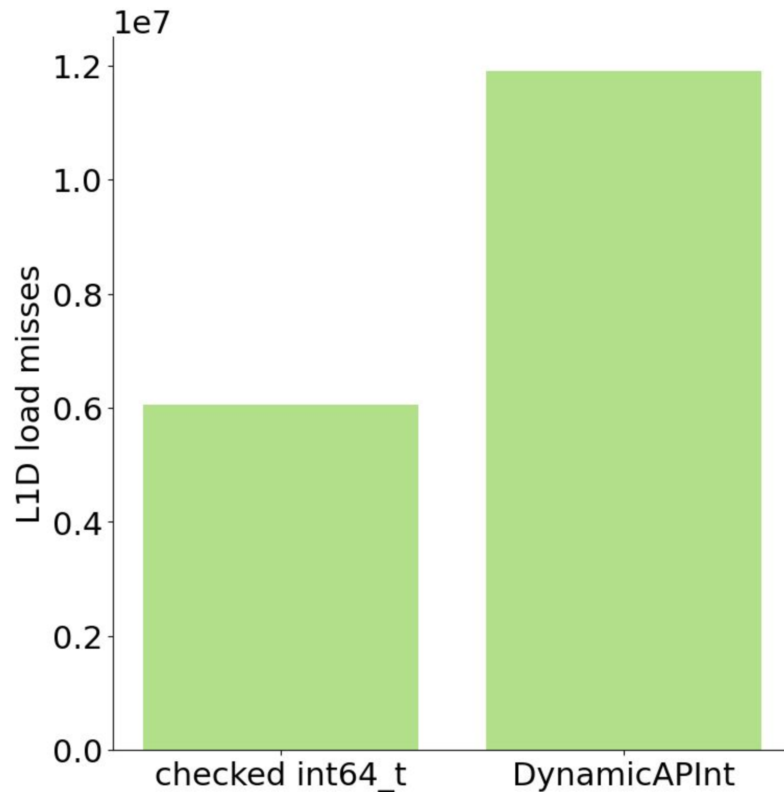
Branch prediction





Memory traffic





Potential next steps

- Library function to run a lambda with just one surrounding fast-path check
- Try a smaller datatype with 32-bit small int instead of 64-bit

Conclusion