Byte Type

Supporting Raw Data Copies in the LLVM IR

Pedro Lobo, Nuno P. Lopes

LLVM Developers Meeting 2025

University of Lisbon

It's not possible to implement `memcpy` in LLVM IR!

Can't implement `memcpy` in LLVM IR

```
void *memcpy(void *dst,
             const void *src,
             size_t n)
{
  unsigned char *d = dst;
  const unsigned char *s = src;
  for (size_t i = 0; i < n; i++)
    d[i] = s[i];
  return dst;
}
```

Can't implement `memcpy` in LLVM IR

> *(An object) value may be copied into an object of type* `unsigned char [n]` *(e.g., by* `memcpy`*); the resulting set of bytes is called the object representation of the value.*
>
> (C99 Standard, 6.2.6.1.4)

Can't implement `memcpy` in LLVM IR

```
void *memcpy(void *dst,
             const void *src,
             size_t n)
{
  unsigned char *d = dst;
  const unsigned char *s = src;
  for (size_t i = 0; i < n; i++)
    d[i] = s[i];
  return dst;
}
```

```
define ptr @memcpy(ptr %dst, ptr %src, i64 %n) {
entry:
  br label %for.cond
for.cond:
  %i = phi i64 [ 0, %entry ], [ %inc, %for.body ]
  %cmp = icmp ult i64 %i, %n
  br i1 %cmp, label %for.body, label %for.end
for.body:
  %arrayidx = gep i8, ptr %src, i64 %i
  %byte = load i8, ptr %arrayidx
  %arrayidx1 = gep i8, ptr %dst, i64 %i
  store i8 %byte, ptr %arrayidx1
  %inc = add i64 %i, 1
  br label %for.cond
for.end:
  ret ptr %dst
}
```

Can't implement `memcpy` in LLVM IR

```
void *memcpy(void *dst,
             const void *src,
             size_t n)
{
  unsigned char *d = dst;
  const unsigned char *s = src;
  for (size_t i = 0; i < n; i++)
    d[i] = s[i];
  return dst;
}
```

```
define ptr @memcpy(ptr %dst, ptr %src, i64 %n) {
entry:
  br label %for.cond
for.cond:
  %i = phi i64 [ 0, %entry ], [ %inc, %for.body ]
  %cmp = icmp ult i64 %i, %n
  br i1 %cmp, label %for.body, label %for.end
for.body:
  %arrayidx = gep i8, ptr %src, i64 %i
  %byte = load i8, ptr %arrayidx
  %arrayidx1 = gep i8, ptr %dst, i64 %i
  store i8 %byte, ptr %arrayidx1
  %inc = add i64 %i, 1
  br label %for.cond
for.end:
  ret ptr %dst
}
```

Two problems with the current `memcpy` lowering

1. Padding Bits
2. Pointer Provenance

Problem 1 - Copying Padding Bits

```
struct S {
  short s;
  int i;
};
```

Problem 1 - Copying Padding Bits

```
struct S {
  short s;
  int i;
};
```

```
%struct.S =
  type { i16, [2 x i8], i32 }
```

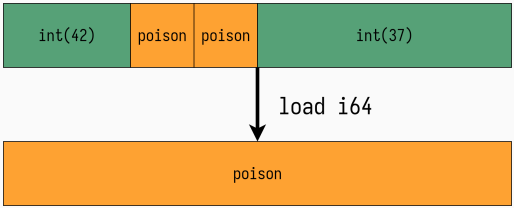
Problem 1 - Copying Padding Bits

```
struct S {
  short s;
  int i;
};
```

```
%struct.S =
  type { i16, [2 x i8], i32 }
```

Problem 1

The loaded value gets tainted if at least one of the loaded bits is poison.



Pointer Provenance

- Pointers are not represented as mere memory addresses.
  - A pointer is represented as a pair (object, offset).

Pointer Provenance

- Pointers are not represented as mere memory addresses.
  - A pointer is represented as a pair (object, offset).
  - Allocations yield a pointer referring to an unique object.

Pointer Provenance

- Pointers are not represented as mere memory addresses.
  - A pointer is represented as a pair (object, offset).
  - Allocations yield a pointer referring to an unique object.

```
char *p = malloc(4); // (obj1, 0)
char *q = malloc(4); // (obj2, 0)
```

Pointer Provenance

- Pointers are not represented as mere memory addresses.
  - A pointer is represented as a pair (object, offset).
  - Allocations yield a pointer referring to an unique object.
  - Pointer arithmetic only modifies the offset portion.

```
char *p = malloc(4); // (obj1, 0)
char *q = malloc(4); // (obj2, 0)
```

Pointer Provenance

- Pointers are not represented as mere memory addresses.
  - A pointer is represented as a pair (object, offset).
  - Allocations yield a pointer referring to an unique object.
  - Pointer arithmetic only modifies the offset portion.

```
char *p = malloc(4); // (obj1, 0)
char *q = malloc(4); // (obj2, 0)
char *pi = p + i;    // (obj1, i)
char *qj = q + j;    // (obj2, j)
```

Pointer Provenance

- Pointers are not represented as mere memory addresses.
  - A pointer is represented as a pair (object, offset).
  - Allocations yield a pointer referring to an unique object.
  - Pointer arithmetic only modifies the offset portion.
  - Pointers referent to different objects do not alias.

```
char *p = malloc(4); // (obj1, 0)
char *q = malloc(4); // (obj2, 0)
char *pi = p + i;    // (obj1, i)
char *qj = q + j;    // (obj2, j)
```

Pointer Provenance

- Pointers are not represented as mere memory addresses.
  - A pointer is represented as a pair (object, offset).
  - Allocations yield a pointer referring to an unique object.
  - Pointer arithmetic only modifies the offset portion.
  - Pointers referent to different objects do not alias.

```
char *p = malloc(4); // (obj1, 0)
char *q = malloc(4); // (obj2, 0)
char *pi = p + i;    // (obj1, i)
char *qj = q + j;    // (obj2, j)
*pi = 1; // UB if (intptr_t)pi == (intptr_t)qj
*qj = 2; // thus we can assume they write to different locations
```

Pointer Provenance

- Pointers are not represented as mere memory addresses.
  - A pointer is represented as a pair (object, offset).
  - Allocations yield a pointer referring to an unique object.
  - Pointer arithmetic only modifies the offset portion.
  - Pointers referent to different objects do not alias.

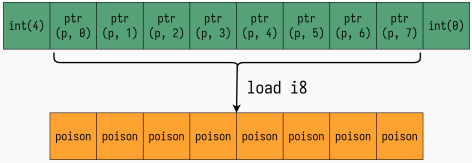
```
char *p = malloc(4); // (obj1, 0)
char *q = malloc(4); // (obj2, 0)
char *pi = p + i;    // (obj1, i)
char *qj = q + j;    // (obj2, j)
*pi = 1; // UB if (intptr_t)pi == (intptr_t)qj
*qj = 2; // thus we can assume they write to different locations
printf("%d\n", *pi); // we can print 1
```

Problem 2 - Copying Pointer Values

```
define ptr @memcpy(ptr %dst, ptr %src, i64 %n) {
entry:
  br label %for.cond
for.cond:
  %i = phi i64 [ 0, %entry ], [ %inc, %for.body ]
  %cmp = icmp ult i64 %i, %n
  br i1 %cmp, label %for.body, label %for.end
for.body:
  %arrayidx = gep i8, ptr %src, i64 %i
  %byte = load i8, ptr %arrayidx
  %arrayidx1 = gep i8, ptr %dst, i64 %i
  store i8 %byte, ptr %arrayidx1
  %inc = add i64 %i, 1
  br label %for.cond
for.end:
  ret ptr %dst
}
```

Problem 2

Loading a pointer byte through a non-pointer type returns `poison`.

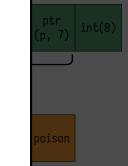


```
define p
entry:
  br lab
for.cond
  %i = p
  %cmp =
  br i1
for.body
  %array
  %byte
  %array
  store
  %inc =
  br lab
for.end:
  ret ptr %dst
}
```

Alternative Semantics

- Loading a pointer through a non-pointer type returns a pointer with no provenance, rather than `poison`.
  - The resulting pointer can only be used in comparisons.

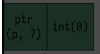
ugh a
ison.



```
define p
entry:
  br lab
for.cond
  %i = p
  %cmp =
  br i1
for.body
  %array
  %byte
  %array
  store
  %inc =
  br lab
for.end:
  ret ptr %dst
}
```

ugh a

ison.



Alternative Semantics

- Loading a pointer through a non-pointer type returns a pointer with no provenance, rather than `poison`.
  - The resulting pointer can only be used in comparisons.
- Cannot solve the problem of copying pointer values, as the pointer's provenance is still discarded.

Byte Type

- Add a new type to the IR, able to represent raw memory data.
  - The byte type is able to hold a memory value as-is (preserving padding bits).
  - Loads with the new type do not perform implicit casts.
  - Provenance information is preserved.

Byte Type

- Add a new type to the IR, able to represent raw memory data.
  - The byte type is able to hold a memory value as-is (preserving padding bits).
  - Loads with the new type do not perform implicit casts.
  - Provenance information is preserved.
- Lower `unsigned char`, `signed char`, `char` and `std::byte` to the byte type.

```
void foo(
  unsigned char arg1,
  char          arg2,
  signed char   arg3,
  std::byte     arg4
);
```

```
void @foo(
  b8 zeroext %arg1,
  b8 signext %arg2,
  b8 signext %arg3,
  b8 zeroext %arg4
);
```

Byte Constants

- Byte constants are equivalent to integer constants.
- Needed to initialize global variables.

Byte Constants

- Byte constants are equivalent to integer constants.
- Needed to initialize global variables.

```
struct s {
  int i;
  char c;
} g = { 42, 'C' };
```

Byte Constants

- Byte constants are equivalent to integer constants.
- Needed to initialize global variables.

```
struct s {
  int i;
  char c;
} g = { 42, 'C' };
```

```
@g = global
  { i32, b8, [3 x i8] }
  { i32 42, b8 67, [3 x i8] poison }
```

bytecast Instruction

- The `bytecast` instruction converts byte values to other primitive types.
  - By default, type punning is performed.

bytecast Instruction

- The `bytecast` instruction converts byte values to other primitive types.
  - By default, type punning is performed.

Performs type punning

```
%v = load b8, ptr %p
%c = bytecast b8 %v to i8
```

bytecast Instruction

- The `bytecast` instruction converts byte values to other primitive types.
  - By default, type punning is performed.
  - The `exact` flag disallows type punning by returning `poison`.

Performs type punning

```
%v = load b8, ptr %p
%c = bytecast b8 %v to i8
```

Does not perform type punning

```
%v = load b8, ptr %p
%c = bytecast exact b8 %v to i8
```

Clang: Lowering `char` to the byte type

C Code

```
char f(char a) {
  return a + 1;
}
```

Clang: Lowering `char` to the byte type

C Code

```
char f(char a) {
  return a + 1;
}
```

Current Lowering

```
define i8 @f(i8 %c) {
  %i32 = sext i8 %c to i32
  %add = add nsw i32 %i32, 1
  %t = trunc i32 %add to i8
  ret i8 %t
}
```

Clang: Lowering `char` to the byte type

C Code

```
char f(char a) {
  return a + 1;
}
```

Current Lowering

```
define i8 @f(i8 %c) {
  %i32 = sext i8 %c to i32
  %add = add nsw i32 %i32, 1
  %t = trunc i32 %add to i8
  ret i8 %t
}
```

New Lowering

```
define b8 @f(b8 %c) {
  %i8 = bytecast b8 %c to i8
  %i32 = sext i8 %i8 to i32
  %add = add nsw i32 %i32, 1
  %t = trunc i32 %add to i8
  %r = bitcast i8 %t to b8
  ret b8 %r
}
```

Extracting bytes

- Extend `trunc` and `lshr` instructions to accept byte operands.

Extracting bytes

- Extend `trunc` and `lshr` instructions to accept byte operands.

```
define i8 @src(b32 %x) {
  %a = alloca b32
  store b32 %x, ptr %a
  %gep = gep i8, ptr %a, i64 2
  %v = load i8, ptr %gep
  ret i8 %v
}
```

Extracting bytes

- Extend `trunc` and `lshr` instructions to accept byte operands.

```
define i8 @src(b32 %x) {
  %a = alloca b32
  store b32 %x, ptr %a
  %gep = gep i8, ptr %a, i64 2
  %v = load i8, ptr %gep
  ret i8 %v
}
```

```
define i8 @tgt(b32 %x) {
  %shift = lshr b32 %x, 16
  %trunc = trunc b32 %shift to b8
  %cast = bytecast exact b8 to i8
  ret i8 %cast
}
```

Fixed `memcpy` Lowering

Before

```
define ptr @memcpy(ptr %dst, ptr %src, i64 %n) {
entry:
  br label %for.cond
for.cond:
  %i = phi i64 [ 0, %entry ], [ %inc, %for.body ]
  %cmp = icmp ult i64 %i, %n
  br i1 %cmp, label %for.body, label %for.end
for.body:
  %arrayidx = gep i8, ptr %src, i64 %i
  %byte = load i8, ptr %arrayidx
  %arrayidx1 = gep i8, ptr %dst, i64 %i
  store i8 %byte, ptr %arrayidx1
  %inc = add i64 %i, 1
  br label %for.cond
for.end:
  ret ptr %dst
}
```

After

```
define ptr @memcpy(ptr %dst, ptr %src, i64 %n) {
entry:
  br label %for.cond
for.cond:
  %i = phi i64 [ 0, %entry ], [ %inc, %for.body ]
  %cmp = icmp ult i64 %i, %n
  br i1 %cmp, label %for.body, label %for.end
for.body:
  %arrayidx = gep i8, ptr %src, i64 %i
  %byte = load b8, ptr %arrayidx
  %arrayidx1 = gep i8, ptr %dst, i64 %i
  store b8 %byte, ptr %arrayidx1
  %inc = add i64 %i, 1
  br label %for.cond
for.end:
  ret ptr %dst
}
```

Fixed `memcpy` Lowering

Before

```
define ptr @memcpy(ptr %dst, ptr %src, i64 %n) {
entry:
  br label %for.cond
for.cond:
  %i = phi i64 [ 0, %entry ], [ %inc, %for.body ]
  %cmp = icmp ult i64 %i, %n
  br i1 %cmp, label %for.body, label %for.end
for.body:
  %arrayidx = gep i8, ptr %src, i64 %i
  %byte = load i8, ptr %arrayidx
  %arrayidx1 = gep i8, ptr %dst, i64 %i
  store i8 %byte, ptr %arrayidx1
  %inc = add i64 %i, 1
  br label %for.cond
for.end:
  ret ptr %dst
}
```

After

```
define ptr @memcpy(ptr %dst, ptr %src, i64 %n) {
entry:
  br label %for.cond
for.cond:
  %i = phi i64 [ 0, %entry ], [ %inc, %for.body ]
  %cmp = icmp ult i64 %i, %n
  br i1 %cmp, label %for.body, label %for.end
for.body:
  %arrayidx = gep i8, ptr %src, i64 %i
  %byte = load b8, ptr %arrayidx
  %arrayidx1 = gep i8, ptr %dst, i64 %i
  store b8 %byte, ptr %arrayidx1
  %inc = add i64 %i, 1
  br label %for.cond
for.end:
  ret ptr %dst
}
```

Fixed `memcpy` Lowering

Before

```
define ptr @memcpy(ptr %dst, ptr %src, i64 %n) {
entry:
  br label %for.cond
for.cond:
  %i = phi i64 [ 0, %entry ], [ %inc, %for.body ]
  %cmp = icmp ult i64 %i, %n
  br i1 %cmp, label %for.body, label %for.end
for.body:
  %arrayidx = gep i8, ptr %src, i64 %i
  %byte = load i8, ptr %arrayidx
  %arrayidx1 = gep i8, ptr %dst, i64 %i
  store i8 %byte, ptr %arrayidx1
  %inc = add i64 %i, 1
  br label %for.cond
for.end:
  ret ptr %dst
}
```

After

```
define ptr @memcpy(ptr %dst, ptr %src, i64 %n) {
entry:
  br label %for.cond
for.cond:
  %i = phi i64 [ 0, %entry ], [ %inc, %for.body ]
  %cmp = icmp ult i64 %i, %n
  br i1 %cmp, label %for.body, label %for.end
for.body:
  %arrayidx = gep i8, ptr %src, i64 %i
  %byte = load b8, ptr %arrayidx
  %arrayidx1 = gep i8, ptr %dst, i64 %i
  store b8 %byte, ptr %arrayidx1
  %inc = add i64 %i, 1
  br label %for.cond
for.end:
  ret ptr %dst
}
```

Fixed `memcmp` Lowering

Before

```
define i32 @memcmp(ptr %p1, ptr %p2) {
  %lhsc = load i8, ptr %p1
  %lhsv = zext i8 %lhsc to i32
  %rhsc = load i8, ptr %p2
  %rhsv = zext i8 %rhsc to i32
  %chardiff = sub nsw i32 %lhsv, %rhsv
  ret i32 %chardiff
}
```

After

```
define i32 @memcmp(ptr %p1, ptr %p2) {
  %lhsb = load b8, ptr %p1
  %lhsc = bytecast b8 %lhsb to i8
  %lhsv = zext i8 %lhsc to i32
  %rhsb = load b8, ptr %p2
  %rhsc = bytecast b8 %rhsb to i8
  %rhsv = zext i8 %rhsc to i32
  %chardiff = sub nsw i32 %lhsv, %rhsv
  ret i32 %chardiff
}
```

Fixed `memcmp` Lowering

Before

```
define
  %lhs
  %lhsv
  %rhs
  %rhsv
  %char
  ret i
}
```

After

```
r %p2) {
  i8
  i8
v, %rhsv
  ret i32 %chardiff
}
```

Alternative Semantics

- Loads of pointers through integer types return a pointer with no provenance.
- Avoids changing the current `memcmp` lowering.

Merging Loads

- GVN merges loads of pointer and non-pointer types from the same address.
  - Can discard the pointer's provenance.

```
%a = load i64, ptr %p
%b = load ptr, ptr %p
```

Merging Loads

- GVN merges loads of pointer and non-pointer types from the same address.
  - Can discard the pointer's provenance.

```
%a = load i64, ptr %p
%b = load ptr, ptr %p
```

Unsound Coercion

```
%a = load i64, ptr %p
%b = inttoptr i64 %a to ptr
```

Merging Loads

- GVN merges loads of pointer and non-pointer types from the same address.
  - Can discard the pointer's provenance.

```
%a = load i64, ptr %p
%b = load ptr, ptr %p
```

Unsound Coercion

```
%a = load i64, ptr %p
%b = inttoptr i64 %a to ptr
```

Fixed Coercion

```
%v = load b64, ptr %p
%a = bytecast b64 %v to i64
%b = bytecast b64 %v to ptr
```

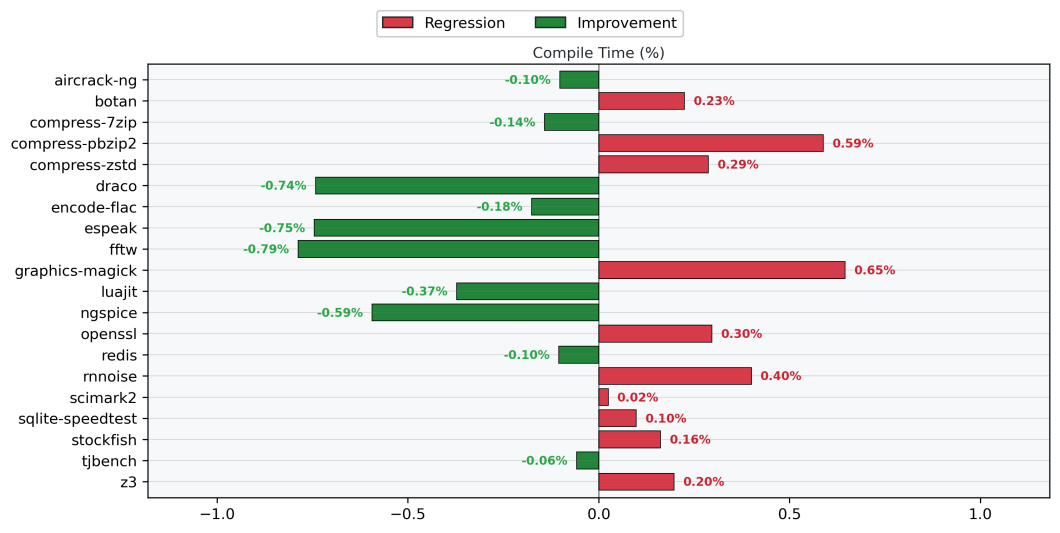
Results - Setup

- Benchmarked the implemented solution using the Phoronix Test Suite.
  - A set of 20 C/C++ applications were selected.
  - All programs compiled with the `-O3` optimization flag.

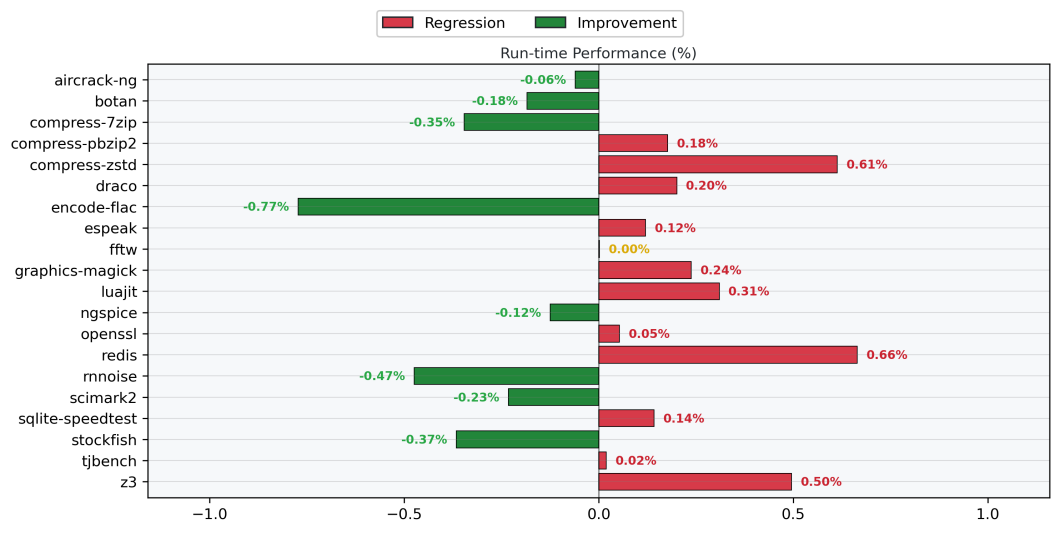
No	Benchmark	Category
1	aircrack-ng-1.3.0	Security
2	botan-1.6.0	Security
3	compress-7zip-1.11.0	Compression
4	compress-pbzip2-1.6.1	Compression
5	compress-zstd-1.6.0	Compression
6	draco-1.6.1	Texture Processing
7	encode-flac-1.8.1	Audio Compression
8	espeak-1.7.0	Speech Synthesizer
9	fftw-1.2.0	HPC
10	graphics-magick-2.2.0	Image Processing

No	Benchmark	Category
11	luajit-1.1.0	Compiler
12	ngspice-1.0.0	Circuit Simulation
13	openssl-3.3.0	Security
14	redis-1.4.0	In-Memory Database
15	rnnoise-1.1.0	Audio Processing
16	scimark2-1.3.2	Scientific Computing
17	sqlite-speedtest-1.0.1	Database
18	stockfish-1.6.0	Chess Engine
19	tjbench-1.2.0	Parallel Processing
20	z3-1.0.1	SMT Solver

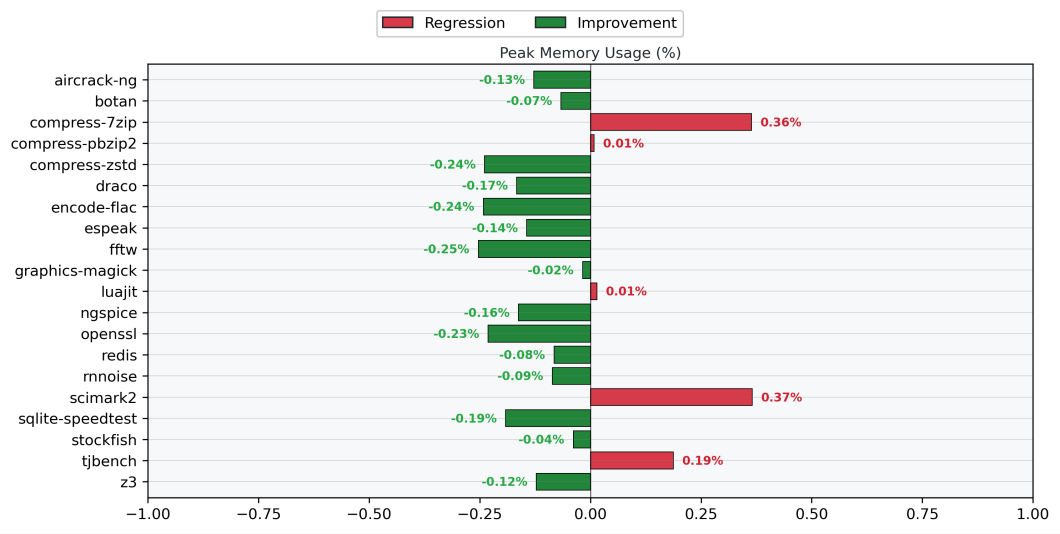
Results - Compile Time



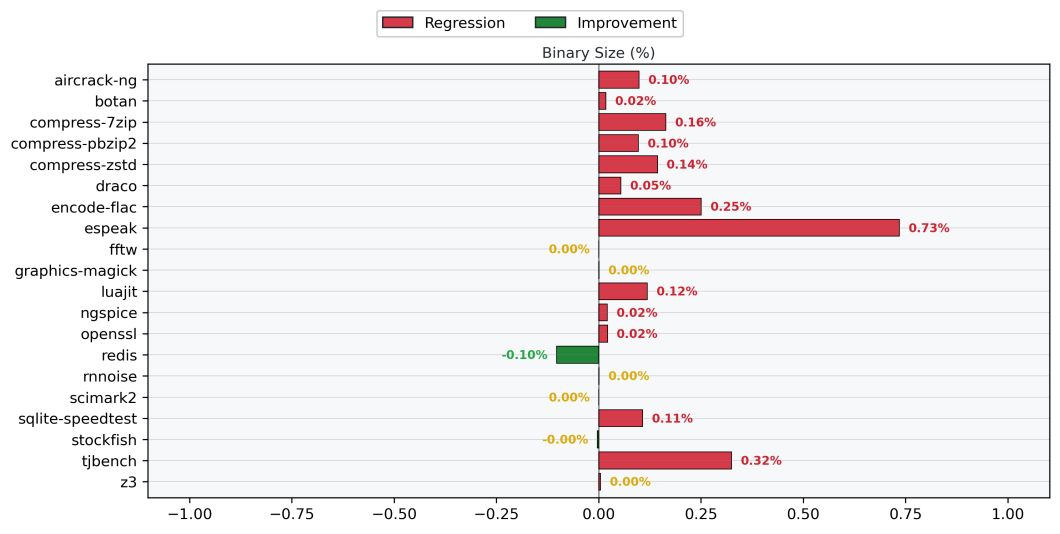
Results - Run-time Performance



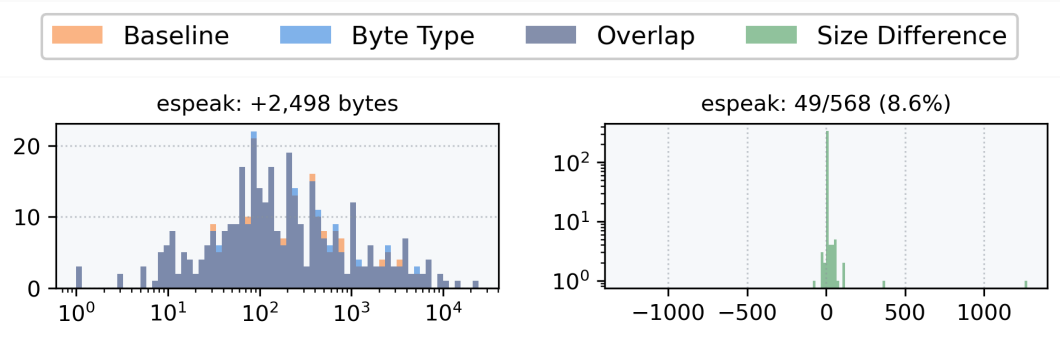
Results - Peak Memory Usage



Results - Binary Size



Results - Binary Size Distribution



Results - LLVM Test Suite

- The byte type was implemented in Alive2.
  - 11 tests in the LLVM test suite were fixed.

Results - LLVM Test Suite

- The byte type was implemented in Alive2.
  - 11 tests in the LLVM test suite were fixed.

Test	Unsoundness Reason
ExpandMemCmp/AArch64/memcmp.ll	`memcmp` to integer load/store pairs
ExpandMemCmp/X86/bcmp.ll	`bcmp` to integer load/store pairs
ExpandMemCmp/X86/memcmp-x32.ll	`memcmp` to integer load/store pairs
ExpandMemCmp/X86/memcmp.ll	`memcmp` to integer load/store pairs
GVN/metadata.ll	Unsound pointer coercions
GVN/pr24397.ll	Unsound pointer coercions
InstCombine/bcmp-1.ll	`bcmp` to integer load/store pairs
InstCombine/memcmp-1.ll	`memcmp` to integer load/store pairs
InstCombine/memcpy-to-load.ll	`memcpy` to integer load/store pairs
PhaseOrdering/swap-promotion.ll	`memcpy` to integer load/store pairs
SROA/alignment.ll	`memcpy` to integer load/store pairs

Conclusion

- Fixes longstanding issues in LLVM IR.
  - Allows optimization passes to safely represent and manipulate raw memory.
  - Enables the implementation of `memcpy`, `memmove` and `memcmp` in the IR.

Conclusion

- Fixes longstanding issues in LLVM IR.
  - Allows optimization passes to safely represent and manipulate raw memory.
  - Enables the implementation of `memcpy`, `memmove` and `memcmp` in the IR.
- Fixes unsound optimizations.
  - The new type addresses real-world miscompilations in C/C++ programs.
  - Lays a solid foundation for new optimizations.

Conclusion

- Fixes longstanding issues in LLVM IR.
  - Allows optimization passes to safely represent and manipulate raw memory.
  - Enables the implementation of `memcpy`, `memmove` and `memcmp` in the IR.
- Fixes unsound optimizations.
  - The new type addresses real-world miscompilations in C/C++ programs.
  - Lays a solid foundation for new optimizations.
- Negligible performance impact.
  - No major regressions were identified.
  - Our implementation spans 2.6k LoC (∼0.05% LoC in LLVM/Clang).