Lasagne: A Static Binary Translator for Weak Memory Model Architectures

PLDI 2022

Rodrigo C. O. Raocha, Dennis Sprokhol, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty and Pramod Bhatotia

---

PRESENTED BY: Divyanjali

Problem Statement

- An end-to-end static binary translator with precise translation rules between x86 and Arm concurrency semantics

Background: Shared Memory Concurrency

- Threads communicate through shared memory accesses
- Memory operations:
  - load (ld) that reads from memory
  - store (st) that writes to memory
  - atomic read-modify-write (`RMW`) that reads and based on the value read, writes to the memory atomically
  - fence operations that forces ordering over memory accesses

Background: Interleaving

t1	t2
x = 1	y = 1
a = y	b = x

t1	t2
x = 1	
	y = 1
	b = x
a = y	

x=1, y=1, a=1, b=1

t1	t2
x = 1	
a = y	
	y = 1
	b = x

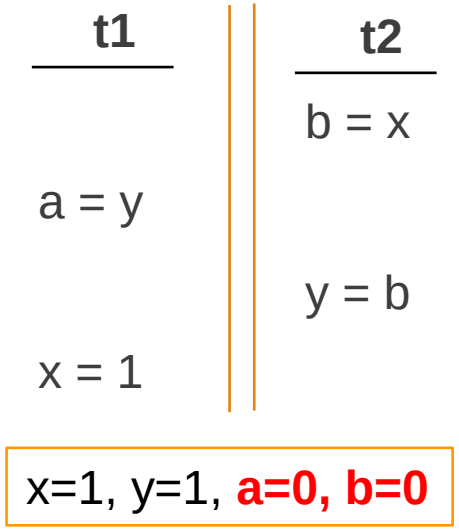
x=1, y=1, a=0, b=1

t1	t2
	b = x
	y = b
x = 1	
a = y	

x=1, y=1, a=1, b=0

Background: Weak Behaviors

Allow instruction of a thread to reorder



x=1, y=1, **a=0, b=0**

Background: Relaxed Memory Models

- Reorderings allowed may differ under different relaxed memory models
- Different relaxed memory models may have different allowed weak behaviors

$$
\begin{array}{l||l}
X = Y = 0; & \\
X = 1; & a = Y; \\
Y = 1; & b = X;
\end{array}
\qquad \text{(MP)}
$$

a=1, b=0 not allowed in x86, but allowed in Arm

SBT (Static Binary Translators)

- Three phase process:
  1. Binary Lifting: translate input program to some IR
  2. Optimize the IR
  3. Compile the optimized IR to target architecture
- Limitations of existing SBT:
  - Very limited support of several advanced architectural features
  - Unable to translate concurrent binaries

Limitations of existing SBT

- Very limited support of several advanced architectural features
- unable to translate concurrent binaries

$$\begin{array}{l} X=1; \\ Y=1; \end{array} \Big\| \begin{array}{l} a=Y; \\ b=X; \end{array}$$

(a) x86

$\xrightarrow{\text{mctoll}}$

$$\begin{array}{l} X_{\text{NA}}=1; \\ Y_{\text{NA}}=1; \end{array} \Big\| \begin{array}{l} a=Y_{\text{NA}}; \\ b=X_{\text{NA}}; \end{array}$$

(b) Unoptimized LLVM IR

$\xrightarrow{\text{opt}}$

$$\begin{array}{l} Y_{\text{NA}}=1; \\ X_{\text{NA}}=1; \end{array} \Big\| \begin{array}{l} a=Y_{\text{NA}}; \\ b=X_{\text{NA}}; \end{array}$$

(c) Optimized LLVM IR

$\xrightarrow{\text{codegen}}$

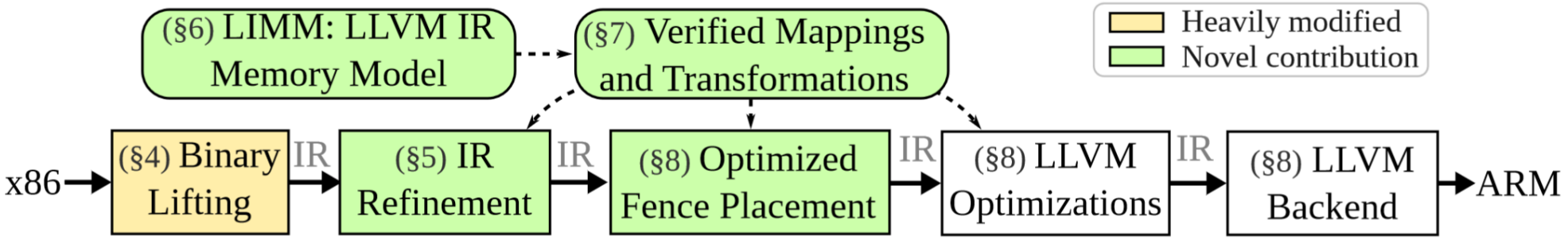
$$\begin{array}{l} Y=1; \\ X=1; \end{array} \Big\| \begin{array}{l} a=Y; \\ b=X; \end{array}$$

(d) Arm

Proposed Solution

- Memory Model at IR level (LIMM) that
  - Allows precise mapping from source to IR to target
  - Allows common optimizations on IR
- Translation tool requirements:
  - Reason about the consistency models for correct and efficient translation
  - Support for source instruction set

Overview


(§6) LIMM: LLVM IR Memory Model
(§7) Verified Mappings and Transformations
Heavily modified
Novel contribution
x86
(§4) Binary Lifting
IR
(§5) IR Refinement
IR
(§8) Optimized Fence Placement
IR
(§8) LLVM Optimizations
IR
(§8) LLVM Backend
ARM

Binary Lifting

- Lift binary to LLVM IR
- Contributions:
  - add support for floating-point arguments/return types and tail-calls
  - add support for around 100 instructions (400 instruction variants), mainly SSE instructions
  - add support for some of the processor status flag
  - implement additional x86 features such as global variables declared in header files
  - fix several bugs discovered when lifting highly optimized programs, e.g., using –O3

Binary Lifting: Methodology

- Function Type Discovery
  - Find function's parameters, return types
  - Based on the calling convention
- Instruction Translation
  - Each machine instruction can be translated to zero, one, or more LLVM instructions
  - Any unnecessary lifted instructions become dead code and are eliminated by traditional LLVM optimizations
- Translating Function Calls
  - Use information from function type discovery
  - Find function arguments and return value based on calling convention
  - Variadic Functions: all parameter registers alive at the callsite are passed an arguments

Binary Lifting: Methodology

- SSE Register Values
  - Based on bit width of source and destination types
- Stack Memory
  - Using byte array with translated indexing for stack offsets

IR Refinements (*PPOpt*)

- Challenges:
  - Machine code does not differentiate between integer and pointer types
  - Machine code instructions do not dictate the type of a pointer
- Solution:
  - Use `inttoptr` and `ptrtoint` to figure out the type
  - Peephole optimizations to optimize interger-based address computation
  - Integer parameters used only as input to `inttoptr` are converted to pointers

Peephole Optimizations

Rule 1: Pointer casting

```
%0 = ptrtoint i8* %stacktop to i64
%RBP = inttoptr i64 %0 to i32*
=>
%RBP = bitcast i8* %stacktop to i32*
```

Rule 2: Stack offset

```
%tos = ptrtoint i8* %stacktop to i64
%0 = add i64 %tos, 16
%RBP = inttoptr i64 %0 to i32*
=>
%0 = getelementptr i8, i8* %stacktop, i64 16
%RBP = bitcast i8* %0 to i32*
```

Rule 3: Parameter offset

```
%0 = add i64 %arg, 8
%RBP = inttoptr i64 %0 to i32*
=>
%0 = inttoptr i64 %arg to i8*
%1 = getelementptr i8, i8* %0, i64 8
%RBP = bitcast i8* %1 to i32*
```

Promoting Pointer Parameters

- Collect all uses of each integer parameter
- If all its users are `inttoptr` instructions, mark it for a pointer type promotion
- Pointer type will depend on all the destination pointer types of the `inttoptr` instructions
- If all of them have the same destination pointer type, promote it to that type and remove all `inttoptr` instructions
- Else promote it to an i8 pointer, replace all the `inttoptr` instructions to a bitcast

Basic Notations

- Notations
  - Events: represented by ⟨id,tid,lab⟩,
    - Where lab = ⟨op,loc,val⟩
  - Relations:
    - $S^{?}$, S^{+}, S^{*}, S^{-1}, S_{imm}, $[A]$, $S1;S2$
    - po
    - rf
    - co
    - rmw
    - fr
  - Executions

Common Axioms in x86 and Arm

- Coherence: $(po|_{loc} \cup rf \cup co \cup fr)^+$ is irreflexive. (sc-per-loc)
- Atomicity: $rmw \cap (fre;coe) = \emptyset$ (atomicity)

x86 Axioms

x86 axiom

(GHB) hb^{+} is irreflexive where

$$ppo \triangleq ((W \times W) \cup (R \times W) \cup (R \times R)) \cap po$$

$$implid \triangleq po;[At \cup F] \cup [At \cup F];po$$

where $At \triangleq dom(rmw) \cup codom(rmw)$

$$hb \triangleq ppo \cup implid \cup rfe \cup fr \cup co$$

Arm Axioms

Arm axiom

(external) ob is irreflexive where

$ob \triangleq (obs \cup aob \cup dob \cup bob)^+$ where

$obs \triangleq rfe \cup coe \cup fre$

$aob \triangleq rmw \cup \ldots$

$dob \triangleq addr \cup data \cup ctrl;[W] \cup \ldots$

$bob \triangleq po;[F];po \cup [R];po;[F_{LD}];po \cup [W];po;[F_{ST}];po;[W] \cup \cdots$

IR Concurrency Model (LIMM)

- Already exist: Rna, Wna, RMWsc, Rsc, Wsc, Fsc
- Introduce fence Frm and Fww into LLVM IR
  - Frm: order load with memory accesses after it
  - Fww: order store pairs

$$(\mathrm{po}|_{\mathrm{loc}} \cup \mathrm{rf} \cup \mathrm{fr} \cup \mathrm{co}) \text{ is acyclic.} \qquad \text{(sc-per-loc)}$$

$$\mathrm{rmw} \cap (\mathrm{fre};\mathrm{coe}) = \emptyset. \qquad \text{(atomicity)}$$

$$\mathrm{ghb} \text{ is irreflexive where} \qquad \text{(GOrd)}$$

$$\mathrm{ghb} \triangleq (\mathrm{ord} \cup \mathrm{rfe} \cup \mathrm{coe} \cup \mathrm{fre})^{+} \text{ where}$$

$$\mathrm{ord} \triangleq [\mathrm{R}];\mathrm{po};[\mathrm{F_{RM}}];\mathrm{po};[\mathrm{R} \cup \mathrm{W}] \qquad (\mathrm{ord}_1)$$

$$\cup\ [\mathrm{W}];\mathrm{po};[\mathrm{F_{ww}}];\mathrm{po};[\mathrm{W}] \qquad (\mathrm{ord}_2)$$

$$\cup\ [\mathrm{F_{SC}} \cup \mathrm{R_{SC}} \cup \mathrm{codom}(\mathrm{rmw})];\mathrm{po} \qquad (\mathrm{ord}_3)$$

$$\cup\ \mathrm{po};[\mathrm{F_{SC}} \cup \mathrm{W_{SC}} \cup \mathrm{dom}(\mathrm{rmw})] \qquad (\mathrm{ord}_4)$$

Translation

x86		IR		Arm
ld	→	ld_{NA};Frm	→	ld;DMBLD
st	→	Fww;st_{NA}	→	DMBST;st
RMW	→	RMW_{SC}	→	DMBFF;RMW;DMBFF
MFENCE	→	Fsc	→	DMBFF

(c) x86 to IR to Arm

Correctness Guarantees

Theorem 7.1 (Mapping Correctness). *Let* $M_s \rightarrow M_t$ *be a mapping scheme which generates target program* $\mathbb{P}_t$ *from the source program* $\mathbb{P}_s$. *The scheme is correct if for each consistent target execution* $X_t \in [[\mathbb{P}_t]]_{M_t}$ *there exists a consistent source execution* $X_s \in [[\mathbb{P}_s]]_{M_s}$ *such that* $\mathrm{Behav}(X_t) = \mathrm{Behav}(X_s)$.

Theorem: *The mapping scheme from x86 to IR and IR to Arm are precise*

Optimizing Transformations (*Popt*)

- Reorderings
- Memory Access Eliminations
- Speculative Load Introduction
- Fence Merging: $F_{rm} \cdot F_{ww} \rightarrow F_{sc} \cdot F_{sc} \rightarrow F_{sc}$

Give proof of correctness of these tranformations

Reorderings

$\downarrow a \setminus b \rightarrow$	R_{NA}	W_{NA}	R_{SC}	$R_{SC} \cdot W_{SC}$	F_{RM}	F_{WW}	F_{SC}
R_{NA}	✓	✓	✓	✗	✗	✓	✗
W_{NA}	✓	✓	✓	✗	✓	✗	✗
R_{SC}	✗	✗	✗	✗	✓	✓	✓
$R_{SC} \cdot W_{SC}$	✗	✗	✗	✗	✓	✓	✓
F_{RM}	✗	✗	✗	✓	=	✓	✓
F_{WW}	✓	✗	✓	✓	✓	=	✓
F_{SC}	✗	✗	✗	✓	✓	✓	=

(a) Reorderings $a \cdot b \rightsquigarrow b \cdot a$.

Memory Access Eliminations

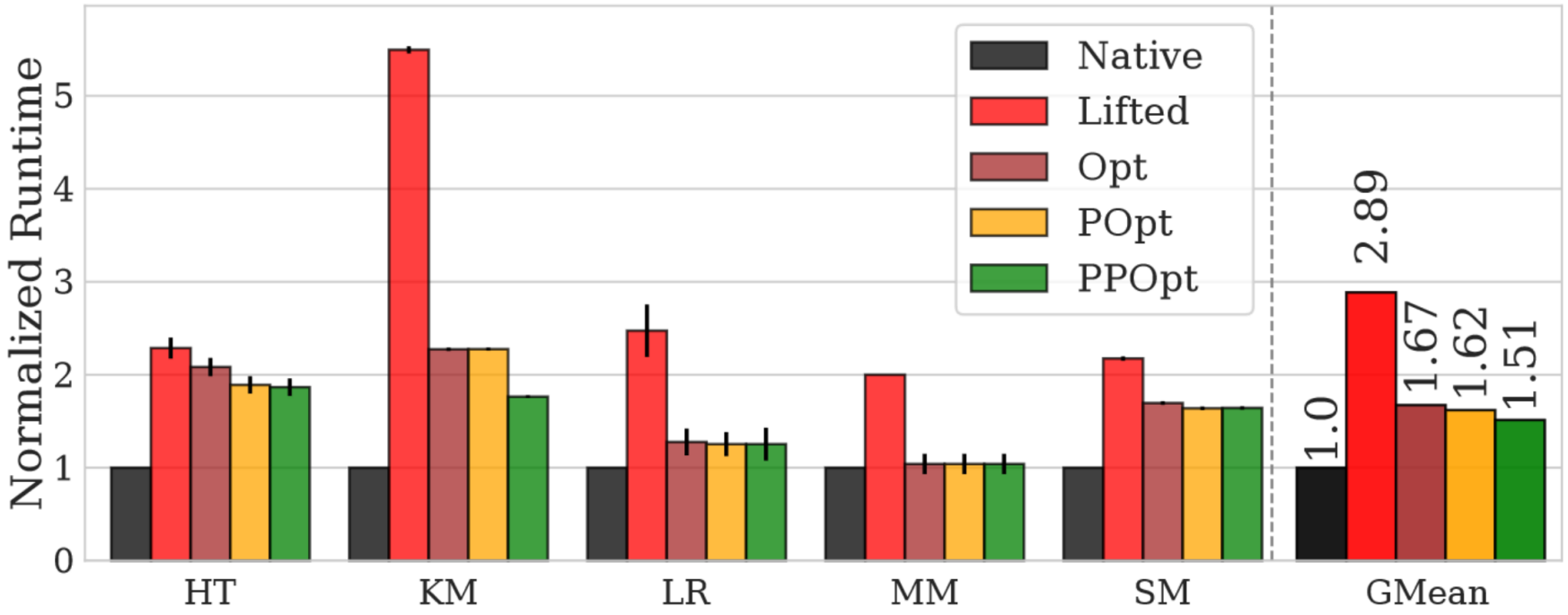
$$\mathrm{R}(X,v)\cdot\mathrm{R}(X,v')\rightsquigarrow\mathrm{R}(X,v) \qquad \text{(RAR)}$$
$$\mathrm{W}(X,v)\cdot\mathrm{R}(X,v)\rightsquigarrow\mathrm{W}(X,v) \qquad \text{(RAW)}$$
$$\mathrm{W}(X,v)\cdot\mathrm{W}(X,v')\rightsquigarrow\mathrm{W}(X,v') \qquad \text{(WAW)}$$
$$\mathrm{R}(X,v)\cdot\mathrm{F}_o\cdot\mathrm{R}(X,v')\rightsquigarrow\mathrm{R}(X,v)\cdot\mathrm{F}_o \qquad \text{(F-RAR)}$$
$$\mathrm{W}(X,v)\cdot\mathrm{F}_\tau\cdot\mathrm{R}(X,v)\rightsquigarrow\mathrm{W}(X,v)\cdot\mathrm{F}_\tau \qquad \text{(F-RAW)}$$
$$\mathrm{W}(X,v)\cdot\mathrm{F}_o\cdot\mathrm{W}(X,v')\rightsquigarrow\mathrm{F}_o\cdot\mathrm{W}(X,v') \qquad \text{(F-WAW)}$$

(b) Eliminations where $o\in\{\text{RM},\text{WW}\}$ and $\tau\in\{\text{SC},\text{WW}\}$.

Implementing LIMM

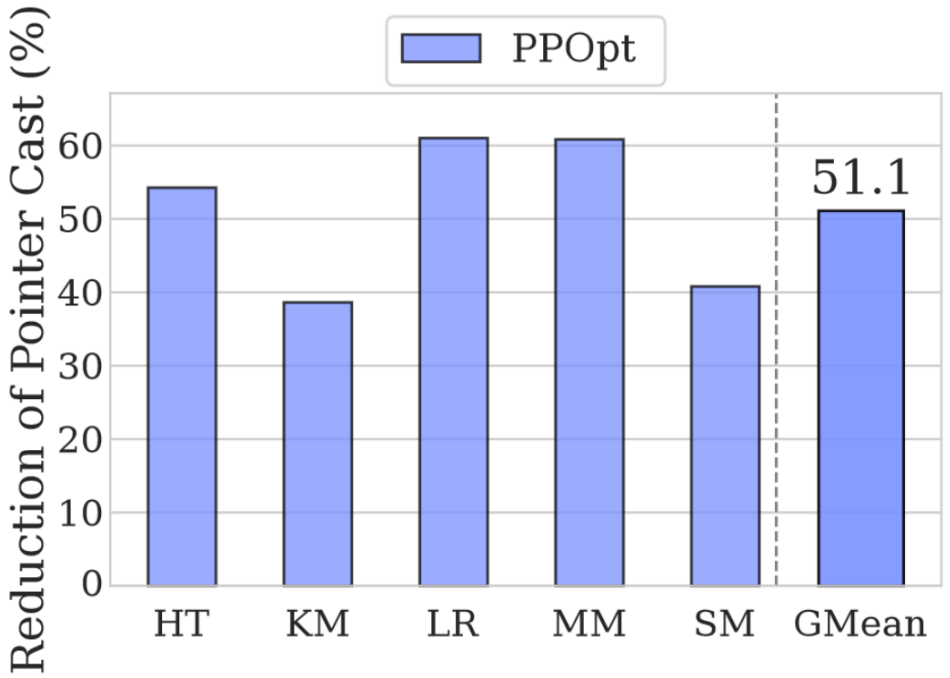
- From x86 to IR:
  - For every ld and st, explore the use-def chain of their pointer operand.
  - If the access is performed on a stack address, then no fence is inserted.
  - Else fences are inserted
  - merge pairs of fences
- Perform LLVM optimizations
- Convert the IR to Arm

Performance over Various Optimizations



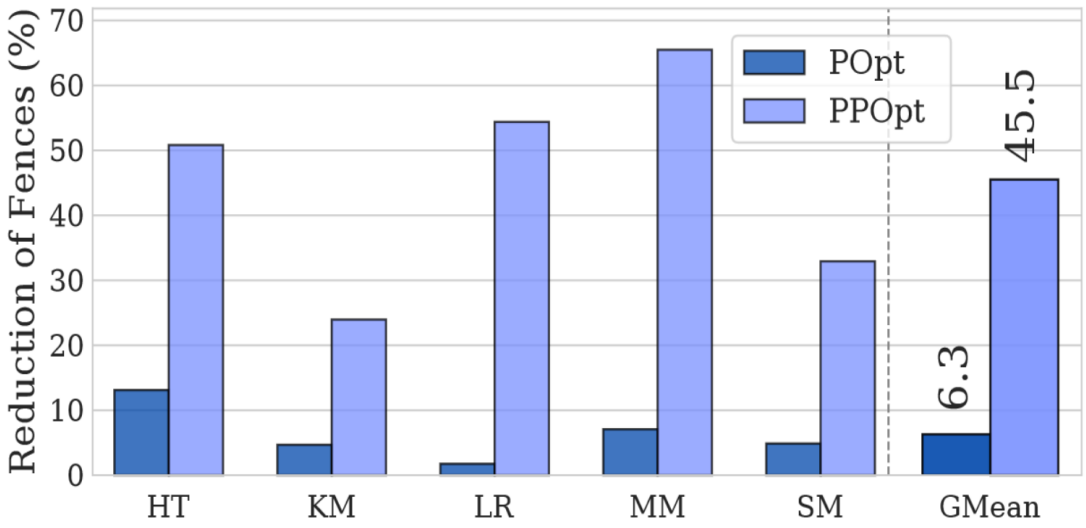
- *Native:* Complied from C source code to Arm binary
- *Lifted:* x86 binary to Arm (no optimizations)
- *Opt: Lifted* with optimizations of lifted LLVM IR
- *Popt: Opt* with fence merging rules
- *PPOpt*: *POpt* with IR refinement

Impact of Peephole Optimizations

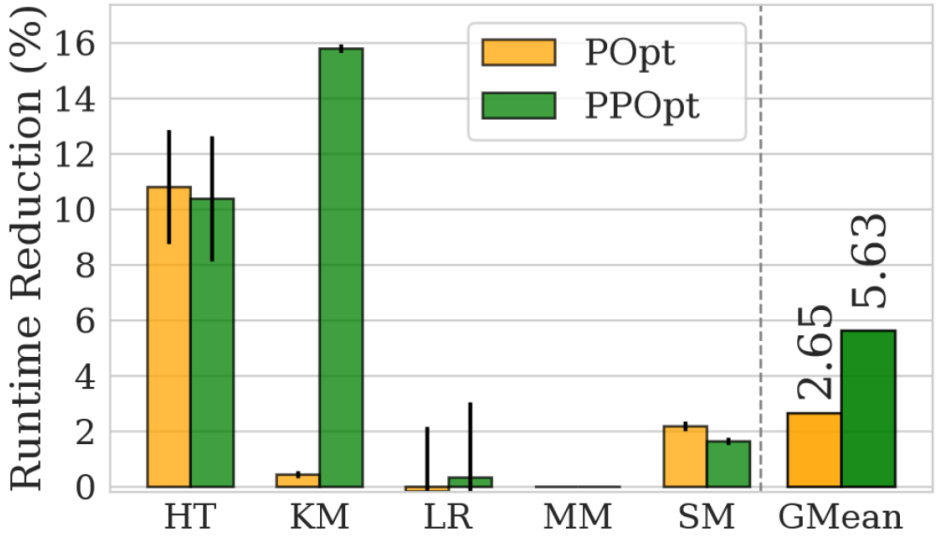


- Result of removing `inttoptr` and `ptrtoint` through peephole optimizations proposed.

Impact of fence placement optimizations

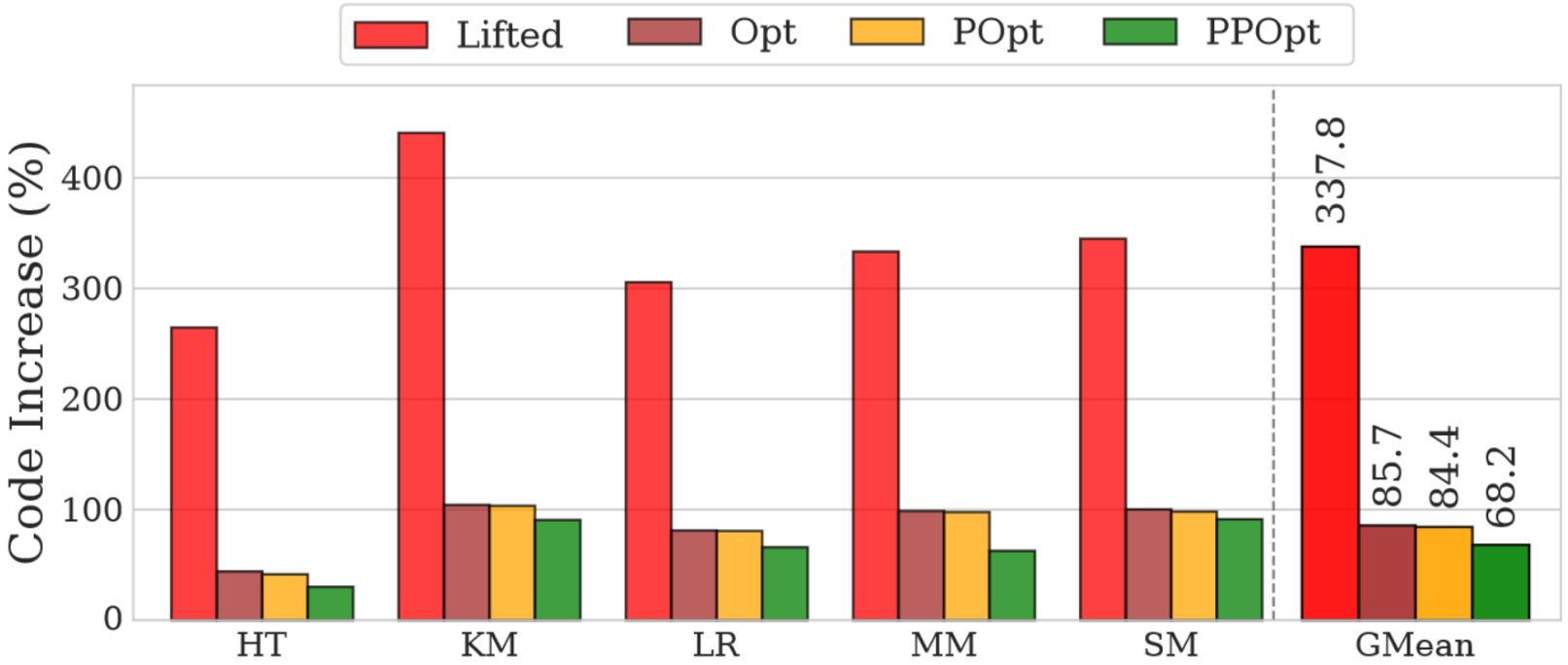


- Reduction in #fences w.r.t unoptimized lifted code (*Lifted*)
- IR refinements (*PPOpt*) allow the fence placement algorithm to avoid adding fences to operations involving the stack memory



- Performance improvement by reducing #fences w.r.t unoptimized lifter code (*Lifted*)

Comparison of Code Size



- Increase in code side w.r.t *Native* code

Summary

- Lift binary to IR
- IR must include some primitives to express memory ordering semantics of the source architecture
- Perform Optimizations
- Convert IR to target architecture

Insights

- This approach works so well for x86 and Arm because they are quite similar
- To translate binary from x86 to some architecture like Power, we need many more types of fences, loads and stores in IR and their translation to Power
- Such precise translation may be hard to come up with

- The optimizations proposed, specially peephole optimization seem to be generic enough
- Reordering optimization need to be redefined for different architecture as per the added memory model primitives in IR
- Memory Access Eliminations, fence merging rules and proof of speculative load introduction is not generic enough