MLGOPerf: An ML Guided Inliner to Optimize Performance

Amir H. Ashouri, Mostafa Elhoushi, Yuzhe Hua, Xiang Wang, Muhammad Asif Manzoor, Bryan Chan, Yaoqing Gao

Heterogeneous Compiler Lab
Toronto, Canada
Function Inlining Optimization

**Pros:**
- Reduces overhead due to entering and exiting functions.
  - Eliminates the instruction required to function calling
  - Not needing registers to pass arguments (reduces register spilling)
- Opens opportunities to subsequent optimizations
  - e.g., Constant propagation, hoisting out part of the function in LICM, expand the scope of register allocation

**Cons:**
- Increases code size
- Larger code size reduces the temporal locality
  - Thus, decreases the performance of the instruction cache
Factors to Consider When Inlining

- **Callee Size:**
  
  - Usually it is better to inline smaller functions and leave the larger ones as calls.

- **Hotness of Call-Site:**
  
  - A call to a large function which is executed frequently could be more beneficial to inline than a call to a small function that is rarely reached.

LLVM has many magic numbers and heuristics to compute the profitability of inlining a callsite.

Include/llvm/analysis/inlinecost.h

```c
// Various magic constants used to adjust heuristics.
int getInstrCost();
const int IndirectCallThreshold = 100;
const int LoopPenalty = 25;
const int LastCallToStaticBonus = 15000;
const int ColdccPenalty = 2000;
/// Do not inline functions which allocate this many bytes on the stack
/// when the caller is recursive.
const unsigned TotalAllocSizeRecursiveCaller = 1024;
/// Do not inline dynamic allocas that have been constant propagated to be
/// static allocas above this amount in bytes.
const uint64_t MaxSimplifiedDynamicAllocToInline = 65536;
```
Inline optimization

Before Inline

**Caller**

```c
int func(int y)
{
    return pred(y) + pred(0) + pred(y+1);
}
```

3 Callsites

**Callee**

```c
int pred(int x)
{
    if (x == 0)
        return 0;
    else
        return x - 1;
}
```

**Calls**

After Inline

```c
int func(int y)
{
    int tmp;
    if (y == 0) tmp = 0; else tmp = y - 1; /* (1) */
    if (y == 0) tmp += 0; else tmp += 0 - 1; /* (2) */
    if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1; /* (3) */
    return tmp;
}
```

In LLVM, inline optimization is done at IR level
O3’s Function Inlining

1. **At lower optimization levels**, function inlining can be controlled in a number of ways, i.e.,
   - explicit use of *always_inline* attribute
   - command-line argument:
     - `--always-inline/disable-inlining`, etc.

2. **At O3**, it is enabled by default and is used only once early in the pipeline. This will ensure more opportunities are produced for latter passes in the pipeline.

3. **At Link-time-optimization**, inlining can also be used to provide “intermodular optimizations”. This aspect is outside the scope of MLGOPerf.
Inlining in LLVM

Inlining is done both at IR and at Linking time:

1. **IR (Intra-procedural Inlining):**
   - Build call-graph for the whole module
   - Iterate call graph bottom-up
     - For each call-site:
       - Simulate the effect of inlining
       - Using some heuristics, estimate the \textit{cost} of inlining
       - Using other heuristics (e.g., how hot is the call-site), estimate the \textit{threshold}
       - if (cost > threshold) ? Inline : NoInline

2. **Linking (Interprocedural Inlining)**
   - Using LTO – Outside the scope of this project
Google’s MLGO

- In April 2020, Google released their first version of MLGO, an ML-based inliner which was later upstreamed into LLVM\cite{1}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{mlgo.png}
\caption{MLGO Overview}
\end{figure}

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|}
\hline
 & PG & ES & ES (L) \\
\hline
Size Reduction & 4.95\% & 3.74\% & 5.94\% \\
Parallelism in Data Collection & 100 & 488 & 488 \\
Training Time & \sim12h & \sim60h & \sim150h \\
\hline
\end{tabular}
\caption{Table 2. Policy Gradient v.s. Evolution Strategies}
\end{table}

\cite{1} https://reviews.llvm.org/D77752?id=256104#inline-710816
MLGOPerf Contributions

1. We extended Google’s MLGO to target *performance* rather than code-size reduction.

2. MLGOPerf employs two ML models, first of which predicts the function speedup post-inlining to help generate the rewards needed to train the second for which makes the decision to whether or not to inline a callsite inside LLVM’s function inlining.

3. We trained the first model, i.e., IR2Perf, by leveraging our autotuner to generate +300k meaningful inlining configurations using SPEC CPU2006 on aarch64. We do so by generating 20 handcrafted features we designed and tested.

4. We leveraged IR2Perf to train the second model, our RL agent, for more than a million iterations in a matter of few days which otherwise wouldn’t be possible without.
MLGO Class Hierarchy
High-level Flow

```c
void caller() {
    statement1;
    statement2;
    statement3;
    function1();
    callee();
    for(int i=0; i<10; i++)
        for(int j=0; j<5; j++)
            statement;
    ...
}
```

**Table 1: MLGOPerf Phases**

<table>
<thead>
<tr>
<th>Phases</th>
<th>IR2Perf</th>
<th>RL Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR2Perf Training</td>
<td>Training</td>
<td>-</td>
</tr>
<tr>
<td>RL Model Training</td>
<td>Inference</td>
<td>Training</td>
</tr>
<tr>
<td>MLGOPerf Deployment</td>
<td>-</td>
<td>Inference</td>
</tr>
</tbody>
</table>
MLGOPerf Three Phases

(1) MLGOPerf::IR2PerfTraining

- Autotuner
- Function Features
- Experiment Stats
- Source.cpp
- Inliner
- Data Preprocessing
- IR2Perf Model
- Deep Regression Model
- Feedback/Stats
- Finegrained Tuning at Inlining Callsites
- Collect FTs
- opt -O3 pipeline
- Subsequent passes of -O3
- Binary Execution

(2) MLGOPerf::RLTraining

- Source.cpp
- Inliner
- Collect FTs
- Training Model
- Call() FTs
- Reward
- Function Features
- Deep Regression Model
- Finegrained Tuning at Inlining Callsites
- Experiment Stats
- Measurements

(3) MLGOPerf::Deployment

- Source.cpp
- Inliner
- Collect FTs
- Call() FTs
- Deployed Model
- Inlining Callsite?
- Y/N
- RL Release
- Binary Execution
- Function Features
- Feedback/Stats
- Finegrained Tuning at Inlining Callsites
- Measurements
- Autotuner
- Source.cpp
- Inliner
- Data Preprocessing
- IR2Perf Model
- Deep Regression Model
- Finegrained Tuning at Inlining Callsites
- Experiment Stats
IR2Perf (1/2)
Post-inlining Function Speedup Prediction

Table 2: IR2Perf Model Features

<table>
<thead>
<tr>
<th>No</th>
<th>Static Feature Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>InstructionPerBlock</td>
</tr>
<tr>
<td>2</td>
<td>SuccessorPerBlock</td>
</tr>
<tr>
<td>3</td>
<td>AvgNestedLoopLevel</td>
</tr>
<tr>
<td>4</td>
<td>InstrPerLoop</td>
</tr>
<tr>
<td>5</td>
<td>BlockWithMultipleSuccersPerLoop</td>
</tr>
<tr>
<td>6</td>
<td>CallsNo</td>
</tr>
<tr>
<td>7</td>
<td>IsLocal</td>
</tr>
<tr>
<td>8</td>
<td>MaxLoopDepth</td>
</tr>
<tr>
<td>9</td>
<td>MaxDomTreeLevel</td>
</tr>
<tr>
<td>10</td>
<td>CallerHeight</td>
</tr>
<tr>
<td>11</td>
<td>CallUsage</td>
</tr>
<tr>
<td>12</td>
<td>IsRecursive</td>
</tr>
<tr>
<td>13</td>
<td>NumCallsiteInLoop</td>
</tr>
<tr>
<td>14</td>
<td>EntryBlockFreq</td>
</tr>
<tr>
<td>15</td>
<td>MaxCallsiteBlockFreq</td>
</tr>
<tr>
<td>16</td>
<td>NoOfInstructions='Ret'</td>
</tr>
<tr>
<td>17</td>
<td>NoOfInstructions='fmul'</td>
</tr>
<tr>
<td>18</td>
<td>NoOfInstructions='fdiv'</td>
</tr>
<tr>
<td>19</td>
<td>NoOfInstructions='fadd'</td>
</tr>
<tr>
<td>20</td>
<td>NoOfInstructions='fsub'</td>
</tr>
</tbody>
</table>

Table 3: IR2Perf Model Architecture

<table>
<thead>
<tr>
<th>Layer No</th>
<th>Layer (type)</th>
<th>Output Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Linear-FC1</td>
<td>[-1, 1, 128]</td>
</tr>
<tr>
<td></td>
<td>Leaky_Relu-1</td>
<td>[-1, 1, 128]</td>
</tr>
<tr>
<td>2</td>
<td>Linear-FC2</td>
<td>[-1, 1, 256]</td>
</tr>
<tr>
<td></td>
<td>Leaky_Relu-2</td>
<td>[-1, 1, 256]</td>
</tr>
<tr>
<td>3</td>
<td>Linear-FC3</td>
<td>[-1, 1, 32]</td>
</tr>
<tr>
<td></td>
<td>Leaky_Relu-3</td>
<td>[-1, 1, 32]</td>
</tr>
<tr>
<td>4</td>
<td>Linear-FC4</td>
<td>[-1, 1, 1]</td>
</tr>
</tbody>
</table>

Table 4: IR2Perf Cross Validation Accuracy

<table>
<thead>
<tr>
<th>NO.</th>
<th>Benchmark</th>
<th>Prediction Error (MSE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>401_bzip2</td>
<td>9.1%</td>
</tr>
<tr>
<td>2</td>
<td>456_hmmer</td>
<td>9.5%</td>
</tr>
<tr>
<td>3</td>
<td>462_libquantum</td>
<td>15.7%</td>
</tr>
<tr>
<td>4</td>
<td>464_h264ref</td>
<td>19.1%</td>
</tr>
<tr>
<td>5</td>
<td>445_gobmk</td>
<td>17.5%</td>
</tr>
<tr>
<td>6</td>
<td>470_lbm</td>
<td>9.8%</td>
</tr>
<tr>
<td>7</td>
<td>458_sjeng</td>
<td>13.5%</td>
</tr>
<tr>
<td>8</td>
<td>429_mcf</td>
<td>12.2%</td>
</tr>
<tr>
<td>9</td>
<td>433_milec</td>
<td>13.9%</td>
</tr>
<tr>
<td>10</td>
<td>482_sphinx3</td>
<td>11.2%</td>
</tr>
<tr>
<td></td>
<td>Geometric Mean</td>
<td>12.8%</td>
</tr>
</tbody>
</table>
IR2Perf (2/2)
Post-inlining Function Speedup Prediction

(a) Training Loss
(First epoch)
Reinforcement Learning

RL is a class of Machine Learning which tries to find an optimal policy for a Markov Decision Process (MDP) that is defined by the tuple of \(< S, A, T, R >\) where \(S\) is the state space, \(A\) represents action space, \(R\) is the reward function that an agent receives by doing action \(a\) from state \(s\) to \(s'\), and \(T\) is the transition probability at time \(t\) from state \(s\) to \(s'\):

\[
T_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)
\]

1. **State** \(S\): Current visiting call site
2. **Action** \(A\): Defines a bool variable whether or not to inline the call site
3. **Transition** \(T\): A deterministic function in our context which updates the call graph upon taking an action over the current call site and switching to visit the next call site
4. **Reward** \(R\): In this work, it is defined as the function execution speedup with respect to the baseline
Training RL Agent

Algorithm 1 Training Inliner RL Model using IR2Perf

1: procedure FUNCTION_SPEEDUP \quad \text{\textit{\# IR2Perf Inference}}
2: \hspace{1em} for Function f in Module do
3: \hspace{2em} FTs $\leftarrow$ getFunctionFeatures()
4: \hspace{2em} funcReward $\leftarrow$ infer(FTs)
5: \hspace{2em} totalReward $\leftarrow$ append(funcReward)
6: \hspace{1em} end for
7: return totalReward
8: end procedure

9:

10: procedure CALLSITE_INLINE \quad \text{\textit{\# RL Model Training}}
11: \hspace{1em} initialize policy $\pi_Y$ randomly
12: \hspace{1em} for iteration i in Training do
13: \hspace{2em} $s_i \leftarrow \text{Sample}_{N(0,1)}(TrainingData)$
14: \hspace{2em} Compile and Get IR with policy $\pi_Y + \sigma s_i$
15: \hspace{2em} $R \leftarrow$ FUNCTION_SPEEDUP(Module)
16: \hspace{2em} Update policy $\gamma$ for using Equation 1
17: \hspace{1em} end for
18: end procedure
1. Results show that on average MLGOPerf is able to outperform O3 by 1.8% and 2.2% on SPEC CPU2006 and Cbench when tested on aarch64.

2. There is a slight increase in the code size of the optimized binaries MLGOPerf generates, and that is measured as 12% and 16% against LLVM’s O3 and MLGO on Cbench and 17.8% and 23.4% for SPEC CPU2006.
Additionally, MLGOPerf provides more opportunities for subsequent optimization passes, i.e., loop unroll and loop vectorize, and an autotuning experiment reveals we can gain at a faster rate and up to 3.7% improvement with respect to O3.
Cbrench Example (1/2)

```bash
> make clean && $(clang_mlgoperf)/clang *.c -03 -mllvm -debug-only=inline -fexperimental-new-pass-manager -lm
rm -f out/*.

Inlining calls in: sha_update
    Inlining (cost=285, threshold=375), Call: call fastcc void @byte_reverse(i64*nonnull %arraydecay)
    NOT Inlining (cost=625, threshold=625), Call: call fastcc void @sha_transform(%struct.SHAINFO* %sha_info)
Updated inlining SCC: (sha_update)

Inlining calls in: sha_final
    Inlining (cost=285, threshold=375), Call: call fastcc void @byte_reverse(i64*nonnull %arraydecay)
    NOT Inlining (cost=625, threshold=625), Call: call fastcc void @sha_transform(%struct.SHAINFO* %sha_info)
    NOT Inlining (cost=14715, threshold=375), Call: call fastcc void @byte_reverse(i64*nonnull %arraydecay)
    NOT Inlining (cost=625, threshold=625), Call: call fastcc void @sha_transform(%struct.SHAINFO* %sha_info)
Updated inlining SCC: (sha_final)

Inlining calls in: sha_stream
    Inlining (cost=-5, threshold=375), Call: call void @sha_init(%struct.SHAINFO* %sha_info)
    NOT Inlining (cost=420, threshold=250), Call: call void @sha_update(%struct.SHAINFO* %sha_info, i8*nonnull %arraydecay, i32 %conv)
    NOT Inlining (cost=625, threshold=625), Call: call void @sha_final(%struct.SHAINFO* %sha_info)
Updated inlining SCC: (sha_stream)

> flush-cache && numactl --localalloc --physcpubind=5 ./run 1
Flushing page cache, directories and inodes...
Done flushing.
Found dataset: 1
Command line: ../office_data/1.txt > ftmp_out
Loop wrap: 321506

real 0m8.367s
user 0m7.334s
sys 0m0.946s
```

```c
/* compute the SHA digest of a FILE stream */

#define BLOCK_SIZE 8192

void sha_stream(SHA_INFO *sha_info, FILE *fin)
{
    int i;
    BYTE data[BLOCK_SIZE];
    sha_init(sha_info);
    while ((i = fread(data, 1, BLOCK_SIZE, fin)) > 0) {
        sha_update(sha_info, data, i);
    }
    sha_final(sha_info);
}
```
• In this test case, MLGOPerf provides around 16% speedup wrt O3 (on an X86 machine).

• Exhaustive autotuning reveals that the optimal solution was not to inline the final callsite, \texttt{call void @sha\_final}.
  • 7% missed additional performance gain wrt optimal solution
Challenges

1. **Function vs. Global Speedup**
   - They don’t necessarily agree with each other

2. **Multi-objective Optimization**
   - Code-size vs. performance

3. **Compiler Optimization Space**
   - NP-hard problem

4. **Compiler Performance Prediction**
   - Noisy measurements

5. **Software Characterization**
   - Static vs dynamic
   - handcrafted vs automatic generated embedding
Thank You!

- The full-length version is at arXiv\textsuperscript{[1]}.
- The short version is to be appeared in ACM/IEEE CASES 2022.
- Any questions, feel free to contact me at: amirh.ashouri@huawei.com