Sprint1-team9-progress

Branch Optimization(홍순범)

true branch가 false branch보다 더 cost가 큰 것을 보고 true branch로 갈 확률이 높은 경우에 false branch로 갈 수 있게 하는 optimization을 진행하였다. 그 중에서도 loop에서 condition을 체크 후 true branch를 통해 다시 loop로 가는 것을 보고 false branch를 통해 loop로 들어갈 수 있게 하는 optimization pass를 만들었다.

[Plan]

1. llvm파일에서 loop info 파악
2. icmp 기존 조건과 반대조건으로 적용
3. Swap branch

[Progress]

1. loop info 파악

```
for (auto &L : LI) {
  BasicBlock *Header = L->getHeader();
  BranchInst *BI = dyn_cast<BranchInst>(Header->getTerminator());

  if (BI && BI->isConditional()) {
    ICmpInst *CI = dyn_cast<ICmpInst>(BI->getCondition());
```

2. branch가 어디로 갈지 확인

```
bool shouldInvertBranch(BranchInst *BI, FunctionAnalysisManager &FAM) {
  // using llvm analysis which provides branch probability information
  // https://llvm.org/doxygen/classllvm_1_1BranchProbabilityInfo.html
  auto &BPI = FAM.getResult<BranchProbabilityAnalysis>(*BI->getFunction());

  // get probability of true branch
  BranchProbability Prob =
      BPI.getEdgeProbability(BI->getParent(), BI->getSuccessor(0));
  BranchProbability threshold(1, 4); // 25%
  return Prob > threshold;          // return true if (true branch > 25%)
}
```

3. icmp 조건 뒤집기

```
// Invert the condition
ICmpInst::Predicate NewPred =
    ICmpInst::getInversePredicate(CI->getPredicate());
CI->setPredicate(NewPred);
```

4. branch 뒤집기

```
// Swap branch
BI->swapSuccessors();
```

원래 계획은 loop info만 파악 후 모든 loop의 branch를 swap하려 했다. 진행과정에서 llvm analysis가 각각의 branch로 갈 확률을 제공해준다는 것을 알게 되어 true cost : false cost = 3:1에 따라 true branch로 갈 확률이 25% 이상인 것들만 swap하는 것으로 세부 구현은 수정하였다. 여기서 사용된 branch analysis는 추후 다른 모든 branch에 적용가능할 것으로 보인다.

[Result]

그리고 모든 벤치마크를 통해 전반적인 성능평가를 진행하였고 cost의 감소를 확인하였다.

	before	after
Bitcount1	2,106	1,301
Bitcount4	1,121,440	1,106,140
Bubble_sort	2,957,756,892	2,957,461,075
All	478,461,360	464,784,469

Rcall Optimization(김경호)

[Progress]

첫 번째 Sprint에서는 Recursive Function Call 의 성능을 향상시키기 위한 LLVM RCall Optimization Function Pass를 구현했다. 이 Pass는 각 Function의 Instruction 을 Iterate 하면서 Call Function을 Identify 하고, Call Function Identification이 된 함수와 Current Function을 비교하여 Recursive Call이 되었는지 확인한다. Recursive Call을 감지하면 Pass가 , Function Call을 Tail Call로 표시하고 코드 선능을 최적화한다. 이 Optimization은 성능을 향상시키고 Stack Space 사용량을 줄일 수 있다. Optimization 의 정확성을 확인하기 위해 3개의 테스트 케이스를 만들어 Implementation을 확인 해보았다. 3개의 Test Case 는 Factorial, Fibonacci과 Divide and Conquer 방식을 적용한 findMax Function이다

[Result]

```
define i64 @factorial(i64 %n) {
entry:
  %cmp = icmp eq i64 %n, 0
  %cmp1 = icmp eq i64 %n, 1
  %or = or i1 %cmp, %cmp1
  br i1 %or, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  ret i64 1

if.else:                                          ; preds = %entry
  %sub = sub i64 %n, 1
  %call = call i64 @factorial(i64 %sub)
  %mul = mul i64 %n, %call
  ret i64 %mul
}
```

```
define i64 @factorial(i64 %n) {
entry:
  %cmp = icmp eq i64 %n, 0
  %cmp1 = icmp eq i64 %n, 1
  %or = or i1 %cmp, %cmp1
  br i1 %or, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  ret i64 1

if.else:                                          ; preds = %entry
  %sub = sub i64 %n, 1
  %call = tail call i64 @factorial(i64 %sub)
  %mul = mul i64 %n, %call
  ret i64 %mul
}
```

```
define i64 @fibonacci(i64 %n) {
entry:
  %cmp = icmp sle i64 %n, 1
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  ret i64 %n

if.else:                                          ; preds = %entry
  %sub = sub i64 %n, 1
  %call = call i64 @fibonacci(i64 %sub)
  %sub1 = sub i64 %n, 2
  %call1 = call i64 @fibonacci(i64 %sub1)
  %add = add i64 %call, %call1
  ret i64 %add
}
```

```
define i64 @fibonacci(i64 %n) {
entry:
  %cmp = icmp sle i64 %n, 1
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  ret i64 %n

if.else:                                          ; preds = %entry
  %sub = sub i64 %n, 1
  %call = tail call i64 @fibonacci(i64 %sub)
  %sub1 = sub i64 %n, 2
  %call1 = tail call i64 @fibonacci(i64 %sub1)
  %add = add i64 %call, %call1
  ret i64 %add
}
```

```
define i32 @find_max(i32* %arr, i32 %size) {
entry:
  %cmp = icmp eq i32 %size, 1
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %arr0 = getelementptr inbounds i32, i32* %arr, i32 0
  %arrload = load i32, i32* %arr0
  ret i32 %arrload

if.else:                                          ; preds = %entry
  %div = sdiv i32 %size, 2
  %arr_right = getelementptr inbounds i32, i32* %arr, i32 %div
  %arr_left = getelementptr inbounds i32, i32* %arr, i32 0
  %call = call i32 @find_max(i32* %arr_left, i32 %div)
  %sub = sub i32 %size, %div
  %call1 = call i32 @find_max(i32* %arr_right, i32 %sub)
  %max = call i32 @llvm.max.signed.i32(i32 %call, i32 %call1)
  ret i32 %max
}
```

```
define i32 @find_max(i32* %arr, i32 %size) {
entry:
  %cmp = icmp eq i32 %size, 1
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %arr0 = getelementptr inbounds i32, i32* %arr, i32 0
  %arrload = load i32, i32* %arr0
  ret i32 %arrload

if.else:                                          ; preds = %entry
  %div = sdiv i32 %size, 2
  %arr_right = getelementptr inbounds i32, i32* %arr, i32 %div
  %arr_left = getelementptr inbounds i32, i32* %arr, i32 0
  %call = tail call i32 @find_max(i32* %arr_left, i32 %div)
  %sub = sub i32 %size, %div
  %call1 = tail call i32 @find_max(i32* %arr_right, i32 %sub)
  %max = call i32 @llvm.max.signed.i32(i32 %call, i32 %call1)
  ret i32 %max
}
```

.

Global Variable Optimization(오준석)

[Progress]

이번 Sprint1에서 나는 계획대로 Global Variable Optimization을 구현했다. 구현 방식은 Global Variable을 함수에서 사용한다면, 함수의 entry block에서 local var(using alloca)에 load하고, local var을 대신 사용하다가, 함수 종료시 다시 global var에 store하는 방식을 택했다. 이를 통해 global(heap) access를 줄이고자 하였다.

이때, 중요한 점으로, 함수 내부에서 다른 함수를 호출하여 다른 함수로 이동하는 경우에는 global var에 다시 값을 저장해야 한다. 또한 해당 함수에서 return시 다시 local var에 global var의 값을 저장해야 한다.

[Result]

- 아쉽게도 전반적으로 다 cost가 증가했다.

```
------- floyd < input1 -------
Final Cost : 71821
------- floyd < input2 -------
Final Cost : 3112656
------- floyd < input3 -------
Final Cost : 28211304
------- floyd < input4 -------
Final Cost : 2882742
------- floyd < input5 -------
Final Cost : 28355550
```

```
------- floyd < input1
Final Cost : 55802
------- floyd < input2
Final Cost : 1647381
------- floyd < input3
Final Cost : 15532125
------- floyd < input4
Final Cost : 2380163
------- floyd < input5
Final Cost : 23513387
```

Global var을 사용하는 benchmark인 floyd의 경우, 우측의 non-optimized 결과보다 좌측의 global → local opt의 cost가 증가했음을 알 수 있다. 그 원인은, 'global var저장용 local var을 관리하는 데에 드는 overhead(global 변수 1개당 최소 110(stack store+load, global var load/stack) 증가, 함수 호출 횟수 만큼 추가로 증가)'가 global->local에 의한 cost감소(load/store시 마다 20씩 감소)보다 크기 때문이다. cost가 증가하는 케이스가 있을 것이라 예상은 했으나, 이 정도로 심할 줄은 몰랐다.

```
------- anagram < input1
Final Cost : 1448808
------- anagram < input2
Final Cost : 1365799
```

```
------- anagram < input1
Final Cost : 1448808
------- anagram < input2
Final Cost : 1365799
```

또한 당연하게도 global var이 없는 benchmark의 경우 Cost가 변하지 않는다.

추가적인 효율성 검사를 위해 내가 만든 testcase에 적용을 해보았다. 해당 testcase는 Github에 올라온 testcase에서 loop도는 횟수를 100 → 10000으로 올렸을 뿐이다.

```c
int global_N = 10000;
int global_SUM = 0;
int main() {
for(int i = 0; i < global_N; i++) {
    global_SUM += i;
}
  return 0;
}
```

```
    Running `target/release/main ./gv_plain.s`
Final Cost : 3126783
```

```
    Running `target/release/main ./gv_opt.s`
Final Cost : 2527054
```

좌측 C코드의 llvm IR을 각각 아무것도 적용안한 compiler과, GVoptPass를 적용한 compiler로 돌려서 나온 결과는 우측과 같다. Pass가 적용된 gv_opt.s의 실행 cost가 더 작음을 알 수 있다. (Github의 testcase 2 변형)

```c
int main() {
for(int i = 0; i < global_N; i++)
    global_SUM += add(i, i);
}
  return 0;
```

```
    Running `target/release/main ./gv_func_loop.s`
Final Cost : 3486783
```

```
    Running `target/release/main ./gv_func_loop_opt.s`
Final Cost : 6087054
```

반면, 위 case는 loop내부에서 다른 function을 부르는 경우이다. 해당 경우는 loop내부에서 local → global → local ->global...의 load/store이 계속 일어나기에 overhead가 더 커져, pass가 적용된 버전인 gv_func_loop_opt.s에서 cost가 더 늘어났음이 확인 가능하다.

Arithmetic Optimization(나동현)

[Progression & explanation]

제시받은 asmspec에서 machine의 Scalar Arithmetic 중 Interger Shift(shl, lshr, ashr)와 Bitwise Operation(and, or, xor), Integer Add/Sub(add, sub)는 cost가 각각 4, 4, 5로 cost가 1인 다른 Scalar arithmetic에 비해 꽤 높은 cost를 가지고 있다. 따라서 해당 연산들에 대해서 아래와 같이 optimization을 실행하고자 하였다.

#	Before	After optimization
1	%b = add i64 %a, %a	%b = mul i64 %a, 2
2	%res1 = shl i64 %a, 1 %shr = lshr i64 %a, 2 %shr = ashr i64 %a, 3	%res1 = mul i64 %a, 2 %shr = udiv i64 %a, 4 %shr = sdiv i64 %a, 8
3	%res1 = and i1 %a, %b %res2 = or i1 %a, %b %res3 = xor i1 %a, %b	%res1 = mul i1 %a, %b %res2 = select i1 %a, i1 %b, i1 0 %res3 = icmp neq i1 %a, %b

4	%inc = add i64 %a, 4 %inc = sub i64 %a, -4 %dec = add i64 %a, -4 %dec = sub i64 %a, 4	(%inc = add i64 %a, 4의 경우) %inc.tmp1 = call i64 @incr_i64(%inc) %inc.tmp2 = call i64 @incr_i64(%inc.tmp1) %inc.tmp3 = call i64 @incr_i64(%inc.tmp2) %inc = call i64 @incr_i64(%inc.tmp3)

이 중 #2, #3, #4에 대한 optimization을 구현하고, instruction에 사용된 변수들(=레지스터들)의 이름이 온전히 원래 변수들의 이름을 유지하면서 instruction만 대응하는 적은 cost의 instruction으로 변화시키는지 FileCheck를 통해 unit test에도 성공하였다.

#1의 경우 우선적으로 구현한 #4를 구현하는 데 있어 incr, decr를 declare하고 call하는 방법을 알아내는 데 시간을 너무 많이 허비하였기에 미처 구현하지 못하였다. #1은 차후 PR로 추가 구현할 예정이다. 또한 구현 시 되도록이면 #2의 shl의 overflow를 감안한 버전도 추가하고자 한다.

[Result]

프로그램명	Average cost(before/after)		프로그램명	Average cost(before/after)	
Anagram	281339432	276882507	Gcd	1413	1413
Binary_tree	274309202	274251356	Jenkins_hash	1847059	1789277
Bitcount1	2107	2065	Matmul1	1831034522	1812091062
Bitcount2	1764	1736	Matmul2	2346869552	2327926092
Bitcount3	1597	1512	Matmul3	962970008	905016979
Bitcount4	1121440	1120407	Matmul4	3273690710	3231459373
Bitcount5	67424	67417	Merge_sort	39633106	39256087
Bubble_sort	2957756893	2885509865	Prime	280302146	280109206
Collatz	4859	4795	Rmq1d_naive	1023292787	1023279835
Floyd	8625772	8608701	Rmq1d_sparsetable	99017495	98482309
Friend	111408055	111242871	Rmq2d_naive	20560080	20546772
Game	31551829	31538008	Rmq2d_sparsetable	436840908	434026603