LOW POWER OPTIMIZATION FOR EMBEDDED COMPILERS

Project Supervisor

Dr. Bibhas Ghoshal

Submitted by:

D. Rajeswar Rao	IIT2014055
Saurabh Tanwar	IIT2014140
Sachin Agarwal	IIT2014501
Nishit Gupta	IIT2014502

MOTIVATION

- The present focus of Compiler Optimization techniques is reducing the "Execution Time" of the program.
- With the advancement in technology, portable devices, embedded systems etc. are being widely developed which are handy and easy to use.
- These portable devices are battery powered.
- Along with performance, the power consumption also needs to be looked at i.e. we need devices which should consume less power along with generating output faster.
- Therefore, the idea of **Low Power Optimization.**

- Energy has become an important design consideration, together with performance in computer systems.
- While hardware optimizations has been the focus of several studies and are fairly mature, software approaches to optimizing power are relatively new.
- Progress in understanding the impact of traditional compiler optimizations and developing new power-aware compiler optimizations are important to overall system energy optimization.
- Here we focus on two questions?
  - Is the most efficient code from Execution Time perspective same as that for the Energy viewpoint?
  - If not, then what are the Power-Aware Optimization Techniques?

OBJECTIVE

- To conduct a survey and find the result of Current Optimization Techniques on Performance and Power.
- Study various Power-aware Optimization Techniques.
- With the Knowledge of these Techniques, develop a patch that could reduce Power and compare it with the present Optimization Techniques.

TARGETS ACHIEVED

- Survey successfully completed.
  - Conclusion :

    "Power Optimization is not a by product of Performance Optimization"

    i.e. with the use of current Optimization Techniques which aim primarily on Performance, Power does not get optimized along with it.
- Studied Different Power-aware Optimization Techniques.
  - Reducing Instruction Count.
  - Tuning Memory Optimizations.
  - Reducing Switching.
  - Instruction Scheduling etc.
- Developed a Pass using LLVM.
  - Conclusion :

    Power reduced.

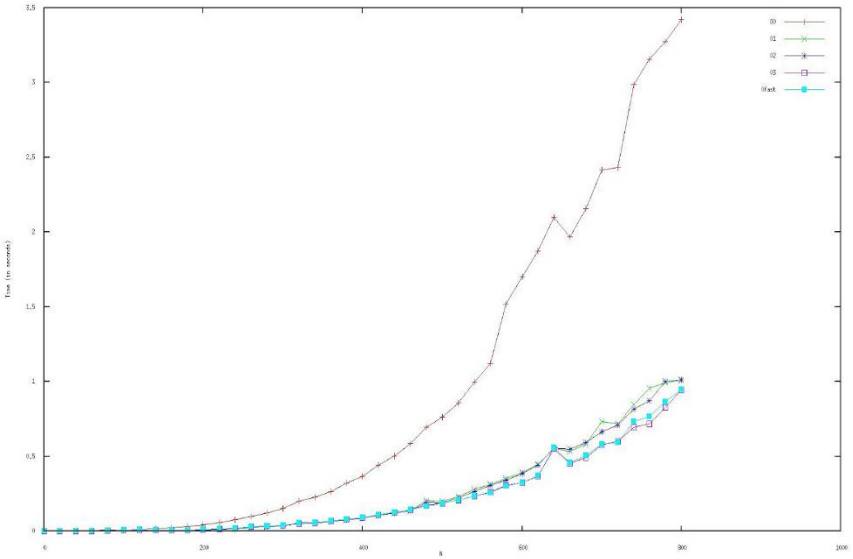
METHODOLOGICAL STEPS

Survey (First Phase)

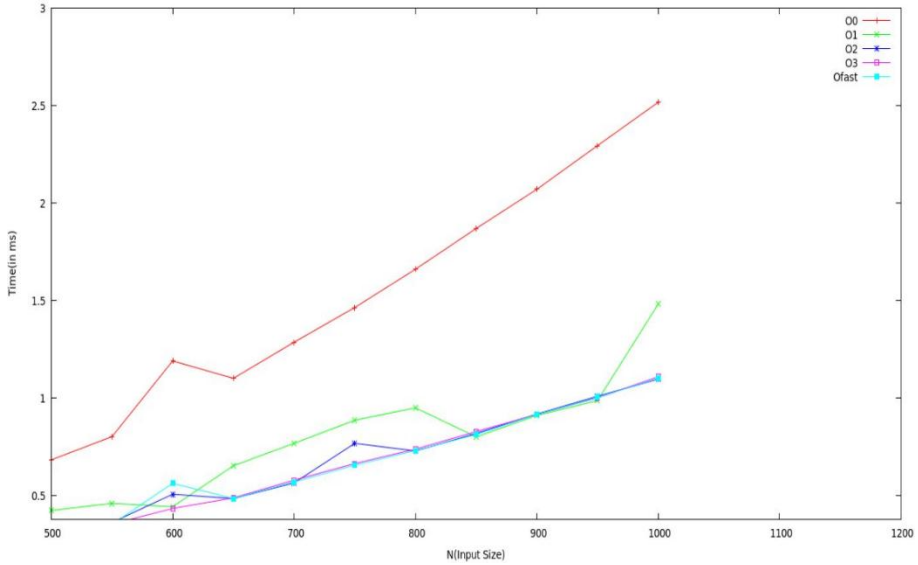
1. Used a Matrix Multiplication code [Running Time – $O(n^3)$] and Bubble Sort code [Running Time – $O(n^2)$]
2. Compiled it with different Optimization Levels – O0,O1,O2,O3 and Ofast using the GCC Compiler.

   Using the command: gcc –Ox matrix_mult.c

   gcc –Ox bubble_sort.c
3. Plotted a curve for Execution Time for different values of n and also various Optimization Levels using GNUPlot.

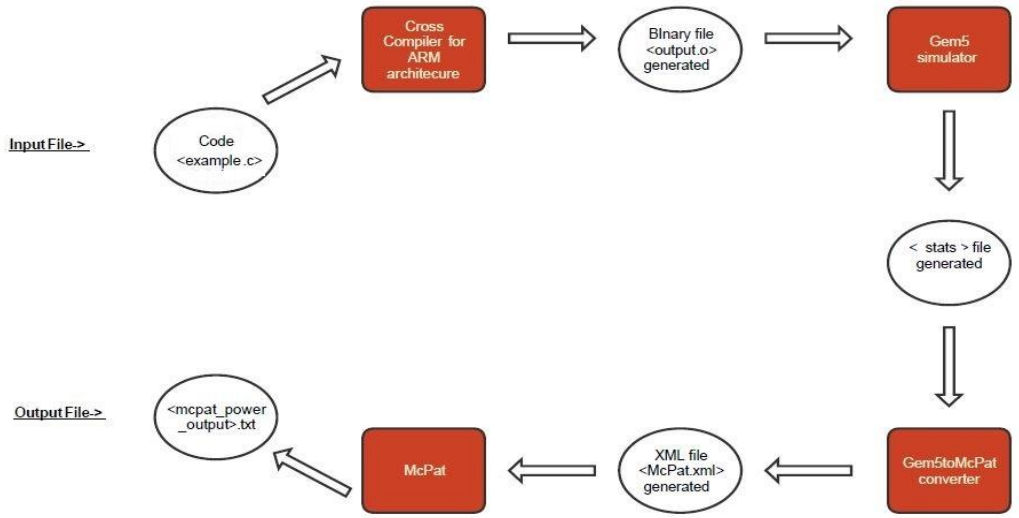


Matrix Multiplication



Bubble Sort

4. Calculated Power Consumed using the following procedure:



BLOCK DIAGRAM(POWER ESTIMATION)

n	O0	O1	O2	O3	Ofast
200	522.784mW	570.949mW	571.1mW	571.1mW	571.1mW
400	511.355mW	517.956mW	517.966mW	517.966mW	517.966mW

Matrix Multiplication Power Consumption

n	O0	O1	O2	O3	Ofast
1000	513.232mW	513.4996mW	513.4996mW	513.4997mW	515.264mW

Bubble Sort Power Consumption

Power Reduction (Second Phase)

1. Various Power Optimization Techniques :
   a) Reducing Instruction Count : Less no. of instructions directly imply less power.
   b) Tuning Memory Optimizations : Replacing *Memory References by Register References*. It will save memory loads & stores. Also we can use Scalar replacement to reduce fetching of data from memory by making good use of Processor Registers.
   c) Constant Folding and Constant Propagation *:*

- Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime.
- Constant propagation is the process of substituting the values of known constants in expressions at compile time.

```
int a = 30;
int b = 9 - (a / 5);
int c;

c = b * 4;
if (c > 10) {
   c = c - 10;
}
return c * (60 / a);
```

Before applying these optimizations

```
int a = 30;
int b = 3;
int c;

c = b * 4;
if (c > 10) {
   c = c - 10;
}
return c * 2;
```

Applying constant folding and propagation once

   d) Strength Reduction *:* Strength reduction is a compiler optimization where expensive operations are replaced with equivalent but less expensive operations.

```
c = 7;
for (i = 0; i < N; i++)
{
    y[i] = c * i;
}
```

```
c = 7;
k = 0;
for (i = 0; i < N; i++)
{
    y[i] = k;
    k = k + c;
}
```

Strength reduction for reducing multiplication with weaker additions.

2. Making of Power Optimization Pass (Patch)

Requires the Use of LLVM and CLANG

- LLVM compiler infrastructure project (formerly Low Level Virtual Machine) is a "collection of modular and reusable compiler and tool chain technologies" used to develop compiler front ends and back ends .
- Allows modifications to be made to the intermediate code which is basically the application of Pass which user makes.
- LLVM uses CLANG as Front-end, so we can use this front-end to avail the facilities of the LLVM Compiler Collection.
- We will be writing a Pass for this LLVM compiler infrastructure .

What is an LLVM Pass?

- Passes perform the transformations and optimizations that make up the compiler.
- They build the analysis results that are used by these transformations, and they are, above all, a structuring technique for compiler code.
- All LLVM passes are subclasses of the Pass class, which implement functionality by overriding virtual methods inherited from Pass.
- LLVM Pass Framework is that it schedules passes to run in an efficient way based on the constraints that your pass meets.

What does our Pass do?

- Our Pass minimises arithmetic instructions making use of the following identities:
  - x + 0 = 0 + x = x
  - x - 0 = x
  - 0 - x = -x
  - x/x = 1
  - x * 1 = 1 * x = x
  - x/1= x
  - x * 0 = 0 * x = 0

  i.e. We will check both the operands and the operator and if they satisfy the above mentioned conditions, then we will directly return the result without actually computing + , - , * or / thereby reducing the instructions to be executed and also a CPU-task.
- It further uses the techniques of Constant Folding and Strength Reduction as discussed previously.

EXPLANATION OF OUR PASS

- All the algebraic identities mentioned previously reduce the no. of instructions in the intermediate code.
- Look at the following C code :

```c
#include <stdio.h>
int compute(int a, int b)
{
    int res = (a/a);
    unsigned int t = (b/b);
    res *= (b/b);
    res += (b-b);
    res /= res;
    res -= res;
    return res;
}
int main()
{
    int ans = compute (2,3);
    return 0;
}
```

- We will now see the effect of our pass on the above C code.
- The pass will first apply Algebraic Reduction, Constant Folding followed by Constant Propagation.

```
; Function Attrs: nounwind uwtable
define i32 @compute(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  %res = alloca i32, align 4
  %t = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %a.addr, align 4
  %div = sdiv i32 %0, %1
  store i32 %div, i32* %res, align 4
  %2 = load i32, i32* %b.addr, align 4
  %3 = load i32, i32* %b.addr, align 4
  %div1 = sdiv i32 %2, %3
  store i32 %div1, i32* %t, align 4
  %4 = load i32, i32* %b.addr, align 4
  %5 = load i32, i32* %b.addr, align 4
  %div2 = sdiv i32 %4, %5
  %6 = load i32, i32* %res, align 4
  %mul = mul nsw i32 %6, %div2
  store i32 %mul, i32* %res, align 4
  %7 = load i32, i32* %b.addr, align 4
  %8 = load i32, i32* %b.addr, align 4
  %sub = sub nsw i32 %7, %8
  %9 = load i32, i32* %res, align 4
  %add = add nsw i32 %9, %sub
  store i32 %add, i32* %res, align 4
  %10 = load i32, i32* %res, align 4
  %11 = load i32, i32* %res, align 4
  %div3 = sdiv i32 %11, %10
  store i32 %div3, i32* %res, align 4
  %12 = load i32, i32* %res, align 4
  %13 = load i32, i32* %res, align 4
  %sub4 = sub nsw i32 %13, %12
  store i32 %sub4, i32* %res, align 4
  %14 = load i32, i32* %res, align 4
  ret i32 %14
}

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %ans = alloca i32, align 4
  store i32 0, i32* %retval
  %call = call i32 @compute(i32 2, i32 3)
  store i32 %call, i32* %ans, align 4
  ret i32 0
}
```

Intermediate Code before applying Pass

```
 5 ; Function Attrs: nounwind uwtable
 6 define i32 @compute(i32 %a, i32 %b) #0 {
 7 entry:
 8   ret i32 0
 9 }
10
11 ; Function Attrs: nounwind uwtable
12 define i32 @main() #0 {
13 entry:
14   %call = call i32 @compute(i32 2, i32 3)
15   ret i32 0
16 }
```

Intermediate Code after applying Pass

CODE SNIPPET

```
switch(operation){
      case 0:{ //multiplication
       if(identity == 0){
                    if(cint = dyn_cast<ConstantType>(operand2)){ //operand 2 is 0 and it's multiplication
                      if(id_compare<ConstantType, APType>(*cint, generalZeroOne))
                        return ConstantType::get(cint->getContext(), generalZeroOne);
                    }
                    else if(cint = dyn_cast<ConstantType>(operand1)){ //operand 1 is 0 and it's multiplication
                      if(id_compare<ConstantType, APType>(*cint, generalZeroOne))
                        return ConstantType::get(cint->getContext(), generalZeroOne);
                    }
                   }else if(identity == 1){
                    if(cint = dyn_cast<ConstantType>(operand2)){ //operand 2 is 1 and it's multiplication
                      if(id_compare<ConstantType, APType>(*cint, generalZeroOne))
                        return cint;
                    }
                    else if(cint = dyn_cast<ConstantType>(operand1)){ //operand 1 is 0 and it's multiplication
                      if(id_compare<ConstantType, APType>(*cint, generalZeroOne))
                        return cint;
                    }

                   }
                   break;
                  }
      case 1:{ //division

       if(identity == 0){
                    if(cint = dyn_cast<ConstantType>(operand1)){ //operand 1 is 0 and it's division 0 / X = 0
                      if(id_compare<ConstantType, APType>(*cint, generalZeroOne))
                        return ConstantType::get(cint->getContext(), generalZeroOne);
                    }
                   }else if(identity == 1){
                    if(cint = dyn_cast<ConstantType>(operand2)){ //operand 2 is 1 and it's division X / 1 = X
                      if(id_compare<ConstantType, APType>(*cint, generalZeroOne))
                          return cint; //if operand 2 is 1 return X
                        }
                      }
                      break;
                    }
                   }

                   return NULL;
                  }
```

```
ConstantInt* fold_constants(unsigned operation, ConstantInt *op1, ConstantInt *op2){
  switch(operation){
    case Instruction::Add:
    return ConstantInt::get(op1->getContext(), op1->getValue() + op2->getValue());
    case Instruction::Sub:
    return ConstantInt::get(op1->getContext(), op1->getValue() + op2->getValue());
    case Instruction::Mul:
    return ConstantInt::get(op1->getContext(), op1->getValue() * op2->getValue());
    case Instruction::UDiv:
    return ConstantInt::get(op1->getContext(), op1->getValue().udiv(op2->getValue()));
    case Instruction::SDiv:
    return ConstantInt::get(op1->getContext(), op1->getValue().sdiv(op2->getValue()));

  }
  return NULL;
  errs() << "An opportunity to fold constants here.." ;

}
```

RESULTS

Execution Time

```
nishit@nishit:~/MAT$ clang matrixmult.c
nishit@nishit:~/MAT$ time ./a.out

real    0m0.498s
user    0m0.496s
sys     0m0.000s
nishit@nishit:~/MAT$ clang -O0 -emit-llvm -c matrixmult.c
nishit@nishit:~/MAT$ opt -mem2reg matrixmult.bc -o mem2reg.bc
nishit@nishit:~/MAT$ opt -stats -load ../my_pass/build/arithmetic_pass/libSkeletonPass.so -my-local-op
ts mem2reg.bc -o withPass.bc
nishit@nishit:~/MAT$ time lli withPass.bc

real    0m0.275s
user    0m0.264s
sys     0m0.008s
nishit@nishit:~/MAT$
```

%Reduction for Execution Time = 44.7%

Power Consumption

WITHOUT PASS	WITH PASS
PROCESSOR	% Reduction = 22.13 %
Runtime Dynamic : 0.658171 W	Runtime Dynamic : 0.512492 W
CORE	% Reduction = 22.23 %
Runtime Dynamic : 0.658171 W	Runtime Dynamic : 0.511807 W

CONCLUSION & FUTURE SCOPE

- FIRST PHASE

On the basis of execution time survey and power survey conducted , the conclusion was that the ***present optimization techniques merely focus on execution time reduction and not on power reduction.***

Hence there is need to explore optimization techniques whose primary goal is to reduce power consumption .

- SECOND PHASE

After going through various power optimization techniques, focus was to reduce instruction count mainly.

We implemented a pass which targeted constant folding and strength reduction optimizations and reducing algebraic operations thereby reducing instruction count and succeeded in ***reducing power as opposed to the increase in power consumption by previously existent optimization techniques.***

This project can further be modified to include other power optimization techniques for other benchmarks like reducing number of loops, loop unrolling etc. which focus on power reduction.

If implemented in a skillful manner and on a broad scale, power reduction can pave a new way for the future by increased use of embedded system and battery-operated devices.

THANK YOU!