Advances in Compilers

Prof. (Dr.) K.R. Chowdhary, Director COE
Email: kr.chowdhary@jietjodhpur.ac.in
webpage: http://www.krchowdhary.com

JIET College of Engineering

August 4, 2017

Outlines:

1. Compiler basics: revisiting
2. New vs old compilers
3. Parallel language features
4. Parallel compilers
5. Modern Compilers
6. CETUS
7. Compiling
8. Lexical analyser generator
9. Resources for lex

Compilation steps

Compilers are language processors (input HLL, output LLL).

1. Tokenizing (lexical analysis)
2. Parsing (syntax analysis)
3. Static semantic checking
4. Intermediate code generation
5. optimization
6. final code generation

Compiler v/s Interpretation

- Compiled programs run faster, if they are compiled into the form that is directly executable on the underlying hardware
- Static compilation can devote arbitrary amount of time for program analysis and optimization
- Hence JIT (just in time) compiler
- Interpreted programs are typically smaller
- Interpreted programs tends to be more portable

JIT: Just in time compilers

Traditionally there are two approaches to translation:

1. Compilation: translates one language to another (say, C to assembly or C to machine). It can be further improved after compilation.
2. Interpretation: No more improvement possible, and does immediate execution.

What practice one should do along with principles?

- Focus on implementing core parts of a compiler, with building the infrastructure
- Small language as an implementing target
- Build from scratch a working interpreter for a small functional language
- The practical should help in understanding the language
- Learning compilers provide strong theoretical foundations.

Early v/s new compilers

- Early compilers: Written in low level languages like C or assembly
- Modern compilers are written in C/C++, C#, F#, Java.
- Single individuals usually crafted compilers, but modern compilers are typically large
- In new: there are RISC, CISC machines of past, vector processors, multicore, etc.

- Number of process are running instructions simultaneously
- Instruction cycle?
- Say, fetch time (t_f) = decode time (t_d) = run-time $(t_r) = t$ μ-secs, then parallel processing possible.
- Then, a program of 100 machine instructions will take $300t$ μ sec. on that machine.
- If instruction I_i is in execution, I_{i+1} is in decode cycle, and I_{i+2} is in fetch cycle. Then, it will take only $100t$ μ sec.!!

Multi-core CPUs

Consider the following code:

1. $b = a + c$;
2. $d = b + e$;
3. $f = g + d$;

   Running 1. and 2. on two cpus at the same time?(Y/N)

   Running 1. and 3. on two cpus at the same time?(Y/N)

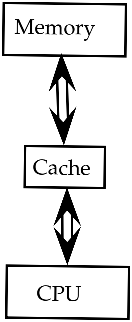
   Running bubble sort on two cpus at the same time?(Y/N)

   Running quick sort on two cpus at the same time?(Y/N)

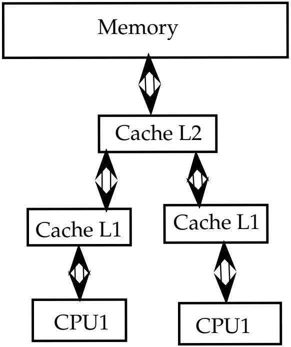
Modern compilers

- Compiler algorithms for *parsing*, *type checking*, *data-flow analysis*, *loop transformations* are based on - data dependent analysis in graphs, register allocation based on graph coloring, are new techniques.
- Coding optimization: avoids redundant computations, register allocation, enhancing locality, instruction level parallelism.
- Optimized compilers produce code having some times peak performance of target machine.
- identifying some of the frequent bugs (e.g., improper memory allocation, locations, race conditions, buffer over-run). Thus providing better security.

Single core:



Multicore:



Parallel Compilers

In the era of multicore processors:

- From now on, clock frequencies will rise slowly, if at all,
- but the number of cores on processor is likely to double every couple of years
- by 2020 microprocessors are likely to have number of cores hundreds of even thousands, with heterogeneous functionalities
- Exploiting large -scale parallel will be essential for improving an applications performance
- This is to do without undue programmer effort

Compiler challenges and agenda for future

The full potential can be achieved only if parallel compilers exist.
Agenda for the Compiler Community:

- Open compiler architectures,
- Open benchmarks for performance evaluation,
- Make parallel program main-stream
- Write compilers capable of self improvement
- Develop algorithms for optimization of parallel code
- Develop software as reliable as airplane
- Enable system software that is secure at all levels
- Expand compiler courses with new problem domains (such as security)

Compiler challenges and agenda for future ...

- Make parallel programming as mainstream
- As on today the parallel programming exists only in databases, and server-side applications
- How to parallel this?: $\sum_{i=0}^{i=999} a_i$
- It is normally done as: fetch instruction, fetch data, decode, execute (cycle repeated 1000 times)
- For parallelism fetch opcode once only, decode once only, fetch data 1000 times, run parallel with fetch !!

Compiler challenges and agenda for future ...

- Enable system software that is secure at all levels: sophisticated program analysis, prevention of software vulnerabilities, like buffer over-flow and dangling pointers arising from coding defects.
- Automatic verification of complete software stack: "Instead of debugging a program, prove that it meets its specifications, and this proof should be checked by a computer program" - John McCarthy.

CETUS: A source-to-source compiler infrastructure for multicores

- After the name of a constellation
- Automatic parallelization
- Input C program, output for 64 bit machines
- CETUS symbolic expression tools:

  $1+2*a+4-a \Rightarrow 5+a$ (folding)

  $a*(b+c) \Rightarrow a*b+a*c$ (distribution)

  $(a*2)/(8*c) \Rightarrow a/(4*c)$ (division)

  $(1-a) < (b+2) \Rightarrow (1+a+b) > 0$ (normalization)

  a && 0 && $b \Rightarrow 0$ (short-circuit evaluation)

CETUS: A source-to-source compiler infrastructure for multicores

Cetus does automatic parallelization:

- by using data dependence analysis
- by array and scalar privatization
- by reduction variable recognition
- by induction variable substitution

gnu C compiler commands

```
#include<stdio.h>

int main(void)
{
   printf("\n Hello World\n");
   return 0;
}

$ gcc main.c

$ gcc main.c -o main

# The output of preprocessing stage
$ gcc -E main.c > main.i
# Assembly level output
$gcc -S main.c > main.s
```

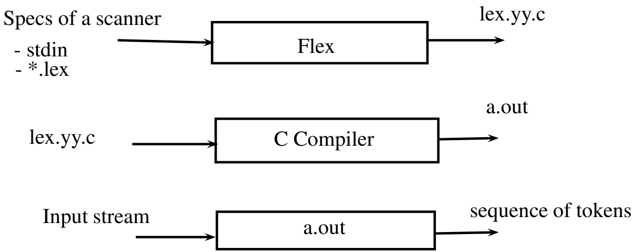
gnu C compiler commands

```
#compiled code (without any linking)
$gcc -C main.c
#all the intermediate files using -save-temps function
$ gcc -save-temps main.c
$ ls
a.out  main.c  main.i  main.o  main.s
# Print all the executed commands using -V option
$ gcc -Wall -v main.c -o main
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/i686-linux-gnu/4.6/lto-wra
Target: i686-linux-gnu
Configured with: ../src/con ....
Thread model: posix
gcc version 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)
```

```
$ gedit test.lex
/* just like Unix wc */
%{
int chars = 0;
int words = 0;
int lines = 0;
%}
%%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n        { chars++; lines++; }
.          { chars++; }
%%
int main(){
yylex();
printf("%d %d %d\n", lines, words, chars);
return 0;
}
```

lexical analyser generator...

```
$ flex test.lex
$ ls
$ gcc lex.yy.c -lfl ; link to flex library
$ ./a.out
```



- FLEX (Fast LEXical analyzer generator) is a tool for generating scanners.
- First, FLEX reads a specification of a scanner either from an input file *.lex, or from standard input, and it generates as output a C source file lex.yy.c.
- Then, lex.yy.c is compiled and linked with the "-lfl" library

lexical analyser generator...

- *.lex is in the form of pairs of regular expressions and C code.
- lex.yy.c defines a routine yylex() that uses the specification to recognize tokens; a.out is actually the scanner!

```
Command Sequence:
   flex sample*.lex
   gcc lex.yy.c -lfl
   ./a.out
  Input file Format:
       definitions
       %%
       rules
       %%
       user code
The definitions section: "name definition"
The rules section: "pattern action"
The user code section: "yylex() routine"
```

lexical analyser generator...

links:

- http://epaperpress.com/lexandyacc/
- http://dinosaur.compilertools.net/
- http://dinosaur.compilertools.net/lex/
- http://en.wikipedia.org/wiki/History_of_compiler_construction
- http://www.drdobbs.com/database/lex-and-yacc/184409830