Simple Inter-Procedural Register Allocation in Compiler for Network Processor

Mostafa Hagog
November 2002.
Part Of the PowerNP C Compiler project.

Design and Implementation by
Igal Golan
Mostafa Hagog
What is the register allocation?
- Find assignment of variables and temporaries to registers.

Why we need register allocation?
- Registers are fast and limited in number.
- Good utilization of registers improves performance.
- In some machines (RISC) ALU operations only on registers.

Register Allocation is divided into three levels (Usually):
- Basic Block local, Basic Block global, and inter-procedural.

Basic Block local
- All registers assumed to be not valid after end of BB.
- Used with the Basic Block global register allocation.
Register Allocation Cont.

- **Basic Block global (intra-procedural)**
  - Register allocation inside procedures.
  - Equivalent to Graph coloring problem - registers are colors, live ranges are nodes; arc when live ranges intersect.
  - Caller or Callee Save/Restore is needed.
  - Spilling is optimized in the procedure level.
  - Library function handling is simple.
  - Good for general applications.

- **Inter-Procedural register allocation:**
  - "Whole Program" scope - Requires call-graph and global program view.
  - Saves and restores of registers only when needed.
  - Profiling to optimize spilling could be useful.
  - Library functions is an issue
  - Good in special cases - Embedded systems is one of these cases.
Why Use Inter-Procedural Register allocation in PowerNP C Compiler?

- Wire speed applications - Every single instruction reduction is important.
- PowerNP procedures are small (hundred of lines).
- A single procedure probably does not use more than half of the registers. (32 general purpose registers).
- The whole program is known at compile time (even library functions); hence register requirements are known.
- Incremental building is not crucial.
- Profiling is possible and maybe efficient.
C Compilation Process in GCC

- Process each procedure separately
  - Pass the procedure through the whole compilation process, then go to the next procedure in the input.

- Parsing pass
  - Build a syntax tree of the procedure in the input.

- RTL generation pass
  - RTL (Register Transfer Language) is used throughout the compilation process.
  - Translate the syntax tree of a procedure into a list of RTL instructions.

- Optimization and Register allocation passes
  - Performed on the RTL representation of the function.
  - Local register allocation (in the level of basic blocks).
  - Global register allocation (across basic block boundaries and within function boundaries).

- Final pass
  - Assembly generation - translate the RTL representation to assembly.
Register Allocation in GCC

- Performed on the RTL representation.
- In RTL generation pass, pseudo registers are used.
- Live ranges according to the SET/USE/CLOBBER in the RTL instructions.
  - Example RTL instruction:
    ```
    (insn 12 10 14 (set (reg/v:HI 307) (and:HI (reg/v:HI 307) (const_int 10))))
    ```

- CALL instructions are special.
  - CALL-USED registers macro - defined per compiler build.
  - CLOBBER/SET/USE in the CALL RTL instruction.
  - Example:
    ```
    (call_insn 21 19 23 ([call (symbol_ref:HI ("fAnd1"))]
        (clobber (reg:HI r8)) (clobber (reg:HI r31)) ] ) (expr_list (use (reg:HI r8))))
    ```

- Local and GLOBAL register allocation:
  - Is to assign pseudo registers to hard registers (real machine registers).
  - Live/Dead analysis using the SET/USE/CLOBBER in the RTL instructions.
Implementing Inter-Procedural Register Allocation in GCC

- **Simplification of the problem:**
  - Find a set of registers per procedure.
  - Colliding procedures must have different sets.
  - If a procedure needs more registers:
    - Use registers of the successors
    - Save/Restore registers around calls if needed.

- **Missing in GCC:**
  - Finding the registers per procedure.
  - Save/Restore callee registers around calls.

- **Solution:**
  - Use External Allocation Tool
    - Input: Register needs for each procedure, Call graph.
    - Output: Set of registers per procedure.
  - CLOBBER in the CALL RTL instruction for each register used in the callee and its successors.
  - Assumptions:
    - No function pointers - All the calls are known at compile time.
    - No recursion - Recursion is possible but not efficient.
Compilation Process with External Allocation Tool

- GCC Based C Compiler
- Register needs, Call-Graph
- External Allocation Tool

C program → GCC Based C Compiler → Register Allocation Per Procedure → External Allocation Tool → Assembly
- **GCC Starts:**
  - Assume all registers available.
  - Generate requirements for each procedure.

- **Allocation routine:**
  - Set of registers per-procedure feedback to GCC.

- **GCC runs Again:**
  - Register Allocation is according to the feedback.
  - May spill registers.
  - Requirements may change - allocation should be repeated.

- **End the Loop:**
  - When the requirements doesn't change.
  - Or reached a maximum number of iterations (to avoid endless loop).

- **Is the compilation-allocation loop required?**
  - No, the register allocation could be applied on the generated assembly.
  - But, code correctness must be insured - re-implement part of the compiler.
Inter-Procedural Allocation Tool Algorithm

- **PowerNP general purpose registers:**
  - 32 16 bit registers (4 of them have a special use).
  - Accessed as singles (r3 for example), or doubles (w2, r3 and r2).
  - Index registers (registers 0 to 7).

- **Input**
  - Procedure information:
    - Amount of registers needed from each register type.
    - The list of calls to other procedures.

- **Output:**
  - A set of registers per procedure.

- **Data Structure:**
  - Graph (call graph): A node represents a procedure, a directed arc represents a call.
  - Node has:
    - Register requirements (double, single, and address).
    - Free registers bitmap.
    - Incoming and outgoing arcs.
    - Allocated registers.
Algorithm

- Traverse the nodes in DFS reverse order (leafs first).
- In each node do:
  - free registers bitmap = Intersection of free registers bitmaps of its successors.
  - Satisfy register requirements: Allocate Index first, then double and single at last.
  - Update the free registers bitmap and the allocation list.

Example:

Free At the beginning: [29,2]

1. R: 1 S, 2 D, I 1
   A: 17, 16, 15, 14, 13, 4

2. R: 1 S, 1 D, I 3
   A: 29, 28, 27, 26, 25, 24, 23
   Free: [26,8],[4,2]

3. R: 3 S, 1 D
   A: 21, 20, 22, 19, 18
   Free Before: [22, 8],[5,2]
   Free After: [17, 8],[5,2]

4. R: 2 S, 0 D, I 2
   A: 24, 23, 7, 6
   Free: [26, 8],[5,2]

5. R: 3 S, 2 D
   A: 29, 28, 27, 26, 25, 24, 23
   Free: [22, 8],[7,2]

6. R: 1 S, 2 D
   A: 29, 28, 27, 26, 25, 24, 23
   Free: [22, 8],[7,2]
This method could be used in general purpose applications.

Only on the part of the Call-Graph which is:
- Critical for performance.
- Doesn't contain cycles.
- All the calls are known at compile time.
Results show efficient code; inter-procedural register allocation is one reason for that.

Profiling could be used to minimize spilling overhead.

Recursion could be allowed theoretically, practically it requires dynamic memory handling - This is not the trend in embedded systems.
End