Bits & Pieces

Advanced Compiler Techniques

2005

Erik Stenman

Virtutech

Exceptions

- In languages that support exceptions there are usually two ways to generate exceptions:
  - Explicit, user controlled throwing of an exception, e.g. throw MyException;
  - Implicit, exceptions thrown when there is a runtime error e.g. x = 0; 42/x;
- And two ways to handle the exceptions:
  - Explicit catch of the exception, e.g. try E catch MyException { … }
  - "Program crash". (In Erlang only the running process ides.)

Exceptions

- Assumption: Exceptions are exceptional, that is, most of the time when executing a try there will be no exception.
- Optimize for the common case: i.e., make it cheap to set up a try by moving the cost to does times an exception is thrown.
- Entering a try expression does not actually have to cost anything at all.

Exception Handlers

- We can distinguish between two cases:
  - A local handler: When the code that throws the expression is syntactically contained in the try.
  - A non-local return: When the try expression contains a function call.

Local Exception Handlers

- For an expression like:

```
try x/y;
catch ArithmeticException (e) { foo() };
```

- There are basically two ways to handle this:
  1. Rewrite each operation in the try that can fail so that it explicitly takes the address of the handler as an argument. (During compilation a stack is needed to find the right handler if there are nested tries.) (Erlang)
  2. Register the address of the handler at load time, and associate each instruction in the try with the handler address. If an instruction fails, do a lookup at runtime with the current PC. (Java)

Non-local returns

- For an expression like:

```
try bar();
catch ArithmeticException (e) { foo() };
```

- There are basically two ways to handle this:
  1. When entering a try expression put the address of the handler on the stack (need a 'permanent' slot in the activation record). Or even better: use a stack map that maps the return address to the address of the exception handler. (Erlang)
  2. Register the address of the handler at load time, and associate each instruction in the try with the handler address. (Java)

Non-local returns: Method 1

- If an exception occurs look at the stack, if there is an exception handler jump to it.
- If there is no handler in the current frame, unwind the stack until a handler is found. When a handler i found, restore values saved on the stack and jump to the handler.

Non-local returns: Method 2

- If an exception occurs lookup the current address, if there is a handler jump to it.
- If there is no handler unwind the stack and do a lookup for each return address until a handler is found. When a handler i found, restore values saved on the stack and jump to the handler.

Complications in the Intermediate Code

- Normally a function call does not need to end a basic block. If a later phase in the compiler takes care of saving of live registers, a call can be seen as just another instruction.
- Function calls within an exception handler will have two possible continuations, one for a normal return and one for the case when an exception is thrown.

Linker & Loaders

- *Linkers & Loaders* links several compilation units together into a program and loads it into memory.
- The compilation units are stored in *object files*.
- Linkers and loaders perform several related but conceptually separate actions.
  - *Program loading*. (Loaders)
  - *Relocation*. (Linker or loaders)
  - *Symbol resolution*. (Linkers)

Symbol Resolution

- Each object file contains a symbol table, which contains:
  - Global symbols defined by the unit.
  - External symbols. (Referenced symbols)
  - Segment names (e.g. text, data).
  - Local symbols (for debugging).
  - Line numbers (for debugging).
- First all names are stored in a global symbol table, in the second phase during relocation references to symbols are replaced by their addresses.

Relocation

- All object files are usually assumed to start at address zero. The relocation phase rewrites the code using the actual offset of each segment.
- Relocation can be done both at link time and at load time.
- Each object file contains a relocation table, a list of all places in the file that needs to be relocated.
- This list must contain information about addressing modes (e.g. PC relative) and what instruction to relocate (e.g. jmp or sethi).

Loading

- Assuming that linking has been done, the loading is quite simple:
  1. Read the header of the object file to find out the memory need of the program.
  2. Allocate memory.
  3. Read the program into the allocated memory.
  4. Create stack space.
  5. Set up runtime information (such as program arguments).
  6. Jump to the first instruction in the program.

Dynamic Linking and Loading

- Dynamic linking defers much of the linking until runtime. This makes sharing of libraries much easier and makes it possible to load and unload parts of the program at runtime.
- These advantages comes at the price of higher runtime, since linking has to be done each time the program is run.
- There is also a constant problem of versioning. A statically linked program will always have the same version of libraries, but a dynamically linked will usually always have the latest version. This is both a blessing and a problem.

Linking and Loading in VMs

- Modern virtual machines usually uses dynamic linking and loading and can support code updates. (Java supports dynamic loading, classes are loaded as they are needed. Erlang supports dynamic updates, a module can be replaced by a new version in a running system.)
- Most of the loader can usually be written in the high level language itself and can be customized and replaced by the user.

Q & A

Summary

- The goal of this course was for you to learn about
  - compilation techniques used to obtain high performance on modern computer architectures.
  - techniques used to implement high level languages.
- In the course we have talked about:
  - Optimization techniques
    - Intermediate representations (CFGs and SSA), [interprocedural] and intraprocedural data-flow analysis, dependence analysis.
    - Optimization across basic blocks, procedures, [and complete programs].
    - Optimization techniques such as CSE, dead code elimination, constant and copy propagation, constant folding, code motion and loop transformations.
    - Instruction scheduling, register allocation.
  - Implementation of high level languages
    - Implementation of objects, higher order functions, [coroutines], and processes.
    - Memory management and uniprocessor garbage collector techniques.
    - Virtual machines and the efficient implementation of their interpreters.