Procedure Call Transformations

Overview In most programming languages, procedure is an important tool for control abstraction. However, this comes with a price of procedure call overhead. *Procedure call transformations* try to reduce this overhead in one of four ways:

- eliminate the call entirely
- eliminate the exeuction of the called function's body
- eliminate some of the entry/exit overhead
- avoiding some steps in making a procedure call when the behavior of the called procedure is known or can be altered

Calling covnention: Register usage First, let's define a sample calling convention for this memo. First, here are the register usage:

- R0: always zero, writes are ignored
- R1: return value when returning from a procedure call
- R2$\cdots$R7: first 6 arguments to the procedure call
- R8$\cdots$R15: 8 *caller-save registers*, used as temporary registers by callee
- R16$\cdots$R25: 10 *callee-save registers*; these registers must be preserved across a call by callee
- R26$\cdots$R29: reserved for use by the operating system
- R30: stack pointer
- R31: return address during a procedure call
- F0$\cdots$F3: first 4 FP argumments to procedure call
- F4$\cdots$F17: 14 caller-save FP registers
- F18$\cdots$F31: 14 callee-save FP registers

Each called procedure must ensure that the values in registers R16$\cdots$R25 is preserved. So, if the callee needs to use any of these registers, they need to restore back the original values after use.

The **stack** begins at the top of memory and grows downwards. There is no explicit frame pointer but the stack pointer is decremented by the size s of the procedure's frame at entry and left unchanged during the call. The value (R30 $+s$) serves as a *virtual frame pointer* that points to the base of the stack frame, avoiding the use of a second dedicated register.

Calling convention: Procedure execution Executino of a procedure consists of six steps:

1. space is allocated on the stack for the procedure invocation
2. the values of registers that will be modified during procedure execution (and which must be preserved across the call) are saved on the stack
3. ...

Leaf Procedure Optimziation A *leaf procedure* is one that does not call any other procedures. This is a leaf in the call graph. The simplest optimization for leaf procedures is that *they do not need to save and restore the return address* (R31). Also, if the procedure does not have any local variables allocated to memory, the compiler does not need to create a stack frame.

Cross-call register allocation Separate compilation reduces the amount of information available to the compiler about called functions. However, when both callee and caller are available, the compiler can take advantage of the register usage of the callee to optimize the call.

If the callee does not need (or can be restricted not to use) all the temporary variables (R8$\cdots$R15), the caller can leave values in the unused registers throughout execution of the callee. In addition, moving instructions for parameters can be eliminated.

For this optimization, **register allocation** must be performed in a *depth-first postorder traversal* of the call graph, ensuring that each caller will know its callees' register usage.

Parameter promotion When a parameter is *passed by reference*, the address calculation is done by the caller, but the load of the parameter value is done by the callee. This wastes an instruction, since most address calculations can be handled with the `offset(R`n`)` format of load instruction.

More importantly, if the operand is already in a register in the caller, it must be spilled to memory and reloaded by the callee. If the callee modifies the value, it must then be stored. Upon return to the caller, if the compiler can not prove that the callee did not modify the operand, it must be loaded again.