MLRISC
'Omit Frame Pointer' Optimization

Lal George
Bell Laboratories
Lucent Technologies
george@lucent.com

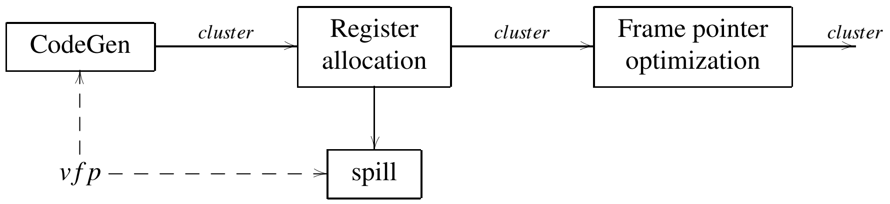
May 14, 2001

Abstract

In many languages, it is convenient to access local variables and spill locations via the use of a frame pointer. The frame pointer may be a physical register, or a virtual register that is later mapped to the stack pointer. This note describes the MLRISC support and requirements to rewrite uses of a virtual frame pointer to uses of the stack pointer – the omit frame pointer optimization.

1 Requirements and Steps

The basic structure of a backend performing the frame pointer optimization is shown in the figure below:



The follwing requirements and steps must be followed.

1. Pre-determine that the omit frame pointer optimization is going to be performed on a specific cluster.
2. If the optimization is being performed then:

(a) Use a dedicated virtual frame pointer for all accesses to local variables and spill locations.
(b) Perform register allocation.
(c) Invoke an optimization (to be described) to replace the virtual frame pointer with uses of the stack pointer.

otherwise:

(a) Use the physical frame pointer for all accesses.
(b) Perform register allocation.

We assume that call statements restore the value of the stack pointer that existed before the call.

2 Dedicated Registers

When performing the omit frame pointer optimization, the virtual frame pointer must not be colored or spilled by the register allocator like other pseudo registers. It is also incorrect to immediately color the virtual frame pointer with the same color as the stack pointer, and later adjust offsets when the frame pointer optimization is performed as this would create incorrect memory aliases.

One of the parameters used to specialize the register allocator is a list of dedicated registers (see `MLRISC/ra/ra.sig`). The registers in this list are excluded from consideration when constructing the interference graph, and are therefore not colored or spilled. The virtual frame pointer must be included in the list of dedicated registers. In contrast, the physical frame pointer need not be marked as a dedicated register, and can be used for other purposes when dead.

3 Callee vs callersave register

Within the same language, the physical frame pointer is treated like a callee save register. The callees save the incoming value, and restore it on exit. This implies that the call statement, does not mention the physical frame pointer in the list of registers it defines and uses.

The virtual frame pointer, in contrast, is a local temporary that does not need to be saved and restored by either the caller or callee.

If a call is being made to a foreign language that uses the physical frame pointer as a caller-save register, and the physical frame pointer is being used by the caller as the frame pointer, then it is necessary for the caller to save and restore the physical frame pointer value around the call. This could be done in a number of ways:

1. The save and restore is done explicitly.
2. A move instruction is inserted before and after the call, that moves the physical frame pointer, to and from a temporary register. This would force the temporary to spill or be allocated to a calleesave register if one is avaiable.

4 New cells

The CELLS interface contains the datatype and functions:

```
datatype cell =
  CELL{id  : cell_id,
       col : cellColor ref,
       ...
      }
val newReg : unit -> cell
val reset : unit -> unit
```

where newReg generates a fresh virtual or pseudo register. The function reset, resets the counters used to generate fresh temporaries and is usually invoked before the compilation of an MLRISC cluster.

The virtual frame pointer cannot be allocated using CELL.newReg() and also be a dedicated register. The dedicated list is used to statically initialize other modules, so when a call to CELL.reset() is executed, the newReg function will start generating fresh temporaries from scratch, and may regenerate a cell that had the same cell id as the dedicated virtual frame pointer.

The virtual frame pointer must be allocated using:

```
val newDedicatedCell: cellkind -> (unit -> cell)
```

which allocates a dedicated virtual or pseudo register. For now, there is an artifical restriction of 256 on the number of such registers in each cellkind than can be generated.

5 Spilling

Spilling is implemented as a call-back from the register allocator, and currently takes the annotation of the block containing the instruction where a register operand must be spilled. Since the omit frame pointer optimization is a cluster property, a cluster annotation must be examined to determine which frame pointer to use. Currently, the cluster annotation is not accessible to the spill call-back, so some other mechanism (such as a global reference) must be used. This situation should change in the near future.

6 Optimization

The optimization module for a specific architecture will provide an implementation matching the OMIT_FRAME_POINTER interface:

```
signature OMIT_FRAME_POINTER = sig
  structure F : FLOWGRAPH
  structure I : INSTRUCTIONS
  val omitframeptr :
       { vfp:I.C.cell,
          cl:F.cluster,
          idelta:Int32.int option }  ->  F.cluster
end
```

- `vfp` is the virtual frame pointer.
- `idelta` is the gap between the frame pointer and stack pointer if one exists, and is the value `NONE` otherwise. The initial condition is:

  ```
  vfp = sp + idelta
  ```

- `cl` is the cluster containing instructions that make references to the virtual frame pointer, and finally,
- the result is the original cluster where the list of instructions in each basic block may have been rewritten.

In general the optimization module does a depth first traversal of the flowgraph, computing the gap between the stack pointer and frame pointer for each program point. It is an error if anything is done to either the stack pointer or frame pointer that makes it impossible to compute the gap. This condition is detected statically.

7 Annotations

The annotation :

```
MLRiscAnnotations.USES_VIRTUAL_FRAME_POINTER
```

can be used to annotate the cluster when the optimization is used.

Each instruction that uses an offset from the virtual frame pointer is also annotated with a comment annotation:

```
MLRiscAnnotations.COMMENT : string Annotations.property
```

where the comment string contains the new offset from the stack pointer. This comment is printed in the assembly code listing.

8 Control Flags

Setting the flag obtained from:

```
MLRISCControl.getFlag "dump-cfg-after-omit-frame-pointer"
```

to `true`, will cause the cluster or flowgraph to be printed out to standard output after the optimization is performed.

9 Example

The table below shows an example in stylized Intel x86 assembly code. `%esp` is the stack pointer, `%eax` is a callee save register, and *vfp* is the virtual frame pointer. On the Intel architecture the convention is that the stack grows from high to low addresses. The first column shows the instructions before the optimization is applied, the middle column is the computed gap between the stack pointer and virtual frame pointer, and the last column is the generated code after the optimization has been applied.

Original	Delta	Omit vfp
f:	NONE	
push %eax		push %eax
	NONE	
vfp ← %esp		
	SOME(0)	
%esp ← %esp- 32		%esp ← %esp- 32
	SOME(32)	
%eax ← mem[*vfp*+ 4]		%eax ← mem[%esp+ 36]
	SOME(32)	
push %eax		push %eax
	SOME(36)	
%eax ← mem[*vfp*+ 8]		%eax ← mem[%esp+ 44]
	SOME (36)	
push %eax		push eax
	SOME(40)	
call printf		call printf
	SOME(40)	
%esp ← %esp+ 8		%esp ← %esp+ 8
	SOME(32)	
mem[*vfp*+ 12] ← %eax		mem[%esp+ 44] ← %eax
	SOME(32)	
pop %eax		pop %eax
	SOME(28)	
ret 32		ret 32
	SOME(-4)	

Note:

- The assignment to *vfp* in the second instruction has been eliminated.
- An error would be signaled if any operation were performed on the stack pointer or virtual frame pointer that precluded the possibility of computing the delta column. This would include instructions that stored these registers to memory, or performed non-linear operations on them.

10 Files

```
MLRISC/cm/MLRISC.cm
MLRISC/omit-frameptr/omit-frame-pointer.sig
MLRISC/instructions/mlriscAnnotations.sig
MLRISC/x86/omit-frameptr/x86omit-frame-pointer.sml
```

Other architectures to be supported soon.

10.1 See Also

```
MLRISC/instructions/cells.sig
MLRISC/ra/ra.sig
```

11 TODO

1. Support other architectures besides the x86. This should be done in conjunction with implementation of the c-call interface.
2. Compare the implementation of the same optimization performed under the gcc flag **-f**`omit-frame-pointer`, and in other compilers. It is quite evident that gcc does not treat this as a separate pass as we do.
3. Support access to cluster annotation from spill call-backs.