Translating ETC to LLVM Assembly

Carl Ritson
C.G.Ritson@kent.ac.uk
School of Computing, University of Kent
1983-1984
INMOS Transputer released.

occam is born.
1989
SGS Thompson acquires INMOS.
occam reaches version 2.1 (occ21).
1993-1994
INMOS subsumed.
Transputer and occam development ends.
1995-1997

occam for All (Welch, BAE, Formal Systems, Marconi, etc)

KRoC is born (octran, tranpc).
2000-2003

occam-\(\pi\) (Welch, Barnes)

tranx86 (Barnes)
2004-2007
More occam-π (Welch, Barnes)
The Transterpreter (Jacobsen, Jadud, Dimmich)
Extended Transputer Code

Targets 3-place stack machine, with workspace.
Small set of RISC-like instructions with CISC secondaries.
LLVM

Machine independent program representation.
Generic analysis and optimisation passes.
Why LLVM?
More control than C.
Code generation for x86, x86-64, ARM, PPC, ...
LLVM Assembly

Typed SSA-form with procedures and calling conventions. Obviates data flow and control flow graphs.
define i32 @cube (i32 %x) {
    %x_0 = mul i32 %x ,%x
    %x_1 = mul i32 %x_0,%x
    ret i32 %x_1
}
PROC foo (VAL INT x, y, CHAN INT out!)

INT z:
SEQ
    z := x + y
    out ! z

:
.L0:

    AJW   -1  -- allocate workspace
    LDL    2  -- load x
    LDL    3  -- load y
    ADD
    STL    0  -- store to z
    LDLP   0  -- load pointer to z
    LDL    4  -- load channel
    LDC    4  -- load size of INT
    OUT
    AJW    1  -- deallocate workspace
    RET
The Transformation
Trace and numerate operand stack.
Extract control flow into procedures.
.L0:

AJW  -1
LDL  2  -- => %reg_0
LDL  3  -- => %reg_1
ADD  -- %reg_1, %reg_0 => %reg_2
STL  0  -- %reg_2 => ()
LDLP 0  -- => %reg_3
LDL  4  -- => %reg_4
LDC  4  -- => %reg_5
OUT  -- %reg_5, %reg_4, %reg_3 => ()
AJW   1
RET
define void @O_foo (i8* %sched, i32* %wptr_0) {
L0:

; AJW -1
%wptr_1 = getelementptr i32* %wptr_0, i32 -1
; LDL 2
%tmp_0 = getelementptr i32* %wptr_1, i32 2
%reg_0 = load i32* %tmp_0

; LDL 3
%tmp_1 = getelementptr i32* %wptr_1, i32 2
%reg_1 = load i32* %tmp_1
; ADD   { (reg_1, reg_0) => (reg_2) }
%tmp_2 = call {i32, i1}
   @llvm.sadd.with.overflow.i32
   (i32 %reg_0, i32 %reg_1)

%reg_2 = extractvalue {i32, i1} %tmp_2, 0
%tmp_3 = extractvalue {i32, i1} %tmp_2, 1
br i1 %tmp_3, label %tmp_4_overflow_error, label %tmp_4_ok

tmp_4_overflow_error:
  %tmp_5 = load i8** @C_0 ; "foo.occ"
  call void @etc_error_overflow
       (i8* %sched, i32* %wptr_1,
        i8* %tmp_5, i32 5)

tmp_4_ok:
  ; STL 0
  store i32 %reg_2, i32* %wptr_1
Control Flow

Branching and blocking, via co-operative scheduling. Functions over workspaces, e.g. $P = \langle f_1, f_2, f_3, \ldots, f_n \rangle$. 
Continuation Passing Style
Collapses stack, obviates dependencies.
Very appropriate for occam-like languages.
Run-time Simplification
Kernel calls become native C calls.
Run-time state more tractable.
spectralnorm
Language Games
agents

CCSP Comparisons
Benefits

Simplifies: run-time, floating point, corner cases.
Off-the-shelf optimisations improve performance.
Difficulties

Unclear semantics of tail-calls (for CPS).
Toolchain issues.
Future Work

Porting, refactoring, bypass ETC.
Add to LLVM support for this style of compilation.
Thanks

Questions?

EPSRC grant EP/D061822