TL0 Version 2.0

Thorsten von Eicken

Klaus Erik Schauser

Threaded Id Compiler Project

Parallel Computing Structures Group

Computer Science Division

University of California, Berkeley

1 Overview

TL0 (standing for Thread Language Zero) is an intermediate language used in the compilation of Id to various non-dataflow machine languages. TL0 started as an implementation of the concepts developed in the Threaded Abstract Machine (*TAM*) as described in [citation]. However, version 2.0 of TL0 departs substantially from TAM, in particular to allow a complete system (i.e. including run-time support functions) to be written in TL0.

The main function of TL0 in the compilation process is to provide a machine independent machine language suitable for rather simple but still efficient expansion to concrete machine languages. Machine independence is understood to mean that all valid TL0 programs can be translated to any of the supported machine languages. Also there are no TL0 constructs which, when used, make a program more suited for a particular class of machines. To guide the language design, TL0 is primarily targeted at distributed memory multiprocessors but care has been taken so that efficient execution on shared memory multiprocessors and uniprocessors can be achieved. Concerning machine independence, one should not forget that the Id to TL0 compilation process may perform high-level optimizations which may favor execution on one of the processor types.

The following sections describe the TL0 language Version 2.0 alternating between a top-down and a bottom-up approach. Regarding terminology, note that the term of *compilation* is used to refer to the translation of TL0 to machine language.

2 Program Structure

A TL0 program is formed by a collection of *codeblocks* and *initialized structure* declarations. The former roughly correspond to Id functions while the latter are used for Id constants (if Klaus and Thorsten can agree on this...). All codeblocks and initialized structures are compiled in isolation and are typically each stored in a file on their own (by convention the file name extension `.tl0` is used).

Each codeblock in in turn consists of a frame declaration section, an inlet section and a thread section. During program execution, for each invocation of a codeblock (i.e. function call) a new frame for local variable storage is allocated. The frame declarations describe the sizes of various parts of the frame. The inlet section describes the interface through which the codeblock receives messages. In general terms, the inlet section provides a message handler (a piece of code) for every message to be received. The thread section contains the code though of as the body of a codeblock.

codeblock ::= `CBLOCK` frame_decl inlet_section thread_section `END_CBLOCK`

3 Storage Model

The storage model of TL0 is composed of three levels: registers, local frame variables ("locals") and heap-allocated structures. Registers are the cheapest storage class to access, however their contents is not preserved beyond the duration of an activation (term defined later). Accessing locals (also called "frame slots") is more expensive that accessing registers in that a local memory access is typically involved. All instructions can take either registers or locals as operands (i.e. TL0 is a memory-to-memory instruction set). To keep TL0 machine independent the number of available registers is not limited, rather TL0 registers are considered to be pseudo registers which can be mapped to hardware registers or to frame slots by the TL0 compiler. By convention lower numbered registers are allocated to hardware registers with higher priority than higher numbered registers.

Heap allocated structures reside in memory just like locals. Unlike locals they can only be acced through special split-phase instructions but remote accesses (i.e. on another processor) are supported transparently. Non split-phase load and store instructions to access local structures are also provided but are mainly intended for low-level run-time system primitives. In addition to any data value, each slot of a structure holds a few tag bits which are used to implement the synchronization primitives of Id I-structures and M-structures.

4 Data Types and Variables

TL0 supports a number of data types shown in table X. All data types except "general" define an interpretation of bit patterns as data values and various operations on values of each data type are provided.

Registers and locals are available in every data type and are referenced through a variable name formed by contracting the variable's 1-letter abbreviation (`g`, `i`, `f`, `c`, `b`, `s` or `p`), the storage class (`reg` for registers and `slot` for local frame slots) and the register/slot number. Note that the registers and slots of each data type are numbered separately (i.e. `ireg0` doesn't refer to the same register as `freg0`).

variable_name ::= [`g`,`i`,`f`,`c`,`b`,`s`,`p`] [`reg`,`slot`][`0-9`][`0-9`]*

Structures only support the data type general and thus values of other types must be cast appropriately (this will hopefully change in the future). With the exception of initialized structures (defined later), structures can only be referenced through pointers which are generated by the memory allocator.

Variables of type general are special in that their purpose is to be used as containers to hold values of any other data type. To that effect the contents of any variable of any type can be moved into a *general* (variable of type general) and re-extracted at a later point. The idea is that generals can be used to hold and move around data of an unknown type as long as the final recipient of the data knows its type (there is no way to query the type of the current content of a general). In other words, the only operations allowed on generals are moves and *type casts* to/from another data type. Type casts allow the content of a general to be interpreted as any other type or they allow a variable of any type to be viewed as a general. The only guaranteed property of type casts is that casting a value of a given type to a general an later casting that general back to the same type yields the original value. Note that a type cast never implies any data conversion however, depending on the instruction, data type and storage class involved, a type cast may require extra data movement.

data_type ::= [`g`,`i`,`f`,`c`,`b`,`s`,`p`]

casted_variable ::= variable_name`.`data_type

Integers are typically 32-bit signed quantities although some machines may support 64-bit integers. Floats (floating-point numbers) are at least in 64-bit IEEE standard format, but some bizarre machines might not really support IEEE conformant arithmetic. The special constants `nan` and `inf` represent "not-a-number" and "infinity" respectively. Characters are 8-bit ASCII and booleans are 1-bit true/false values, but variables of these two types may actually occupy more space.

Synchronization variables are small 8-bit unsigned integers on which only two operations are defined: conversion to/from integers and an atomic decrement-and-test-zero.

Pointers are typically 64 bits wide and contains a global address consisting of a processor number and a local address on that processor. Pointers may point to a variety of memory objects but hold no information (tag) about the type of the object pointed to. As a special case, a pointer variable may hold a small integer in the range 0..255 instead of an address. A special instruction tests whether a given pointer variable holds an address or an integer. A pointer holding the integer 0 is equivalent to a `nil` pointer. The objects pointers may point to are structures, frames and codeblocks. Pointers to structures are generated by the memory allocator or by referencing an initialized structure by name; pointers to frames are also generated by the memory allocator or by a reference of the current frame pointer `fp`; pointers to codeblocks are obtained by referencing a codeblock by name.

data type	1-letter abbrev.	typical size	constants
general	g	64 bits	—
integer	i	32 bits	`-1 103`
float	f	64 bits	`1.0 -1e-12`
character	c	8 bits	`'a /007`
boolean	b	1 bits	`true false`
synch.	s	8 bits	—
pointer	p	64 bits	`nil fp 1p` `cb_foo is_foo`

5 Inlets

A codeblock may receive an arbitrary number of arbitrary messages. However, a message handler must be provided for each message type to be received, these handlers are called *inlets* and are numbered 0..N ($N < 256$). All messages in the system are sent to a specific inlet of a codeblock invocation (i.e. to a frame).

An inlet handler is a thread executed at the time the message is received. In general such an *inlet thread* behaves like a normal thread, however there are a few important differences. Inlet threads are syntactically marked as such by using `ithrN` as thread names. Inlet threads may fork other threads, however forking a normal thread pushes the thread onto the remove continuation vector, and as a result, the thread will only execute when the frame is activated by the global scheduler.

6 Arithmetic Instructions

TL0 provides a rather generous set of arithmetic operations. All unary operations have two operands and all binary operations have three oprands. TL0 use traditional assembly language style prefix notation and the first operand is generally the destination for the result. All oprands must be of the same data type and that type is indicated by a type suffix to the opertion.

unary_instr ::= unary_opcode type_suffix dest source

binary_instr ::= binary_opcode type_suffix dest source source

The following table lists all the unary operations and the allowed operand types. Note that an operand of a given type can be obtained by casting a variable to type genral to that type.

```
Examples:
   ADD.I ireg0 = ireg1 1
   ADD.I islot10 = gslot3.I greg2.I
   MUL.F freg0 = gslot0.F 100.0
   NOT.I ireg0 = ireg1
```

7 Type Conversion Instructions

A number of instructions convert values of one data type to another. These should not be confused with the type casts. The following table lists the conversion instructions:

opcode	conversion	caveats
ITOF	int to float	—
ITOC	int to char	undefined if not in 0..255
ITOS	int to synch	undefined if not in 0..255
ITOP	int to pointer	undefined if not in 0..255
		result is a small int int a pointer
FTOI	float to int	undefined if not in -2^31..2^31-1
CTOI	char to int	—
STOI	synch to int	—
PTOI	pointer to int	undefined if not a small int in a pointer

8 Inlet Instructions

Syntax:

```
INLET inlet6 = islot5 fslot6 greg4.b thr9
```

9 Opcodes

9.1 Type Conversion

Type conversion must be explicit. Supported are conversions between integer and floats (ITOF, FTOI), between characters and integer (CTOI, ITOC), and between small pointers and integers (PTOI, ITOP).

Examples:

```
ITOF greg4 = greg5
FTOI ireg4 = freg5
```

9.2 Split Phase Instructions

Syntax:

```
IFETCH inlet5 = preg5[ireg6]
RALLOC inlet6-inlet10 = cb_foo
RALLOC inlet6-inlet10 = cb_foo inlet3 = ireg0.g
RSEND preg4 inlet3 = freg4.g
RCALL inlet7-inlet10 = cb_foo inlet3 = ireg0.g
```

9.3 Arithmetic Operations

Unary Operations:

Binary Operations: `ADD.[IF], SUB.[IF], EQ.[GFIPCBS], ...`

Syntax:

```
ADD.F greg5 = greg9 freg4
ADD.F freg5 = freg5 greg4
```

10 Questions

```
rcall
gc_foo
```

character constants