Tiger – An Interpreter Generation Tool

Kevin Casey[1], David Gregg[1], and M. Anton Ertl[2]

[1] Department of Computer Science, Trinity College, Dublin 2, Ireland
{Kevin.Casey, David.Gregg}@cs.tcd.ie
[2] Institut für Computersprachen, TU Wien, A-1040 Wien, Austria
anton@complang.tuwien.ac.at

Abstract. `Tiger` (Trinity Interpreter GEneratoR) is a new interpreter generator tool along the lines of `vmgen`, but with significant improvements in flexibility and feedback. Support for important new features such as instruction specialisation, replication and improved analysis of code at runtime are presented. A simple 'C' virtual machine imported into `Tiger` is used for demonstration purposes. Various realistic benchmarks (such as sorting and Davis-Putnam backtracking algorithms) are used to show the utility of these new features in `Tiger`.

1 Introduction

`Tiger` is a new interpreter generator tool along the lines of `vmgen`[1], but with significant improvements in flexibility and feedback. Some of these features which are to be demonstrated are outlined briefly in the remainder of this document.

2 Input Language

A typical opcode defined in `Tiger` is depicted as follows:

```
ADD SP( int a, int b - int c )
    IP( - next);
    c=a+b;
```

The first token is the opcode name, then followed either by the stack behaviour (`SP`) or the instruction stream behaviour (`IP`). The stack behaviour specifies what types and instances needs to be popped off the stack before the core of the opcode is to be executed and what is to be pushed onto the stack after the core of the instruction has completed. The '-' symbol represents the separator between what is to be popped and what is to be pushed in the stack descriptor. The instruction stream behaviour allows us to specify what operands are to be loaded from the instruction stream (none in this case). The '-' symbol represents the end of the current instruction. The keyword `next` indicates that another instruction will follow in the instruction stream. `Tiger` uses the stack and instruction stream descriptors supplied and the code core specified to generate 'C' code for the instruction.

R. Bodik (Ed.): CC 2005, LNCS 3443, pp. 246–249, 2005.

3 Instruction Specialisation

Often in a compiler we find that certain operands occur in combination with particular opcodes quite frequently. For example, if we find that a large number of `PUSHINT 0` instructions occur in our interpreter, we might consider replacing it with a single instruction `PUSHINT_0`. The advantage of this is that the `PUSHINT_0` instruction no longer requires an extra read from the instruction stream (to retrieve the operand), since the operand is 'hardwired' into the instruction. For example:

```
+SPECIAL PUSHINT 0;
```

generates code for the `PUSHINT` instruction seen above, but where the operand `a` is specialised to 0 (#defined to 0). This eliminates the instruction stream read. `Tiger` provides support for a translation-time specialisation of opcodes in the form of a variable argument `vm_specialise` macro. For example, when translating an opcode with a single operand:

```
OPCODE'=vm_specialise(OPCODE,OPERAND);
```

This macro will attempt to find a specialised version of `OPCODE,OPERAND` and will return the original opcode if no specialisation has been found, or the specialised version if one has been found. For unspecialised opcodes, the application of the macro has no computational cost.

4 Instruction Replication

One strategy to improve branch prediction accuracy in interpreters is to create copies of commonly occurring opcodes. The idea here is that using multiple copies of an opcode increases the number of entries in the Branch Target Buffer (BTB) due to the extra dispatches in the code. `Tiger` supports the creation of replications in the following manner.

```
+ALIAS OPCODE COUNT;
```

This creates `COUNT` copies (in addition to the original) of `OPCODE`. `Tiger` also creates a macro `vm_alias` and supporting data structures to facilitate the inclusion of these aliases into the instruction stream (replacing the original). During the code-translation phase this macro can be used to replace original versions of opcodes with their copies in the following way:

```
OPCODE'=vm_alias(OPCODE)
```

If the opcode is not replicated at all, then `OPCODE'=OPCODE` (and there is no computational cost associated with the macro). For replicated instructions however, each successive call to `vm_alias` yields the next copy of the opcode in a cyclical order. `Tiger` maintains an array of counters and replication counts to support this approach.

5 Superinstructions

The generation of compound instructions, or superinstructions is quite straightforward in `Tiger`. If one finds that the sequence of instructions `PUSHINT ADD` occur quite frequently, one could define a superinstruction `PUSHINT_ADD` as follows:

```
PUSHINT_ADD = PUSHINT ADD;
```

Superinstruction parsing routines for greedy parsing and optimal parsing are also supplied with `Tiger`. The parse tables for all superinstructions are combined into a large compressed Deterministic Finite-state Automata which is accessed via the supplied routines. The actual implementation of the DFA is as a number of overlapping hashtables, one hashtable for each set of transitions from a particular state.

6 Specialised Superinstructions

`Tiger` also allows the creation of specialised superinstructions. In the example above, we came across the `PUSHINT_ADD` superinstruction. If we encountered the instruction sequence `PUSHINT 1 ADD` sequence, we could decide to create a specialised superinstruction such as:

```
PUSHINT1_ADD = PUSHINT 1 ADD;
```

`Tiger` will then generate the superinstruction and modify the parsing tables so that this instruction will be added automatically when applicable.

7 Other Optimizations

Early Loading: On some architectures it is advantageous to retrieve the address of the next instruction as early in the current instruction as possible. In opcodes where the keyword `next` appears in the instruction stream specifier, `Tiger` will automatically retrieve the address of the next instruction from that slot in the instruction stream. This will happen at the beginning of the current instruction. This optimisation can be turned on or off easily, making a determination of its utility relatively straightforward.

Deferred Reading/Writing allows the reading or writing of an item in the stack/instruction stream descriptors to be deferred until the programmer wishes it to happen. This mechanism is accomplished by the use of the `+DEFER` directive which is used in combination with automatically generated macros. For example, a `+DEFER` might be useful in a conditional branch where we do not want to load the target address from the instruction stream unless we have tested the condition and are sure the branch is to take place.

8 Diagnostics

Histogram: Turning on the histogram option creates a logfile containing a frequencies of all opcode calls. A tool is provided that interprets this data and

generates a Scaled Vector Graphic (SVG) file containing a histogram that can be viewed in a web-browser.

Indirect Branch Data: Virtual machine interpreters execute an indirect branch for each VM instruction executed. The prediction accuracy of these branches has a huge impact on running time. Tiger generates data and graphs which allow us to visualize the order in which instructions are executed, and estimate the indirect branch prediction accuracies. Figure 1 depicts a sample output of this tool, showing the transitions between VM instructions, along with their frequency and an estimate of their indirect branch misprediction rate.

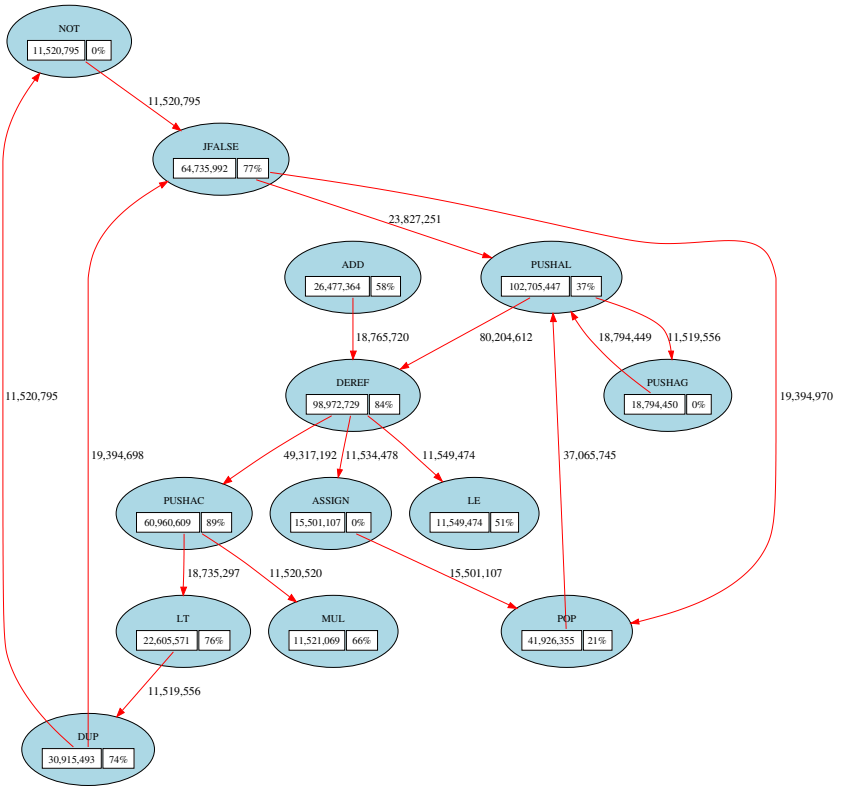

Fig. 1. Automatically generated instruction-transition graph

Reference

1. M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. vmgen — A generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.