An Optimising Compiler for Generated Tiny Virtual Machines

Doug Palmer Pavan Sikka Philip Valencia
Peter Corke
CSIRO ICT Centre
{*Doug.Palmer,Pavan.Sikka,Philip.Valencia,Peter.Corke*}*@csiro.au*

Abstract

VMSCRIPT *is a scripting language designed to allow small programs to be compiled for a range of generated tiny virtual machines, suitable for sensor network devices. The* VMSCRIPT *compiler is an optimising compiler designed to allow quick re-targeting, based on a template, code rewriting model. A compiler backend can be specified at the same time as a virtual machine, with the compiler reading the specification and using it as a code generator.*

1. Introduction

Many experimental sensor networks are designed to be essentially homogeneous, with identical sensor devices connecting to perhaps a single gateway. In many cases, each node of the sensor network runs an identical program. However, commercially deployed sensor networks can be expected to be heterogeneous, for reasons of network life, architectural issues and changing use of the environment the network is part of. The broad motivation behind the work described is to develop a software engineering stack that allows application programmers to develop efficient application programs, independent of the precise structure of the sensor network that will host them.

As part of this aim, a technique of generating families of virtual machines (VMs), suitable for hosting programs in a heterogeneous environment has been developed.[5] We have completed an initial implementation that allows us to run programs on a Fleck.[6] The target virtual machines can differ greatly in terms of instruction set, storage management and other features, ranging from minimal systems that support a handful of instructions and pieces of data to garbage-collected heaps. A specific feature of this approach is that *subsets* of a specification can be generated, allowing families of related VMs to be generated. This flexibility allows experimentation (either by a human or a machine) in finding the right mix of generic and environment- and platform-specific features for an environment. The extreme flexibility of the generated virtual machines results in different orientation to the Maté/Bombilla/TinyScript programming model.[3]

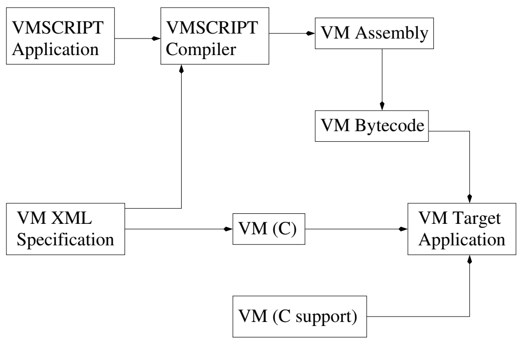

Figure 1. Overview of VMSCRIPT Tools and Objects

This paper presents the next step of this work, a compiler for a simple scripting language, VMSCRIPT, that can be used to generate efficient byte code for the broad class of virtual machines that can be generated. When specifying a VM, the designer supplies a description to a generator. An extension to the description allows the designer to also describe the instruction sequences that will implement a small number of basic operations. Ideally, specifying an acceptably efficient compiler for a virtual machine should take no more effort than specifying the virtual machine itself.

Figure 1 shows the various VMSCRIPT-related tools and objects.

2. The Compiler

An initial version of the VMSCRIPT compiler simply relied on the stack architecture of the generated VMs and generated code based on a simple parse tree traversal and templating process. While correct, the generated programs could hardly be described as efficient and a rewrite was undertaken to allow more sophisticated analysis and code gen-

```
<template>
  <cost domain="size"
    priority="1" cost="2"/>
  <pattern>
    <get source="2" target="?Mul"/>
    <op type="multiply"
      arg1="?Source:integer"
      arg2="?Mul" output="?Target"/>
  </pattern>
  <transform>
    <acquire source="?Source"/>
    <opcode code="dup"/>
    <opcode code="add"/>
    <place output="?Target"/>
  </transform>
<template>
```

Figure 2. Example Code Generation Template

eration. The compiler now uses a number of compilation techniques, derived from the extensive literature on compilers, such as intermediate representation, abstract interpretation and single static assignment form to globally analyse an input program and produce optimal code.[4, 1, 2]

A key feature of the IR chosen for VMSCRIPT is the need to provide support for easy specification of multiple targets. To allow this, a relatively small set of intermediate forms are permitted, something the simplicity of VMSCRIPT allows. Despite the generated virtual machines being stack-based, named temporary variables are used in the intermediate form, since these are easier to manipulate during most operations, with stack mapping occurring at code generation.[7]

Abstract interpretation is used in a number of optimisations in the compiler: type inference, stack estimation, constant propagation and dead code removal.

2.1. Code Generation

Peephole optimsation at several levels and code generation uses a uniform rewriting strategy.[8] The compiler uses techniques inspired by logic programming.[9] Transformations are encoded using logical variables. Any input that unifies with the input pattern is transformed into the corresponding output pattern. Sequences of arbitrary intermediate representation code are repeatedly replaced by more basic IR sequences until just ground forms that can be directly translated into assembly, remain. Figure 2 shows an example code generation template, in this case one which generates special case code for multiplication by two. Virtual machine subsets are handled by ignoring templates that would generate instructions or store references that are not present in the subset.

The IR uses temporary variables instead of stack positions for intermediate values. During code generation, the compiler tracks the stack position of temporary variables, which can then be used with the special Place and Acquire IRs to generate stack manipulation sequences.

3. Conclusions and Further Work

The VMSCRIPT compiler is a very flexible compiler for generated virtual machines. In order to use the compiler with a virtual machine, approximately 45–60 simple rewriting templates need to be specified; A typical generated virtual machine will have some 30–40 instructions. The templates can be incrementally added to, allowing a repository of optimal sequences to be built up over time.

There are a number of parts of the compiler that can be made more sophisticated: the handling of ambiguous types, stack estimation techniques, code generation cost analysis and constant propagation cost analysis.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[2] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, California, 1977.

[3] P. Levis, D. Gay, and D. Culler. Bridging the gap: Programming sensor networks with application specific virtual machines. Technical Report UCB//CSD-04-1343, UC Berkeley, Aug. 2004. `http://www.cs.berkeley.edu/~pal/pubs/mate-tr.pdf`.

[4] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[5] D. Palmer. A virtual machine generator for heterogeneous smart spaces. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM04)*, pages 1–11, San Jose, California, May 2004.

[6] P. Sikka, P. Corke, and L. Overs. Wireless sensor devices for animal tracking and control. In *Proceedings of the 29th Conference on Local Computer Networks, Workshop on Embedded Networked Sensors (EmNetS-I)*, pages 446–454, Tampa, Florida, Nov. 2004.

[7] R. Vallée-Rai, P. Co, E. Gagnon, L. Hedren, P. Lam, and V. Sundaresan. Soot — a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 13–23, Mississauga, Canada, 1999.

[8] E. Visser. A survey of rewriting strategies in program transformation systems. In *Proceedings of the Workshop on Reduction Strategies in Rewriting and Programming (WRS 01)*, Utrecht, The Netherlands, May 2001.

[9] D. H. D. Warren. Logic programming and compiler writing. *Software, Practice and Experience*, 10(2):97–125, 1980.