PIPS/SAC: SIMD Architecture Compiler

François Ferrand

ENST Bretagne

August 31, 2024

Introduction

This document defines and describes the data structures used by SAC, the SIMD Architecture Compiler. SAC is a new PIPS phase, which allows to generate code optimized for architectures supporting multimedia instruction sets, such as MMX, SSE or VIS.

```
import entity from "ri.newgen"

import expression from "ri.newgen"

import statement from "ri.newgen"

import reference from "ri.newgen"

import reduction from "reductions_private.newgen"

reductionInfo = persistent reduction x count:int x persistent
vector:entity
```

Opcodes and opcode classes

```
opcode = name:string x vectorSize:int x argType:int* x cost:float

tabulated opcodeClass = name:string x nbArgs:int x
opcodes:opcode*
```

Statement matching

Statement matching is used to detect "patterns" in the code. It works on the expression tree representing the program. The actual patterns are read from a file, to create a matchTree that is used to efficiently parse the expression tree. This process returns a list of matches, indicating the various opcodes that can thus be generated, and with which arguments.

A `patternArg` specifies how an argument is to be generated. It can be an integer constant, with the specified value, or extracted from the actual expression tree.

patternArg = `static:int + dynamic:unit`

A `pattern` identifies what to generate. It specifies the opcode class corresponding to the pattern, as well as a list that can be used to translate from original statement references or constants to arguments for the opcode.

patternx = `class:opcodeClass x args:patternArg*`

matchTreeSons = `int->matchTree`

External operator_id_sons

operator_id_tree = `id:int x sons:operator_id_sons`

`matchTree` is a structure used to efficiently identify patterns corresponding to a statement. When traversed, it can thus map a statement to a list of patterns that can be used.

matchTree = `patterns:patternx* x sons:matchTreeSons`

A `pattern` can be translated into a `match` by mapping the arguments properly. The argument list in a `match` is constructed from the arguments of the statement, following the rules of pattern arguments (list of `patternArg`).

match = `type:opcodeClass x args:expression*`

Statement information

simdstatement = `opcode x nbArgs:int x vectors:entity[16] x arguments:expression[48]`

Transformation

transformation = `name:string x vectorLengthOut:int x subwordSizeOut:int x vectorLengthIn:int x subwordSizeIn:int x nbArgs:int x mapping:int[16]`