HPBCG Documentation
High Performance Binary Code Generation

Henri-Pierre Charles

February 6, 2009

Contents

1 Introduction

HPBCG is a tool which help to build binary code generator.

1.1 What is it ?

A binary code generator is a tool which can generate binary code (runnable), at run-time, without using assembly (textual) representation.

It can be usefull in a lot of situations :

- For vector code generation
- For multimedia code generation
- For code optimization at run-time using data as optimizing parameter

1.2 Motivation

Actually (january 2009) computer architecture reach a complexity point which leed to

- compiler which are unable to vectorize or use multimedia instructions easily
- a bad use of huge register set. Compiler are still using algorithm allocator which came from ages where register are rare.
- data are the main important parameter that actual compiler cannot take into account because code generation is done at static compile time.

1.3 Related projects

ccg C Code Generator [2] is the direct predecessor of **HPBCG** . **HPBCG** differ from ccg on many points :

- The architecture description has been simplified to the strict minimal binary description.
- The source parser has been rewrited using antlr, which greatly simplify the porting process

lightning `http://www.gnu.org/software/lightning/` is an other tool (which has take the architecture description from ccg), but it does not allow a full use of the large register set, the vector instructions or the special multimedia instructions.

2 Actual status

2.1 Working targets

Cell spu Working draft

Itanium Working draft.

Power4 Initial stage

Working demo / architecture :

Demo name	Cell-spu	Itanium	Power4
simple-multiply	ok	ok	ok
rpn	ok [1]	compile	ok

2.2 Tested platforms

cell-spu Playstation 3 / linux yellow dog

power4 Playstation 3 / linux yellow dog

itanium Bull ia64 platform / linux

3 Todo

All targets Things to be done

- Write a .isa verifyer, to check the insn coherency, the opcode usage, ...
- Include expression in the binary description.
- Find a way to hangle instructions aliases

itanium Things to be done :

1. Verify the intruction tabuled scheduler in `ia64-utils.h`
2. Add a mini scheduler which allow to break bundle into sub-bundle if a schedule does not exist
3. Look at the L+X instruction : how to choose the template value ?
4. How to choose between two possible templates ?

cell Things to be done :

1. Add more working examples
2. Hide the worker communication somewhere
3. Solve the divide problem

power4 Things to be done :

- Complete the .isa file

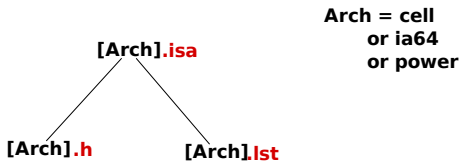

Figure 1: Installation scheme

4 Using HPBCG

4.1 Installing HPBCG

HPBCG sould work on any reasonnable unix like target. The requirements are :

antlr The parser is implemented with `antlr.org`

java As antlr is writed in java. But java is not needed at run-time, only at static compile time.

HPBCG contain two parts :

Architecture description contains the architecture description and the parser used to generate the macro instructions. This part is in the `src/isatobcg` directory.

Parser contains the parser in charge of the translation from the `.hg` file to the `.c` file. This par is in the `src/parser` directory.

4.2 Using hpbcg

5 Compilettes examples

5.1 simple-multiply

This example is very simple, it's just a proof of concept.

The obtained result is

```
turner:simple-multiply/>./simple-multiply-cell 42
Code generation for multiply value 42
Code generated
  1   2   3   4   5   6   7   8   9  10
 42  84 126 168 210 252 294 336 378 420
```

The `simple-multiply` program should generate a specialized version of a very simple program. The non specialized version is :

```
int multiply (int a, int b)
{
  return a * b;
}
```

This compilette will specialize this code with an "optimized" version at runtime. For example the previous code will be specialized as

```
int multiply (int a)
{
  return a * 42;
}
```

This specialized version shoud be faster than the previous one because

- the code is less specialized (Well, for a 1 instruction function, it's not so evident, but you get the idea)
- the function contain less parameter, which use less memory and less register

5.1.1 Cell version

Use the command `make cell` to build the program.

The `cell` version contain 2 files :

simple-worker-cell.c contain the initial SPU code. It will

1. download the binary code in a buffer
2. use this buffer as a function
3. call this function for all incoming parameter

simple-multiply-cell.hg is the code for the PPU. It will

1. generate a specialized code depending on the data given by the user.
2. sent it to the worker
3. use the worker 10 times for printing a array of multiplied values

5.1.2 Itanium version

Use the command `make ia64` to build the program `simple-multiply-ia64`. Run it !

5.1.3 Power4 version

Use the command `make power4` to build the program `simple-multiply-power4`. Run it !

6 Porting HPBCG

Porting **HPBCG** to a new architecture should be as simple than the architectural model you plan to target.

6.1 Architecture description

The actual version contain processor description for

cell.isa This file contain all SPU instruction description

ia64.isa This file contain all instruction set description

power4.isa This file contain all instruction set description

A processor description file should contain

Comments A comments line is a line starting with #

Arch length A line containing the architecture name and the bit lenght of one instruction. For instance the first lone of the `cell.isa` containing the cell description contain :

```
cell 32
```

Instruction description Each line of this part describe one machine instruction. Each line is divided in two part separated by a `|`. The general for is :

```
Binary description | Syntax description
```

For instance the `cell.isa` description contain a line with :

```
00011000000  r3_7 r2_7  r1_7   | a   r1,r2,r3
```

The Binary description contains bits fields describing the instruction. In the previous example we have 4 bit fields

00011000000 witch is the opcode of the instruction comming from the manual[1] page 55.

Registers description `r3_7` which mean that the 3nd register should be encoded on 7 bits.

Syntax description contains the instruction syntax allowed to be used in compilettes.

6.2 isatobcg Parser

6.3 HPBCG Parser

7 Assembly languages

This part is devoted to different assembly langaguages that **HPBCG**

7.1 cell-spu

Integer register names one of

- `$lr`, `$sp`
- `$0` : link register, `$1` : stack pointer, `$2` : volatile
- `$3` .. `$79` function arguments & return value, volatile
- `$80` .. `$127` local variables, non-volatile

Calling convention

7.2 power and cell-ppu

Integer register name one of

- `r0` .. `r32`
- `r3` 1st function argument & return value

Calling convention

7.3 ia64

Registers name one of

- `r0` .. `r128`, `f0` .. `f0` are floating point or multimedia registers
- `r0` is always 0
- `r1` is always 1
- `f0` is always 0.0

Calling convention depending on the used datatype, differents register can be used :

- `r32` is the first integer parameter, `r33` the second, etc.
- `f8` is the first floating point parameter, `f9` the second, etc.
- `r8` or `f8` is the return value depending on the used data type.

8 Reporting bug

Please mail your comments to `mailto:hpc@prism.uvsq.fr`

References

[1] IBM. *Synergistic Processor Unit, Instruction Set Architecture*, August 2005.

[2] I. Piumarta, F. Ogel, and B. Folliot. Ynvm: dynamic compilation in support of software evolution. 2001.