

HPBCG Documentation

High Performance Binary Code Generation

Henri-Pierre Charles

April 10, 2009

Contents

1	Introduction	2
1.1	What is it ?	2
1.2	Motivation	2
1.3	Related projects	2
2	Actual status	3
2.1	Working targets	3
2.2	Tested platforms	3
3	Todo	3
4	Installing HPBCG	4
5	Using HPBCG	4
5.1	Outside parser	4
5.2	Inside parser	5
6	Complettes examples	5
6.1	simple-multiply	5
6.1.1	Cell version	6
6.1.2	Itanium version	7
6.1.3	Power4 version	8
7	Porting HPBCG	8
7.1	Architecture description	9
7.2	isatobcg Parser	9
7.3	HPBCG Parser	9
8	Assembly languages	9
8.1	cell-spu	10
8.2	power and cell-ppu	10
8.3	ia64	10

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 useful in a lot of situations :

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

1.2 Motivation

Actually (January 2009) computer architecture reach a complexity point which lead 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

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

- The architecture description has been simplified to the strict minimal binary description.
- The source parser has been rewrote 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 **cgc**), 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

Itanium Working draft.

Power4 • Basic power4 instruction set : Working draft

- Cell spu : Working draft
- FP2 extension : Working draft
- AltiVec extension : To be integrated
- Arm : To be integrated

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	power4	itanium
PS3 linux yellow dog	Sony/PS3	Bull ia64 / linux
	IBM/BlueGene	

Java on PS3 : <http://www.ibm.com/developerworks/java/jdk/linux/> could be useful

3 Todo

All targets Things to be done

- Improve the .isa verifier, to check the insn coherency, the opcode usage.
- Find a way to handle instructions aliases

itanium Things to be done :

1. Verify the instruction tabulated 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

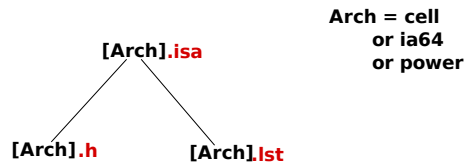


Figure 1: Installation scheme

3. Solve the precision divide problem

power4 Things to be done :

- Complete the .isa file
- Test FP2 on bluegene

4 Installing HPBCG

HPBCG should work on any reasonable unix like target. The requirements are :

Build time	java and antlr 3.x
Install time	nothing
Static compile time	java and antlr 3.x
Run-time	nothing

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 parser is in the **src/parser** directory.

This parser translate pseudo assembly file into fast binary code generator.

5 Using HPBCG

Mainly **HPBCG** is a parser which translate C code whith special parts to a real C code. The text contain two parts inside compilette block or outside

5.1 Outside parser

The outside parser only recognize the **#cpu** token

- **#cpu** : this allow to define the architecture which will be used in the future compilettes blocks

The actual cpu recognized cpu are

- #cpu power4
- #cpu ia64
- #cpu cell

5.2 Inside parser

The compilette block is delimited by theses tokens :

- #[: beginning of compilette block
-]# : end of compilette block

Inside a compilette block a programmer can write standard assembler instructions. The used syntax for the assembly instructions is defined by the previous #cpu. An example of CELL instruction :

```
mpyi    $3, $3, 4
```

The special characters (and) are used to insert C expressions inside the assembly code. Please note that these expressions will be evaluated only at run-time.

It allow to write parametrized instruction. An example of CELL instruction :

```
mpyi    $3, $3, (multiplyValue)
```

6 Compiettes examples

6.1 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 comppilette will specialize this code with an “optimized” version at run-time. For example the previous code will be specialized as

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

Please note that the value 42 is choosed by the user and can not be include into the code before run-time.

This specialized version should 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

6.1.1 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

The cell ccomppilette is simple as :

```
1 #cpu cell
2
3 typedef int (*pifi)(int);
4 pifi multiplyFunc;
5
6 pifi multiplyCompile(int multiplyValue)
7 {
8     insn *code= (insn *)_malloc_align(1024, 7);
9     printf("Code_generation_for_multiply_value_%d\n", multiplyValue);
10    #[
11        .org      code
```

```

12         mpyi    $3, $3, (multiplyValue)
13         bi $lr
14     ]#;
15     printf("Code_generated\n");
16     return ( pifi)code;
17 }

```

line 3 and 4 define a type pointer on a function which take one parameter

line 8 alloc a block in memory where the function will be generated.

line 10 The token `#[` define the beginning of the compilette block

line 11 define the beginning address of the function

line 12 and 13 will generate 2 binary instructions. The `mpyi` instruction take the `multiplyValue` as parameter.

`bi $lr` is the return instruction.

line 14 The token `]#` define the end of the compilette block

6.1.2 Itanium version

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

```

1 #cpu ia64
2
3 typedef int (* pifi)(int);
4 pifi multiplyCompile(int multiplyValue)
5 {
6     insn *code= (insn *)calloc(1024, sizeof (insn));
7     printf("Code_generation_for_multiply_value%d\n", multiplyValue);
8     #[
9         .org    code
10        .proc    code+16, 0
11        mov     r33 = (multiplyValue)
12        nop.i 22
13        nop.i 22 ;;
14
15        setf.sig f32 = r32
16        setf.sig f33 = r33
17        nop.i 22 ;;
18
19        nop.m 22
20        xmpy.l  f32 = f32 , f33
21        nop.i 22 ;;

```

```

22
23         nop.m 22
24         getf.sig r8= f32
25         br.ret.sptk.many b0 ;;
26     ]#;
27     iflush (code, asm_pc);
28     printf("Code_generated\n");
29     return (pifi)code;
30 }

```

The Itanium version is more complicated due to the fact that this processor does not have integer multiplication.

The `setf.sig` instruction convert integer values to floating point values and the `getf.sig` does the opposite.

6.1.3 Power4 version

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

```

1 #cpu power4
2
3 typedef int (*pifi)(int);
4 pifi multiplyCompile(int multiplyValue)
5 {
6     insn *code= (insn *)calloc(1024, sizeof (insn));
7     printf("Code_generation_for_multiply_value_%d\n", multiplyValue);
8     #[
9         .org      code
10        mulld r3, r3, (multiplyValue)
11        blr
12    ]#;
13    iflush (code, asm_pc);
14    printf("Code_generated\n");
15    return (pifi)code;
16 }

```

The compilette code is similar to the cell one except for the `mulld` instruction.

7 Porting HPBCG

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

7.1 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 length 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 coming 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 complettes.

7.2 isatobcg Parser

7.3 HPBCG Parser

8 Assembly languages

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

8.1 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

8.2 power and cell-ppu

Integer register name one of

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

Calling convention

8.3 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, different 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.

9 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.