HPBCG Documentation
High Performance Binary Code Generation

Henri-Pierre Charles

June 2, 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 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

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 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 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

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]	ok	ok

2.2 Tested platforms

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

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 verifyer, 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. Inprove the mini scheduler which allow to break bundle into sub-bundle if a static schedule does not exist
3. Look at the L+X instruction : how to choose the template value ?

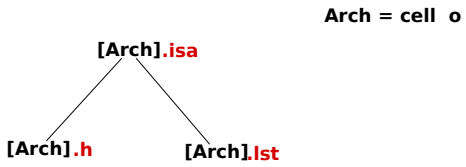

Figure 1: Installation scheme

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 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.

4.1 Installation

HPBCG can be installed everywhere, the usual location should be `/usr/local`.

In the main directory type the commands :

```
make build
sudo make install WHERE=/usr/local
```

1. The first one will create all the `.h` files
2. The second one will install the files in the `WHERE` directory.

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 wil be evaluated only at run-time.

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

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

6 Compilettes 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 compilette 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 ccompilette 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
5  pifi multiplyCompile(int multiplyValue)
6  {
7    insn *code= (insn *)calloc(1024, sizeof (insn));
8    printf("Code generation for multiply value %d\n", multiplyValue);
9    #[
10         .org    code
11         .proc   code+16, 0
12         mov     r33 = (multiplyValue)
13         setf.sig f32 = r32
14            nop.i 0 ;;
15         setf.sig f33 = r33              ;;
16
17         xmpy.l  f32 = f32, f33          ;;
18
19         getf.sig r8= f32
20         br.ret.sptk.many b0 ;;
21   ]#;
22   iflush (code, asm_pc);
23   printf("Code generated\n");
24   return (pifi)code;
25 }
```

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.

The `nop.i` instruction allow the scheduler to find a bundle with these 3 instructions.

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         mulli 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 `mulli` instruction.

6.1.4 Power4 bluegene version

Use the command `make bluegene` to build the program

To run it you can do it :

- interactively : `bgrun -np 64 -mode VN -mapfile TXYZ -exe ./simple-multiply-bluegene`
- in a batch :

```
# @ job_name = simple-multiply-bluegene
# @ job_type = BLUEGENE
# @ output = $(job_name).out
# @ error = $(output)
# @ wall_clock_limit = 0:00:05
# @ bg_size = 64
# @ queue
mpirun -mode VN -np 256 -mapfile TXYZ -exe ./simple-multiply-bluegene
```

The generated code is similar to the power4 version with differences. All the processors compute a multiplication table depending on their processor number. The result is sent to the processor 0 which print the results :

```
Proc 0:    1    2    3    4    5    6    7    8    9   10
Proc 0:    0    0    0    0    0    0    0    0    0    0
Proc 1:    1    2    3    4    5    6    7    8    9   10
Proc 2:    2    4    6    8   10   12   14   16   18   20
Proc 3:    3    6    9   12   15   18   21   24   27   30
Proc 4:    4    8   12   16   20   24   28   32   36   40
Proc 5:    5   10   15   20   25   30   35   40   45   50
Proc 6:    6   12   18   24   30   36   42   48   54   60
Proc 7:    7   14   21   28   35   42   49   56   63   70
Proc 8:    8   16   24   32   40   48   56   64   72   80
Proc 9:    9   18   27   36   45   54   63   72   81   90
Proc 10:   10   20   30   40   50   60   70   80   90  100
Proc 11:   11   22   33   44   55   66   77   88   99  110
../..
```

6.2 rpn

rpn is a more complicated example. It compute conversion table from Celcius to Farenheit by "compiling" RPN expression. The interesting part of the program is the code generation which is build with

```
pifi c2f= rpnCompile("9*5/32+");
pifi f2c= rpnCompile("32-5*9/");
```

The main part is the RPN code generator which convert RPN expression to a binary code function. In this compilette example the registers are used as a stack.

The first register contain the number to convert.

The result should be :

```
allaoua:rpn/>./rpn-ia64

C:  0  10  20  30  40  50  60  70  80  90 100
F: 32  50  68  86 104 122 140 158 176 194 212

F: 32  42  52  62  72  82  92 102 112 122 132 142 152 162 172 182 192 202 212
C:  0   5  11  16  22  27  33  38  44  50  55  61  66  72  77  83  88  94 100
```

6.2.1 power4 version

```
1  pifi rpnCompile(char *expr)
2  {
3    insn *code= (insn *)calloc(64, sizeof(insn));
4    int top= 3;
5  #ifdef _IBMR2
6    code = code + 2;
7  #endif
8    #[    .org    code   ]#;
9
10   while (*expr)
11     {
12       char  buf[32];
13       int n, tmp ;
14       if (sscanf(expr, "%[0-9]%n", buf, &n))
15       {
16         expr+= n - 1;
17         if (top == 10)
18           {
19             fprintf(stderr, "expression too complex");
20             exit(0);
21           }
22          ++top; tmp = atoi(buf);
```

```
23          #[
24            subf  r(top), r(top), r(top)
25            ori   r(top), r(top), (tmp)
26          ]#;
27        }
28      else if (*expr == '+')
29        { --top; #[ add  r(top), r(top), r(top+1)  ]# }
30      else if (*expr == '-')
31        { --top; #[ subf r(top), r(top), r(top+1)  ]# }
32      else if (*expr == '*')
33        { --top; #[ mullw r(top), r(top), r(top+1) ]# }
34      else if (*expr == '/')
35        { --top; #[ divw  r(top), r(top), r(top+1) ]# }
36      else
37        {
38          fprintf(stderr, "cannot compile: %s\n", expr);
39          abort();
40        }
41      ++expr;
42    }
```

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 compilettes.

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.