HPBCG Documentation
High Performance Binary Code Generation

Henri-Pierre Charles

February 10, 2010

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 is build to be

- Depending on the target speed may vary, but FAST mean some instructions clocks per generated instruction
- Easy to port : Target a new architecture need only a simple low level description. For example the `mulli` instruction from the power 4 is describe by :

```
001110  r1_5  r2_5  i1_15-0  | addi  r1, r2, i1
```

  Then, the process to build multiple dynamic generator for this instruction is completely automatic.

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 [4] 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

Power4
- Basic power4 instruction set : Working
- FP2 extension : Working
- Altivec extension : Working

Cell spu : Working draft

Arm : To be integrated

x86 : Initial version

Working demo / architecture :

Demo name	Cell-spu	Itanium	Power4	x86
simple-multiply	ok	ok	ok	ok
rpn	ok [1]	ok	ok	ok
mendelbrot	ok	TODO	ok	TODO

2.2 Tested platforms

cell-spu	power4	itanium	x86
PS3 linux yellow dog	Sony/PS3 IBM/BlueGene	Bull ia64 / linux	FreeBSD

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 ?
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

x86 Thinks to be done

- Complete isa description

4 Installing HPBCG

4.1 Installation dependancies

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

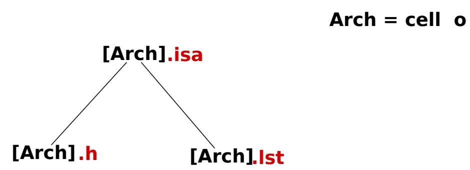


Figure 1: Installation scheme

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.

4.2 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 : the classical C code or the “compilette” block code.

5.1 Outside parser

The outside parser only recognize the `#cpu` token and replace it by a `#include <hpbcg-XXX.h>`

- `#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`
- `#cpu power`
- `#cpu x86`

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

This section show basic compilettes examples the article [2] describe the first compilette examples and results.

6.1 simple-multiply

This example is very simple, it's just a proof of concept. The `simple-multiply` program generate a specialized version of a very simple program. The non specialized version is :

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

The compilette will specialize this code with an “optimized” version at run-time. The following code will be specialized as

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

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
```

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 in the stack and less register

6.1.1 Cell version

Use the command `make cell` to build the program. For this example, the compilation chain is :

```
turner:simple-multiply/>make clean cell
../..
hpbcg simple-multiply-cell.hg > simple-multiply-cell.c
cc ../.. -c -o simple-multiply-cell.o simple-multiply-cell.c
cc -lspe2 simple-multiply-cell.o -o simple-multiply-cell
spu-gcc ../.. -g -o simple-worker-cell simple-worker-cell.c

turner:simple-multiply/>./simple-multiply-cell 42
Code generation for multiply value 42
0x10025080 : mpyi_iRRI r3 r3 0x2A
0x10025084 : bi_iR r0
Code generated
   1    2    3    4    5    6    7    8    9   10
  42   84  126  168  210  252  294  336  378  420
```

This differents steps are describe here :

1 `make cell`

3 **HPBCG** translate .hg file to a plain C code

5 & 6 C compilation (for clarity swithes has been removed)

8 Run time, at the startup time, the compilette generate the binary code

14 The binary generated code is used. The binary code is printed thanks to the switches `ASM_DEBUG` and `WITH_HPBCG_FUNCTIONS` used at compile time.

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 !

```
#cpu ia64

typedef int (*pifi)(int);

pifi multiplyCompile(int multiplyValue)
{
  insn *code= (insn *)calloc(1024, sizeof (insn));
  printf("Code generation for multiply value %d\n", multiplyValue);
  #[
        .org     code
        .proc    code+16, 0
        mov      r33 = (multiplyValue)
        setf.sig f32 = r32
           nop.i 0 ;;
        setf.sig f33 = r33               ;;

        xmpy.l   f32 = f32, f33          ;;

        getf.sig r8= f32
        br.ret.sptk.many b0 ;;
  ]#;
  iflush (code, asm_pc);
  printf("Code generated\n");
  return (pifi)code;
}
```

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 !

```
#cpu power4

typedef int (*pifi)(int);
pifi multiplyCompile(int multiplyValue)
{
  insn *code= (insn *)calloc(1024, sizeof (insn));
  printf("Code generation for multiply value %d\n", multiplyValue);
  #[
        .org    code
        mulli r3, r3, (multiplyValue)
        blr
  ]#;
  iflush (code, asm_pc);
  printf("Code generated\n");
  return (pifi)code;
}
```

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
../..
```

6.1.5 x86 version

Use the command `make x86` to build the target `simple-multiply-x86`, run it !

```
1  pifi multiplyCompile(int multiplyValue)
2  {
3    insn *code= (insn *)calloc(1024, sizeof (insn));
4    printf("Code generation for multiply value %d\n", multiplyValue);
5    #[
6          .org    code
7          push    %ebp
8          mov     %esp,%ebp
9          mov     0x8(%ebp),%eax
10         imul    $(multiplyValue),%eax,%eax
11         pop     %ebp
12         ret
13   ]#;
14   iflush (code, hpbcg_asm_pc);
15   printf("Code generated\n");
16   return (pifi)code;
17 }
```

The `x86` version is very simple, the two main instructions are lines 9 and 10 (mov and imul). The other instructions are context management.

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
```

The `main.c` file contain 3 different version of the same program.

1. The interpreted version which use a stack for function evaluation
2. The “compilette” generated program, which use a dynamic generated version of the code.
3. The static compiled version.

6.2.1 power4 version

```
pifi rpnCompile(char *expr)
{
  insn *code= (insn *)calloc(64, sizeof(insn));
  int top= 3;
#ifdef _IBMR2
  code = code + 2;
#endif
  #[    .org     code   ]#;

  while (*expr)
    {
      char  buf[32];
      int n, tmp ;
      if (sscanf(expr, "%[0-9]%n", buf, &n))
      {
        expr+= n - 1;
        if (top == 10)
          {
            fprintf(stderr, "expression too complex");
            exit(0);
          }
        ++top; tmp = atoi(buf);
```

```
        #[
          subf  r(top), r(top), r(top)
          ori   r(top), r(top), (tmp)
        ]#;
      }
    else if (*expr == '+')
      { --top; #[ add  r(top), r(top), r(top+1)  ]# }
    else if (*expr == '-')
      { --top; #[ subf r(top), r(top), r(top+1)  ]# }
    else if (*expr == '*')
      { --top; #[ mullw r(top), r(top), r(top+1) ]# }
    else if (*expr == '/')
      { --top; #[ divw  r(top), r(top), r(top+1) ]# }
    else
      {
        fprintf(stderr, "cannot compile: %s\n", expr);
        abort();
      }
    ++expr;
  }
```

6.2.2 Itanium version

6.2.3 x86 version

```
    if (sscanf(expr, "%[0-9]%n", buf, &n))
      {
        expr+= n - 1;
        n = strtol(buf, 0, 0);
        #[
          push        %eax
          mov $(n), %eax
          ]#
       }
    else if (*expr == '+') #[
      pop      %ecx
      add      %ecx, %eax
    ]# else if (*expr == '-') #[
      mov      %eax, %ecx
      pop      %eax
      sub      %ecx, %eax
    ]# else if (*expr == '*') #[
      pop      %ecx
      imul     %ecx, %eax
    ]# else if (*expr == '/') #[
```

```
        mov     %eax, %ecx
        pop     %eax
        cltd
        idiv    %ecx
      ]# else {
        fprintf(stderr, "cannot compile: %s\n", expr);
        abort();
      }
      ++expr;
    }
  #[
        leave
        ret
  ]#;
```

6.3 Mandelbrot set

The Mandelbrot set is a mathematical set of point in the complex plane defined as the set of complex values of c for which the orbit of 0 under iteration of the complex quadratic polynomial $z_{n+1} = z_n^2 + c$ remains bounded. That is, a complex number, c, is in the Mandelbrot set if, when starting with $z_0 = 0$ and applying the iteration repeatedly, the absolute value of z_n never exceeds a certain number (that number depends on c) however large n gets. [2]

This mathematical set use complex arithmectic which is not supported by the C ansi.

6.4 power 4 implementation

A complex number is implemented with two successive floating point registers.

6.4.1 cell implementation

A complex number is implemeted with one register. The two first slots are the real and imaginary parts.

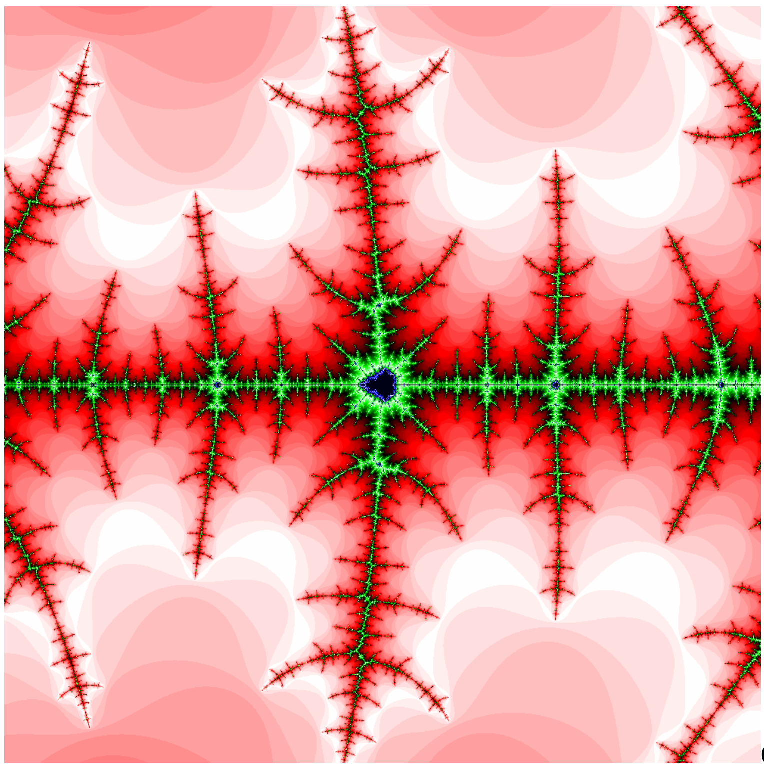
6.4.2 bluegene implementation

A complex number is implemented with one register (which contain 2 parts).

7 Porting HPBCG

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

[2] `http://en.wikipedia.org/wiki/Mandelbrot_set`



6

Figure 2: Meandelbrot set in a region centered on the point $(-1.41 + i * +0.00)$

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[3] 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

This parser translate the isa description into a code generation program.

For example the "multiply by a constant" instruction which is in the `hpbcg-power4.isa` file is defined as :

```
000111  r1_5  r2_5  i1_15-0  | mulli  r1,  r2,  i1
```

ISA	Power4	Altivec	CELL	IA64	x86
Integer register	32	32	128	128	
FP register	32	32	128	128	
Vector lenght	32	128	128	64	

Figure 3: Isa principal facts

is translated into theses two instructions bloc. The first one is a macro instruction which can be partially evaluated at static compile time, the second one is a function version which is usefull for debugging purpose.

```
#define mulli_iRRI(r1, r2, i1)      \
     ADDINSN((((LENOK(7,6))<< 5   \
      | LENOK(r1,5))<< 5          \
      | LENOK(r2,5))<< 16         \
      | LENOK(i1 & 65535, 16)))
```

```
void mulli_iRRI(r1, r2, i1)      \
{    ADDINSN((((LENOK(7,6))<< 5 \
     | LENOK(r1,5))<< 5         \
     | LENOK(r2,5))<< 16        \
     | LENOK(i1 & 65535, 16)));
#ifdef ASM_DEBUG
     printf("%p : %s%s 0x%X\n", asm_pc, "mulli_iRRI", *(asm_pc-1));
#endif /* ASM_DEBUG */
}
```

7.3 HPBCG Parser

8 Assembly languages

This part is devoted to different assembly languages syntax that **HPBCG** support.

The figure 3 try to summarize the supported data type.

8.1 cell-spu

The cell SPU has 128 bit wide registers.

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
- `$3` is the imput register

Return convention

8.2 power, cell-ppu and power FP2

Full programming description can be found in [5] or [1].

Integer register name one of

- `r0` .. `r31` integer registers name
- `f0` .. `f31` floating point register name
- `v0` .. `v31` vector register name (if altivec)
- `Cst(rx)` `Cst` indexed value for register `rx`

Calling convention
- `r3-r10` first interger parameter value
- `f1-f13` first floating point register value (simple or double precision)

Return convention
- `r3` return integer value
- `f1` return FP value

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.

8.4 x86

x86 is an exception for almost everything. It has stack based calling convention, specialized register names and variable lenght instruction encoding.

Register name

Calling convention

Return convention

9 Reporting bug

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

References

[1] Jonathan Bartlett. *Assembly language for Power Architecture*. IBM, http://www.ibm.com/developerworks/library/l-powasm1.html.

[2] Karine Brifault and Henri-Pierre Charles. Efficient data driven run-time code generation. In *In proceedings of Seventh Workshop on Languages , Compilers, and Run-time Support for Scalable Systems, Houston, Texas, USA*, Oct 2004.

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

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

[5] Ian Lance Taylor. *64-bit PowerPC ELF Application Binary Interface*. IBM, http://refspecs.linuxfoundation.org/ELF/ppc64/PPC-elf64abi-1.9.html.