POWERPC 755
INSTRUCTION-SET SIMULATOR
Reference manual 1.1.6

Authors: Daniel Gracia Pérez, Christine Rochange and Pascal Sainrat
# Table des matières

1 **Introduction**  

2 **Organization**  
   2.1 Files organization  
   2.2 Description of the file `ppc.nmp`  
   2.3 Description of `uisa_fp_instr.nmp` file  
   2.4 Description of the `vea_instr.nmp` file  
   2.5 Description of the `oea_instr.nmp` file  
   2.6 Description of external files  

3 **Installation and use of the iss**  

4 **Modifying and extending the simulator**  
   4.1 Configuration  
   4.2 Extending to microarchitectural simulation  
   4.3 Executing instead of emulating the Operating System  

5 **Test**  

6 **Performances**  

---

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Organization</td>
<td>4</td>
</tr>
<tr>
<td>Files organization</td>
<td>4</td>
</tr>
<tr>
<td>Description of the file <code>ppc.nmp</code></td>
<td>4</td>
</tr>
<tr>
<td>Description of <code>uisa_fp_instr.nmp</code> file</td>
<td>6</td>
</tr>
<tr>
<td>Description of the <code>vea_instr.nmp</code> file</td>
<td>7</td>
</tr>
<tr>
<td>Description of the <code>oea_instr.nmp</code> file</td>
<td>7</td>
</tr>
<tr>
<td>Description of external files</td>
<td>8</td>
</tr>
<tr>
<td>Installation and use of the iss</td>
<td>11</td>
</tr>
<tr>
<td>Modifying and extending the simulator</td>
<td>13</td>
</tr>
<tr>
<td>Configuration</td>
<td>13</td>
</tr>
<tr>
<td>Extending to microarchitectural simulation</td>
<td>14</td>
</tr>
<tr>
<td>Executing instead of emulating the Operating System</td>
<td>14</td>
</tr>
<tr>
<td>Test</td>
<td>15</td>
</tr>
<tr>
<td>Performances</td>
<td>16</td>
</tr>
</tbody>
</table>
Chapitre 1

Introduction

This document describes the nMP model of the Power PC 755 processor. It does not describe the instruction set itself since it is perfectly described in the manuals of IBM and Motorola [2], [3], [4]. Instead, it explains how the nMP description language has been used to describe this instruction set and realize an application level simulator.

With the help of the instruction-set simulator (iss) generator GLISS [1], this description generates a PowerPC iss which is capable of executing linux applications on a machine under linux Mandrake 9.0. It executes correctly several programs of the Specint 2000 suite.

The document starts with the explanation of the PPC755 Sim-nML description organization. Then, it is described how can the simulator be generated and used.

Finally, the document tries to explain how the simulator can be modified and extended.
Chapitre 2

Organization

The PPC755 description is organized in four .nmp files and some external .c files for things that can’t be described in NMP. The main file is ppc.nmp. It describes the registers and memory of the PPC755 and its main visa instructions. The other visa instructions are described in visa_fp_instr.nmp, these are floating point instructions and the management instructions of the floating point control registers.

The two other .nmp files define the VEA instruction set (vea_instr.nmp) and the OEA instruction set (oea_instr.nmp). This last file is not used by our final simulator, nevertheless it has been included to make easier the conversion of the simulator from an application level simulator\(^1\) to an operating system level simulator (in which system calls would really be executed by the simulator).

All the .nmp description files make full use of the capabilities of GLISS, for example defining macros or external functions calling. As such, it can be considered as a model for an ISA description. The additional fields user0 and category are not used in this simulator. They are just here as an example of additional field.

2.1 Files organization

The directory ppc contains the description files .nmp and three subdirectories:
- ppc/extern Contains external files.
- ppc/prog Contains some PPC755 programs to test the simulator that will be produced.
- ppc/simul contains the main file of the iss that will use the library created by the iss generator.
- ppc/debug Also contains a main file which is a debugger. See the debug manual for more information.

2.2 Description of the file ppc.nmp

This file is the core of the PPC755 processor description. It defines all the user level instructions of the PPC755 processor with the exception of the floating point arithmetic instructions and the floating point control instructions. This division has been made with a double purpose:

- Not all the implementations of the PPC755 have a floating point unit, so it’s seems logical to separate it from the rest of visa instruction set. This way a version without the floating point instructions of the simulator could be made just removing the include of the visa_fp_instr.nmp file.

\(^1\)In this simulator system calls are replaced by the system calls of the host machine
It is easier to work with the description of the simulator if we keep the floating point instructions separate from the rest of the isas instruction set. Both groups represent a big number of instructions, so it is easier to keep them separate to make the description files smaller and easier to work with.

Both things were accomplished thanks to the ability of the nmp file parser to work with multiple files using the include op "nmp file" statement.

All the definitions of data types, memory, registers and memory addressing modes that will be used by the description of the isas. But additionally it contains the registers that the oes instruction set (described in oes_instr.nmp) can use. The reason for that is that the initialization of the processor (see figure 2.1 on page 5) must be done in the main file of the description (i.e. in ppcm.nmp), and some registers of the oes instruction set have to be initialized since this simulator executes only applications and not an OS. So the registers initialized by the initialization operation (see figure 2.1) are described in the ppcm.nmp.

```c
op init ()
action = {
    MSR<MSR_FP..MSR_FP> = 1;
    MSR<MSR_PR..MSR_PR> = 0;
    MSR<MSR_FE1..MSR_FE1> = 0;
    MSR<MSR_FE0..MSR_FE0> = 0;
    MSR<MSR_LE..MSR_LE> = 0;
    TB = 0;
}
```

**Fig. 2.1** – Initialization code.

There is one instruction in the ppcm.nmp file that breaks the rule that this file just defines isas instructions. The instruction is sc (see figure 2.2 on page 5). This instruction was added to the isas instruction because it is necessary for the application level simulator and it is the only oes instruction that can be executed in user mode. This instruction calls an external function defined in an external file, because its operation is difficult to describe in the nmp language. This implementation also allows to change the operating system that the simulator runs easily, the user has just to redefine the external function to implement a different behavior.

```c
// System Call
op sys_call ()
syntax = "sc"
image = "0100010000000000000000000000000010"
action = {
    "sc_impl"("instr","state",M);
}
```

**Fig. 2.2** – Code nMP de l’instruction sc.

Two other special instructions appear in this file: the mf spr and mt spr instructions (see figure 2.3 on page 6). They are special instructions because they belong at the isas and oes instructions sets. Their opcode is exactly the same in both instruction sets. The decision was to put these instructions in both description files (ppcm.nmp and oes_instr.nmp), with the same syntax but with different opcode images. That can be done because these instructions do not access the same registers if they belong to user mode or supervisor mode, actually there is one bit defining the register to be used that is different if the instruction is executed in user mode.
or supervisor mode. The description takes advantage of this fact making the image definition of
the instructions different in one bit, this way the simulator can afterwards decide which one to
execute, the user mode instruction or the supervisor mode instruction\(^2\).

```
op mov_to_spr ( rs : index, spr : card ( 9 ) )
  syntax = format ( "mtspr \%d,\%d", spr, rs )
  image = format ( "011111%5b0%9b01110100110", rs, spr )
  action = {
    TMP_HWORD = spr;
    TMP_WORD = 0;
    TMP_WORD = TMP_HWORD < 4..0 >;
    TMP_WORD = TMP_WORD << 5;
    TMP_WORD1 = 0;
    TMP_WORD1 = TMP_HWORD<8..5>;
    TMP_WORD = TMP_WORD | ( TMP_WORD1 & 0x000000f);

    if TMP_WORD > 9 then
      "print" ( "mtspr is only user level instruction set
                   implementation." );
    endif;
    switch ( TMP_WORD ) {
      case 1 : XER = GPR [ rs ];
      case 8 : LR = GPR [ rs ];
      case 9 : CTR = GPR [ rs ];
      default : "print" ( "mtspr: rs error in user mode." );
    }
  }
```

**FIG. 2.3 – mtspr instruction code.**

### 2.3 Description of uisa_fp_instr.nmp file

The `uisa_fp_instr.nmp` file describes the PPC755 floating point instructions and the corresponding control instructions.

Le fichier `uisa_fp_instr.nmp` décrit les instructions flottantes du PowerPC et les instructions de contrôle correspondantes.

The definition of the instructions makes use of the tree operation structure that Sim-nML allows to make. That way the root operation checks that the floating point instruction can be executed, by looking at the FP bit of the MSR register, that defines if floating point instructions are supported. The FP bit is set to one in the initialization of the simulator and it is defined in the `init` operation of the `ppcm.nmp` file.

The instructions defined in this file use base functions and constants provided by GLISS in order to interface correctly with the floating-point model of the host machine. Examples are: `fpi_setround`, `fpi_testexception`, `FPI_TOWARDSZERO` (for rounding purposes), `FPI_INEXACT` (for testing exceptions) ....

\(^2\)The parser gives the user a warning if two instructions have the same syntax but different images, but it is just a warning that can be ignored in this case
The instruction-set simulator produced by GLISS uses the floating-point instructions of the host machine to emulate the floating-point instructions of the PowerPC. Thus, the non-IEEE mode of the PowerPC cannot be implemented in the simulator since it does not exist on host machines (except a PowerPC one of course). Furthermore, the way certain exceptions are detected in the PowerPC is special and cannot be reproduced as is on other machines like an x86 or a Sparc. Thus, the iss executes correctly floating-point instructions in the IEEE mode and when there are no exceptions. Exceptions will be detected as well on the host machine as on a PowerPC but the result might not be exactly the same. When an exception is raised, it is managed by an external function `launch_exception` in the file `exception.c`. Changing this function is possible. In particular, the instruction causing the exception could be reexecuted with an external mathematic library. Another way is to execute all floating-point instructions with such a library but it would significantly increase the execution time.

### 2.4 Description of the vea_instr.nmp file

The `vea_instr.nmp` file describes the PPC755 VEA instruction set. Most of the instructions defined in this file do nothing, they are just defined because their effects do not concern the logical view of the processor but the hardware one (for example, TLB management). For example the function that returns the value of the TB register of the PPC755 will return always 0 unless a cycle-level simulator added to this iss updates the value of the TB register.

There are no registers defined in this file; VEA registers are defined in `ppcm.nmp` in order to be initialized by the `init` operation (cf. section 2.2 on page 4).

### 2.5 Description of the oea_instr.nmp file

The `oea_instr.nmp` file describes the PPC755 OEA instruction set. As in the description of the VEA instruction set, some instructions are just defined but not implemented, because they manage some external components of the processor.

There are no registers defined in this file; they are defined in `ppcm.nmp` in order to be initialized by the `init` operation (cf. section 2.2 on page 4).

As the iss executes only application-level applications and these instructions are executed only under superuser level, they have never been tested. The root of the OEA instructions (so all OEA instructions) tests if the processor in superuser mode and raise a `PROGRAM` exception if it is not the case (see 2.4).

```c
// checking if the processor is in superuser mode,
op oea_instr ( x: oea_instr_action)
syntax = x.syntax
image = x.image
action = {
    if MSR<MSR_PR..MSR_PR> then
        "launch_exception"(PROGRAM);
    else
        x.action;
    endif;
}
```

**Fig. 2.4** – If an OEA instruction is executed in user mode, an exception is raised.
As mentioned in section 2.2 page 4, two instructions appear in both descriptions files ppc.nmp and oea_instr.nmp. These instructions are mtspr and mfspr. If you check figures 2.5 and 2.3 page 6, you will see that both look the same except 1 bit in the image and the action which is different. In the case of op oea_mtspr the image is "...%b1%b..." and in op mtspr it is "...%b0%b...". This small difference allows us to define the instructions mtspr and mfspr in both files.

```c
op oea_mtspr ( spr : card( 9 ) , rs : index )
syntax = format ( "mtspr %d,%d", spr , rs )
ingame = format ( "0111115b1%9b0110100110", rs , spr )
action = {
    TMP_HWORD = spr;
    TMP_WORD = 0;
    TMP_WORD = TMP_HWORD < 4.0 >;
    TMP_WORD = TMP_WORD << 5;
    TMP_WORD = (TMP_WORD |
        (TMP_HWORD < 8.5 & 0x000000f )) |
    0x00000010;

    switch(TMP_WORD) {
        case 1:    XER   = GPR[rs] ;
        case 8:    LR    = GPR[rs] ;
        case 9:    CTR   = GPR[rs] ;
        case 18:   DSISR  = GPR[rs] ;
        case 19:   DAR   = GPR[rs] ;
        ...
        case 1019: ICTC  = GPR[rs] ; // mpc750 & mpc755 ONLY
        case 1020: THR[0] = GPR[rs] ; // mpc750 & mpc755 ONLY
        default : "print" ( "mtspr: rs error in superuser mode." );
    };
}
```

Fig. 2.5 – Instruction mtspr description (OEA version).

### 2.6 Description of external files

The external files (present in the extern directory) provide functions that are needed to initialize the iss. They also provide some functionalities that are easier to describe using the c language than using nMP.

In this iss are provided modules essential to the execution of the iss like an elf program loader in modules load and all files starting by elf, modules providing functions used in the nMP description like print which is described in the module random or launch_exception described in the module exception, and finally functions specific to the PowerPC which are described in the files starting by mmu or bat.
More external facilities can be added easily. All the external files can access to the registers and memory defined by the nMP description files, using the macros provided by GLISS.

**Description of exception.c**

This file shows how an exception manager can be written. This manager is very simple, it just print the state of the iss using function iss_dump provided by GLISS in issExtern.h and finishes the execution.

**Description of system.c**

This file provides two different functionalities:

1. a function which loads an ELF PowerPC object file (the program to simulate),
2. a LinuxOS system call emulation (provided by sys_call.c).

The functions provided by this file must be initialized with the PPC755 file to execute, its argv and argc arguments, and three optionals ones:

1. the memory page size of the system to emulate (1024 by default),
2. a debugging option that can be activated or desactivated (desactivated by default),
3. an additional system handler that the system.c will access to modify its state when doing OS functions. That handler is useful to link the library with an external simulator, that way our simulator and the external one will obtain the same results from the OS system emulator that system.c provides. The external system should interface with the library through the system_handler_t structure (see figure referefrig :emulstatendcode on page pagererfrfig :emulstatecode) that can be found in system.h. The instance variable in system_handler_t structure will be the first parameter passed to functions defined in the structure.

The emulation of system calls by the system calls of the host machine (Linux) is made in function sc_impl provided in sys_call.c.

**Description of the mmu module**

Two kind of Memory Management Modules are currently modelled.

mmu_bat_notrans uses the BAT registers in order to check access authorizations but does not translate memory addresses.

In addition to access rights checking, mmu_bat_trans translates addresses according to the content of the BAT registers. These registers must be correctly initialized either in the nMP init op or in the initialization function of the module or by the loader as this initialization might depend on the programs to execute.
typedef struct {
    /* Registers interface */
    void (*write_gpr)(void *, int, int32_t);
    int32_t (*read_gpr)(void *, int);
    void (*reset_cr0so)(void *);
    void (*set_cr0so)(void *);

    // Memory interface
    uint8_t (*mem_read_byte)(void *, address_t);
    void (*mem_write_byte)(void *, address_t, uint8_t);
    uint16_t (*mem_read_half_word)(void *, address_t);
    void (*mem_write_half_word)(void *, address_t, uint16_t);
    uint32_t (*mem_read_word)(void *, address_t);
    void (*mem_write_word)(void *, address_t, uint32_t);
    uint64_t (*mem_read_dword)(void *, address_t);
    void (*mem_write_dword)(void *, address_t, uint64_t);
    void (*mem_set)(void *, address_t, uint8_t, int);
    void (*mem_read)(void *, void *, address_t, int);
    void (*mem_write)(void *, address_t, void *, int);

    int memory_page_size;

    void *instance;
} system_handler_t;

**Fig. 2.6** – the system_handler_t structure.
Chapitre 3

Installation and use of the iss

Sources of the iss are provided as a compressed archive. Extracting the files is done in the following way:
- under Linux: `tar -zxvf ppc-v1.0.tgz`
- under Solaris: `gunzip -stdout ppc-v1.0.tgz | tar xvff -`

The directory `ppc` is created and contains all extracted files.

To build the emulator, you must change the Makefile in the `ppc` directory in order to change the path to GLISS (default is at the same level as the `ppc` directory). If you want to use one of the mmus, read the Makefile and modify it according to what is written in the Makefile.

Then, you just have to type `make all` in order to build the emulator. You may have some warnings, don’t worry about them.

Once the Makefile has finished, files are organized as follows:
- `/ppc` The root directory of the iss where `.nmp` files are, the main `Makefile`, ...
- `/ppc/extern` Contains all external files.
- `/ppc/prog` Contains some programs compiled for the PowerPC and used to test the iss.
- `/ppc/simul` Contains the main file of the iss which calls the functions of the library generated by GLISS. It might be replaced by a microarchitectural simulator.
- `/ppc/src` Contains the `.c` and `.h` files generated by GLISS.
- `/ppc/lib` Contains the library created with the source files above.
- `/ppc/include` Contains the `.h` files corresponding to the library.

GLISS creates a library which can be used by a simulator (in our case, the simplest simulator which realizes an iss). This library is in the directory `ppc/lib`. The library contains the definition of all instructions defined in the nMP description and the functions to decode them, see their results and store them in the state of the simulator. The simulator itself can also read or write this internal state. The library also contains the functions written in the external files.

The simulator (`simul.c`) is very simple. Except calling the main functions provided by GLISS, it calls the function to initialize the state of the simulator which among other things, initializes the analyser and the different options. It is in charge of loading the program to execute in the memory of the simulator. Then, in a loop, it uses the fetch, decode, execute and complete functions (see figure 3.1 page suivante).

To finish to generate the simulator, this file (or another one of your own) must be compiled using `make` in the directory `ppc/simul`.

The executable `simul` file is created in `ppc/simul`. Its arguments are given in figure 3.2 page suivante.
real_state=iss_init(...); // simulator initialization
while (running){
    iss_fetch(NIA(real_state),buff_instr);
    i=iss_decode(NIA(real_state),buff_instr);
    iss_complete(i);
    iss_free(i);
    instr_count++; // instructions PPC755 executed
}

Fig. 3.1 – the body of simul.c.

./simul [<simul_options>] -- <nom_prog> [<options_prog>]
simul_options:
    -i <num>    maximum number of instructions
    -v[v_file>] verbose system calls in stderr
                or v_file if defined
    -m <num>    sets the memory page size (4096
                if not defined)
    -d [debug_file] debugging information is written
                in debug_file
    -s <num>    first instruction to debug (only
                if -d is set)
    -n <num>    number of instructions to debug
                (only if -s is set)

Fig. 3.2 – simul arguments
Chapitre 4

Modifying and extending the simulator

4.1 Configuration

It is very easy to create a new PowerPC processor without floating point units or initialize differently the registers just by modifying the init op in ppc.mmp.

It is also easy to specify if you want to use a mmu (currently segmentation is not implemented only BAT arrays). You just have to modify the Makefile adding some parameters in the external files constant either

-e mmu_bat_notrans+M -E bat -t code mmu_bat_notrans_code -t data mmu_bat_notrans_data

or

-e mmu_bat_trans+M -E bat -t code mmu_bat_trans_code -t data mmu_bat_trans_data

depending on the mmu you want. Note that as you add a module with initialization (option '-e' and the initialization functions of these mmus do not need any parameter, you just must add a NULL pointer in the arguments of iss_init in simul.c.

You can disable the decode cache whose aim is accelerating the simulation. You should add the line:

OPT = -DEMUL_NO_CACHE_DECODE

It’s only needed to disable it when the iss executes self-modifying code.

Finally, you can add the following line

OPT = -DEMUL_DISASM

This provides the function iss_disasm which disassembles an instruction. Its use is shown in the debug program.

In these two last cases, you can also specify them on the make command line.

Exemple

make all OPT=-DEMUL_DISASM

You probably notice in the nMP descriptions two additional fields: category and type, these are used in applications which are not provided. To use such additional fields or add some of your own see the GLISS manual. You just have to add parameters to specify the additional fields you want to use in the iss using the -a option.
4.2 Extending to microarchitectural simulation

There are two possibilities to take into account performance and not only functionnality. Both consist in replacing the file `simul.c` by a simulator which is interfaced to our library.

In our `simul` program, only one state is initialized. It is the state of the instruction-set simulator.

First you might want to privilidge fiability against speed of simulation. In this case, the iss might be connected to a simulator which makes also decode, execution and so on but on its own logical state. This way, the iss is a reference and the simulator can check both states to verify that it is functionnal correct. The simulator can use the functions of the library or its own functions if they already exist. It is preferable that system calls are executed by the same library in order to be sure they are identical. The simulator can use the system call manager of our library (see section 2.6 page 9).

Second, you want a simulator as fast as possible. The iss is connected to a micro-architectural simulator which deals only with temporal and purely hardware aspects. Then the micro-architectural simulator calls systematically the functions of the library except for the instructions which have only an impact on the hardware and not on the logical state like `eieio`. It is the responsibility of the simulator to take into account the execution of the instructions. Their action in the nMP file is most of the time empty. Eventually, you can extend the resources of the iss to purely hardware resources. The simulator will be able to modify these resources using the access macros which will be generated by GLISS. However, the interest is low. The principle of GLISS is to have only resources which can be viewed by the programmer.

To surpass the limitations on floating-point exceptions, floating-point operations should be executed by external functions. These functions should be defined in order to reflect exactly the floating-point model of the PowerPC. Some floating-point libraries are available on the web (for example, one is provided in gcc) but we don’t know if it can model exactly the PowerPC. We didn’t investigate this because the speed of the iss will be much lower and our applications don’t have such exceptions.

4.3 Executing instead of emulating the Operating System

Our iss, as it is currently implemented, is intended to execute only linux applications and does not support the emulation of all system calls. Of course, it can be extended to support more system calls or to emulate system calls on other OS (see `sys_call.c`).

One could also need to execute the OS itself, especially in embedded systems where the OS is quite light and applications running on this OS. In this case, system calls are not emulated by the system calls of the host machine. The external function `sc_impl` or the nMP description of `sc` should be modified to have the correct behaviour. Furthermore, exceptions are not dealt correctly for such a complete system simulation. The function `launch_exception` should be rewritten. Finally, note that you can write the `init` description in `ppc.nmp` so that it initializes all the logical state as if the bootstrap was executed. This significantly reduce simulation time.
Chapitre 5

Test

This instruction-set simulator has been tested using 5 programs of the Spec2000 suite in the test mode. These five programs are *eon*, *parser*, *crafty*, *gzip*, *twolf*. It has also been tested with the programs *argv* and *primes* provided in the *ppc/prog* directory.

All have been executed up to the end and they produced the good results. They have been compiled with *gcc* :

```
powerpc-linux-gcc -static -O3 -fomit-frame-pointer -o <nom_executable> <nom_fichier_source>
```

In the directory *ppc/prog*, quelques programmes de test sont fournis. Few test programs are provided :

- **primes** : prime numbers calculus
- **argv** : program which shows its environment (you can give any argument)
- **crafty/crafty-base.ppc32-linux** : a chess program.

**crafty** must receive on its standard input ("<") one of the entries provided in *ppc/prog/crafty/data* for example *ppc/prog/crafty/data/test/input/crafty.in*. 
Chapitre 6

Performances

The simulation performance highly depends on the simulator configuration, especially parameters which configure the memory of the simulator (parameters of the function `iss_init`). The default is a memory with 4 modules of 1 Mbytes and we use this option in our tests. This configuration is interesting for short programs using few data. In the case of `parser`, the memory has been configured as 8 modules of 4 megabytes because simulation was very long due to a high number of swaps. Thus, the host machine should have enough memory. In our case, we have 512 MB of RAM on our machines.

The following table gives some performance results on the programs cited above with the test inputs. The decode cache was activated et no mmu was used. Tests have been done on an Athlon TBird at 1,33GHz with 512 méga-octetsmegabytes of DDR-DRAM and a Gigabyte GA-7DX motherboard. The OS was Linux Mandrake 9.0. Numbers given in this table are means of several executions because the simulation vary according to the state of the host machine.

<table>
<thead>
<tr>
<th>Prog.</th>
<th>nombre d'instructions</th>
<th>instr/sec</th>
<th>temps cpu (ms.)</th>
<th>temps système (ms.)</th>
<th>temps total (ms.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>crafty</td>
<td>9 172 715 255</td>
<td>2,72</td>
<td>2 582 090</td>
<td>410</td>
<td>2 582 500</td>
</tr>
<tr>
<td>gzip</td>
<td>5 268 434 111</td>
<td>4,70</td>
<td>1 121 840</td>
<td>250</td>
<td>1 122 090</td>
</tr>
<tr>
<td>bzip2</td>
<td>24 675 238 449</td>
<td>4,60</td>
<td>14 200</td>
<td>1 125 960</td>
<td></td>
</tr>
<tr>
<td>parser</td>
<td>7 541 774 619</td>
<td>3,32</td>
<td>2 268 910</td>
<td>2 590</td>
<td>2 271 500</td>
</tr>
<tr>
<td>mcf</td>
<td>382 615 159</td>
<td>1.01</td>
<td>376 890</td>
<td>540</td>
<td>376 890</td>
</tr>
</tbody>
</table>
Bibliographie


