Typologie des "temps" de compilation

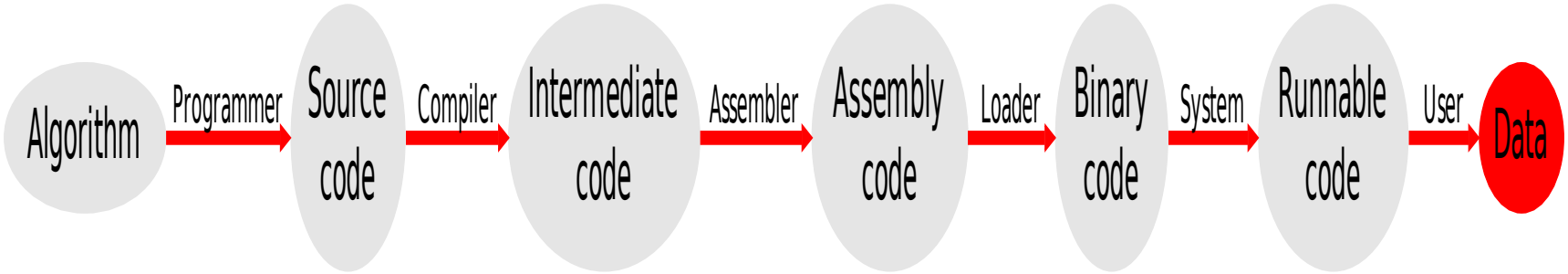
Séminaire MOAIS Grenoble

Henri-Pierre Charles

Université de Versailles Saint-Quentin en Yvelines

7 mai 2009

Scientific computation



Algorithm is supposed to be target independent

Source code is supposed to be “high level”

Runnable code is supposed to be “low level”

Optimization is supposed to be “data independant”

The target machine is supposed to “be known”

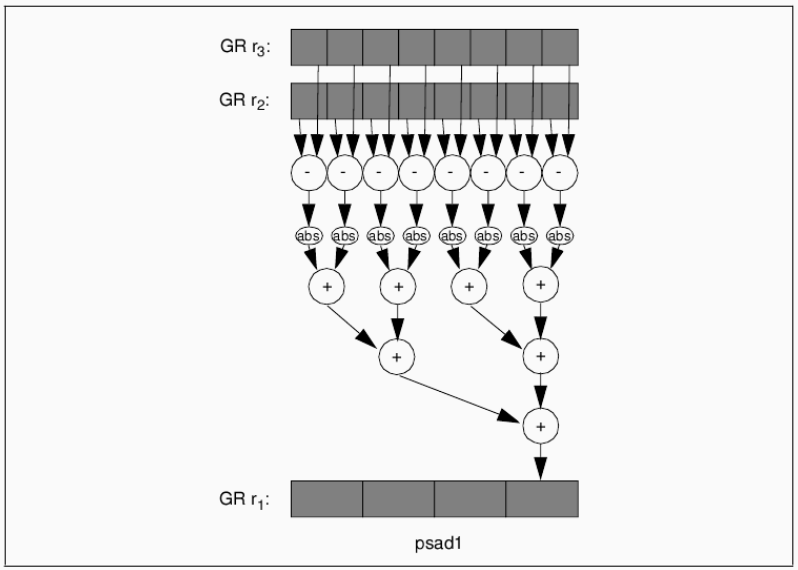
“Low level instruction” (Itanium ISA)

psad — Parallel Sum of Absolute Difference

Format: (*qp*) psad1 $r_1 = r_2, r_3$ I2

Description: The unsigned 8-bit elements of GR r_2 are subtracted from the unsigned 8-bit elements of GR r_3. The absolute value of each difference is accumulated across the elements and placed in GR r_1.

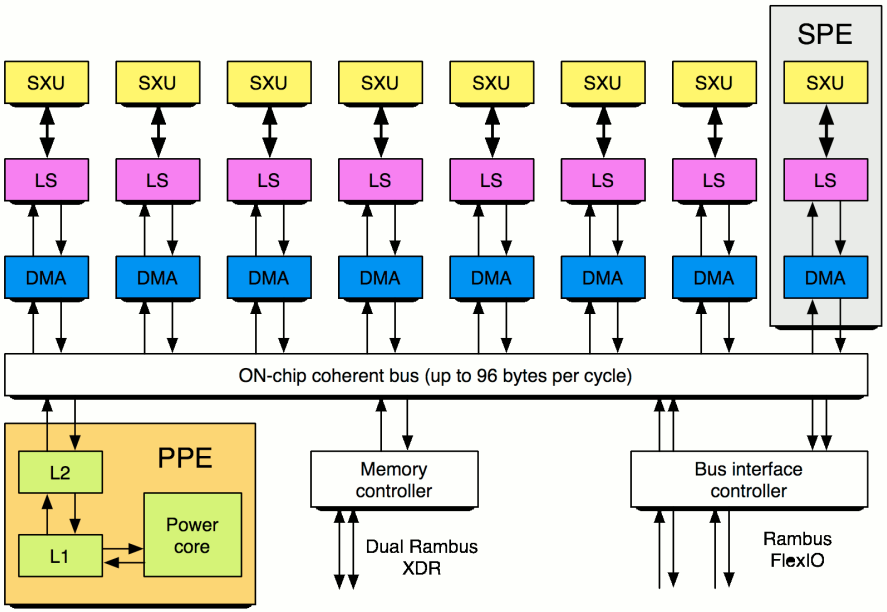
Figure 2-38. Parallel Sum of Absolute Difference Example



Instruction PSAD : Parallel sum of absolute differences

Compilers designer are always in late in race with processors architect.

Cell ISA

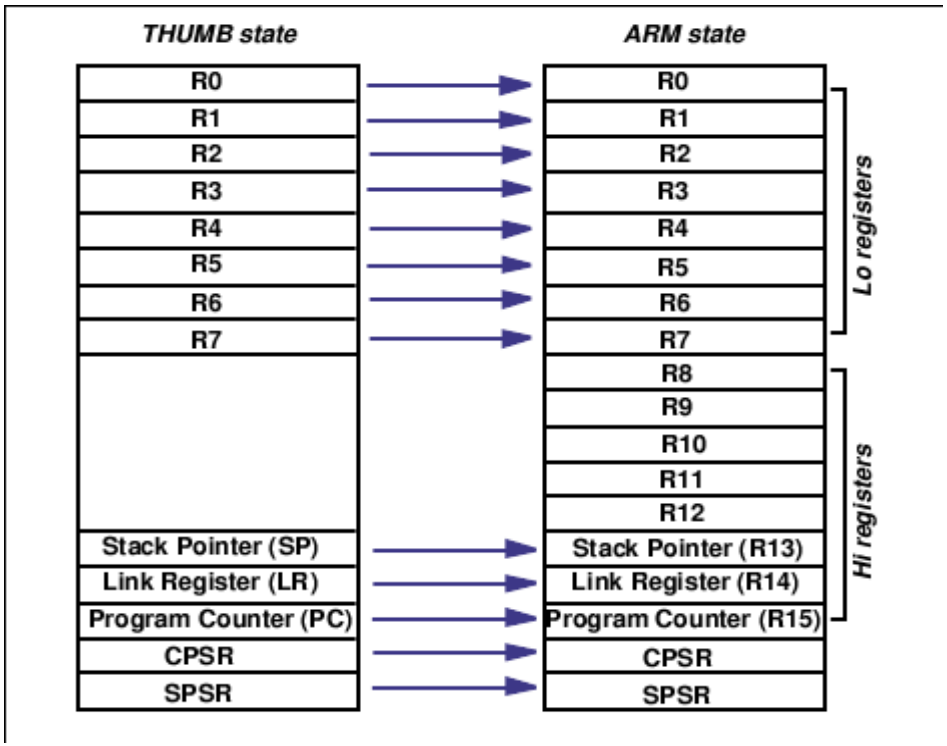


Multiple ISA :

- Power4 32/64 bits data manipulation
- Altivec 128 bits, multimedia instruction set
- VMX 128 bits, vector instruction set

http://upload.wikimedia.org/wikipedia/commons/8/82/Schema_Cell.png

ARM



Ref : Arm thumb documentation

Multiple ISA :

- Jazelle 80 % Java ISA
- Thumb 16 bits ISA
- ARM32 32 bits ISA
- Neon 32 bits multimedia ISA

“Low level instruction” (Power ISA)

- PPC440 FP2 Architecture
- Blue Gene L architecture
  - 32 Pair of FP registers (Seen as Complex or vector)
  - Special instructions like : `fxcxnpma t, a, b, c`
    $t_p = (A_s * C_S - B_P)$, $t_s = A_S * C_P + B_S$
- **BUT** : C has no standard support for complex number (intrinsic or libraries are not portable)

Blue gene IBM xlc compiler documentation

"Exploiting the Dual Floating Point Units in Blue Gene/L"

Future directions

IBM plans to continue to address performance of dual FPU code in future updates and releases. Improvements in the SIMD framework will also benefit BlueGene. We expect that this will lead to better exploitation of the dual FPU.

Summary

The presence of a second FPU on the BG/L processors potentially allows double the performance on floating point algorithms over just using a single FPU. The ability of the IBM XL compilers to automatically use the dual FPU unit depends strongly on the properties of the source code. The more regular the accesses to floating point data, the more the compiler is able to exploit the dual FPU. Examples of regular access include *matrix multiplication* and *vector processing*. This paper has described the implementation of the dual FPU in BlueGene/L and some limitations of automatic compiler exploitation of this dual FPU. Our long term goal is to ensure that using the dual FPU will be no slower than single FPU code. This may not be achievable, due to the extra versioning necessary for alignment or aliasing checks, but the overhead should be minimized.

Source code is not high level

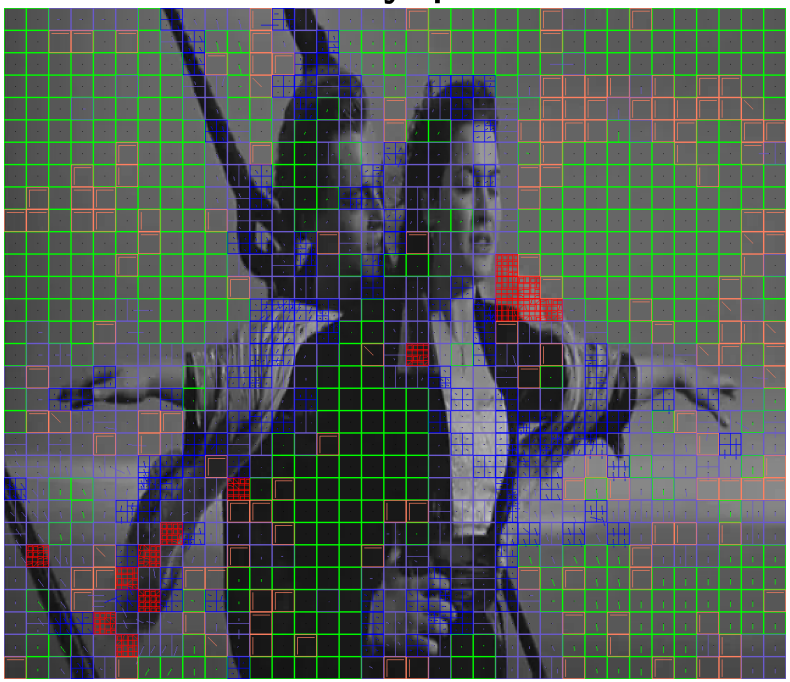
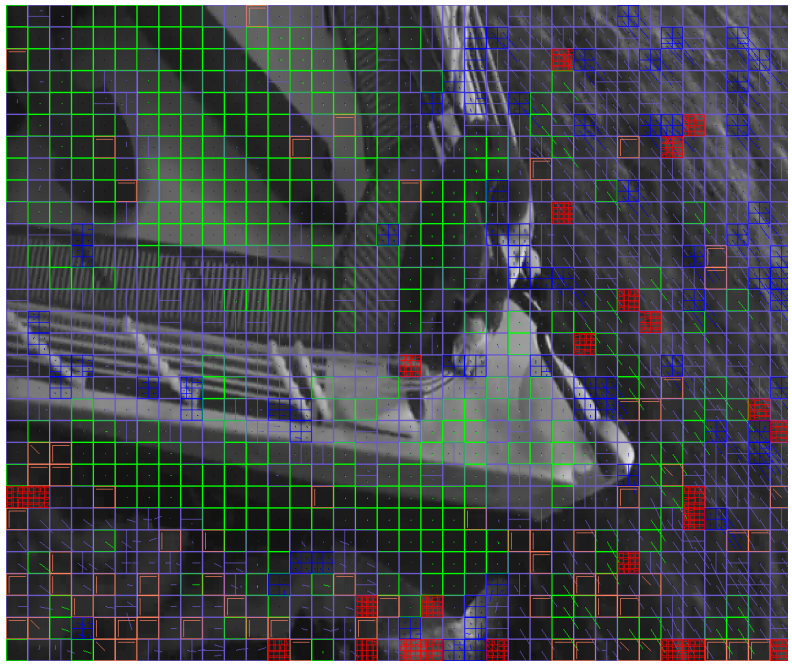
ARM teach how to write code with architecture in mind

How does the programmer's model affect C?

- Choose the right algorithm
- Make best use of the instruction set
- Make good use of available memory types
- Lay out data structures carefully
- Write C with the underlying architecture in mind

The “Titanic” effect 1/2

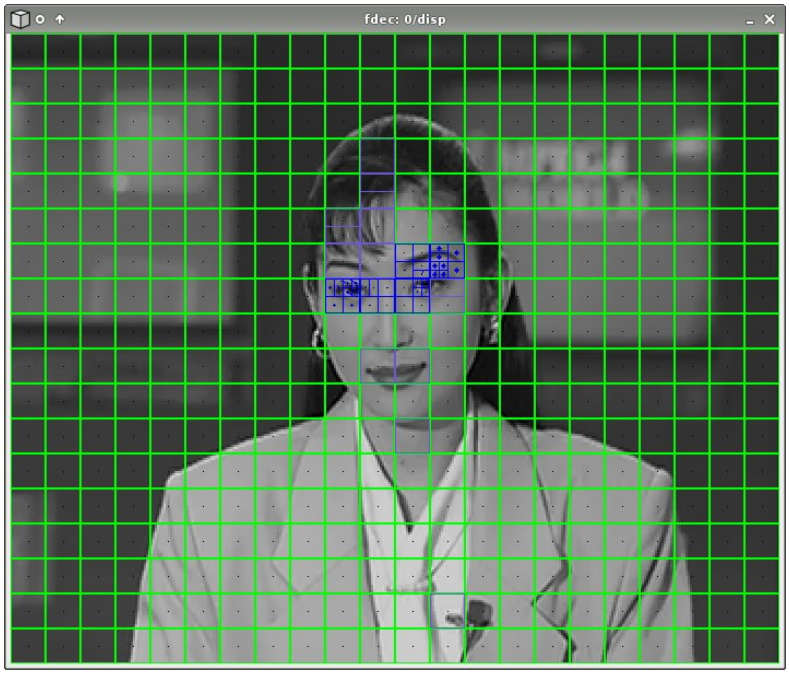
Data set modify performance application during run-time

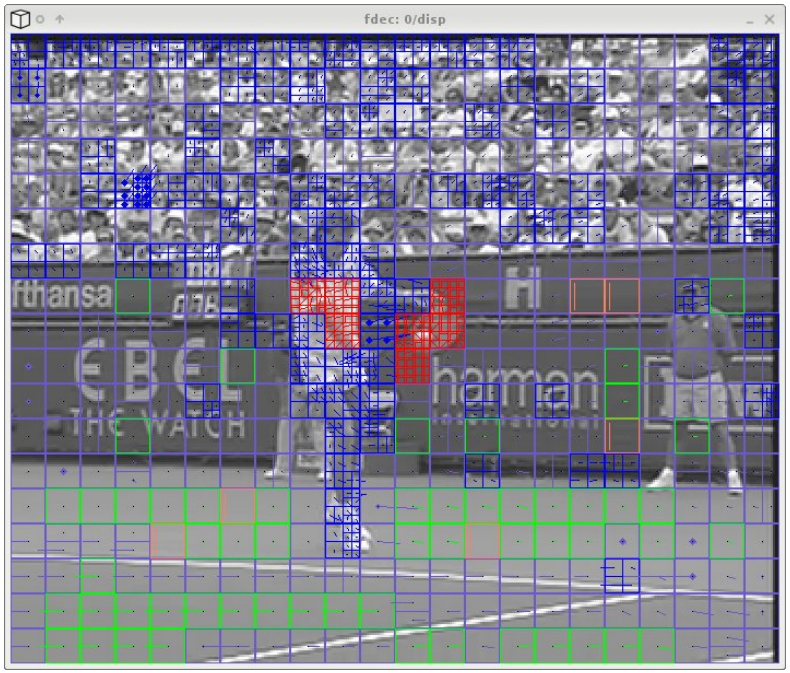



Using a FreeBOX (ADSL Modem and TV/IP broadcasting), my home computer (Intel Celeron 1.70GHz) play correctly France3 channel (which use MPEG TS video format) but cannot play RTL9 (X264 video format)

The “Titanic” effect 2/2

Data set modify performance application during run-time





Extract from MPEG group benchmarks

Optimization is not data independant

Multimedia arithmetics

Vector operand from 1 to 4

Specific arithmetic Integer, Floating point, Saturated, 8/16/32/64/128 bits

Specific operators SAD, Byte counts, interpolator, etc

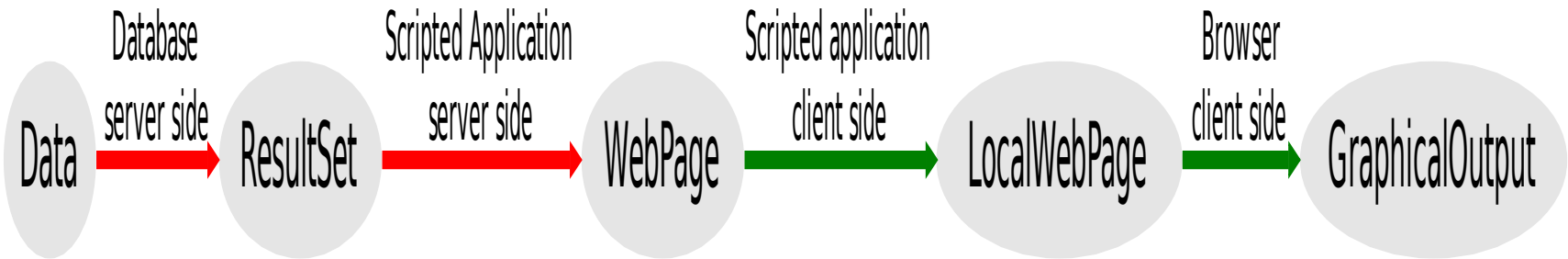
openoffice.org file extensions

# of file	File Ext	File type
1603	ott	Template Text Openoffice
2243	xhp	Help Text
2480	mk	Part of Makefile
3639	java	Java sourcecode
3958	idl	Interface Definition language
10015	cxx	C++ source code
10392	hxx	C++ header
11706	png	Fichiers d'image

Openoffice.org version 2.4.1 contain 62646 files. 1.7 Gb.

33 open-source libraries (Scanner access, images, Beanshell, etc)

Web page devellopment



Server side Interperter (php, python, perl), Compiler (C, Java)

Client side Interperter (javascript + JIT), Interpreter (XULrunner + javascript)

Server side is able to specialize the code for the final client ! (Google specialize the javascript code depending on the client browser)

What time is it ?

Install time Program installation

Thinking time Programmer activity

Static compile time Translate program to binary code

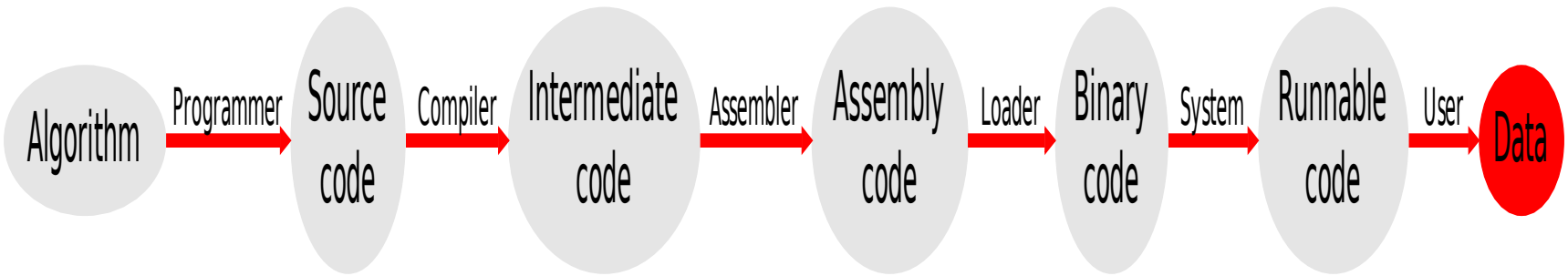
Start time Target processor and context are known

Run time Data values and alignment are known

Run-time model

Proc. Familly	Domain	
Small speed processor	Embeded system	Correct ISA to choose
High Perf processor	HPC	Memory access / Specificarithmetic
Interpretor	Glue programming	Speed of code generation
GPU	Graphic / HPC ?	ILP extraction / Arithmetic
Virtual processor	Portability	Target specialization

Standard compiler chain



- Optimization decision too early, even with profiling
- Final target is't known

Virtualisation

Virtualize a system (client) as a software on a host
Need to switch contexts from differents systems
A virtualizer is a virtual machine able to

Native mode Run binary code on the same target when possible

Virtual mode Rewrite or interpret binary code

Virtualisation : QEMU

QEMU is used by Sun virtualization system (Busybox) in and development platform for Android system

Native mode Run binary code on the same target when possible

Virtual mode Each target processor instruction is defined by a function.
An emulated code is translated into target code by concatenation theses “instructions”

No possible run-time optimization, but optimized sequence of target code.

JIT

Compile from high level to low level (bytecode)
Compile at run time from low level to binary code

- **HotSpot** Java JIT
- **LLVM** Compiler infrastructure
- **Mono** virtual machine C#

Compile at correct time but lack of essential information : used arithmetic, data allocation, etc
Intermediate representation kill any possible optimization.
Bytecode is compact but disallow run-time optimization.

JIT : HotSpot example

Sun JIT run-time compiler

- Use profile (Client, Server)
- Detect at run-time the hotspot
- Apply increasingly costly optimization
- Unable to use specific instructions

Good code opmization “asymptotically”
Need to reconstruct program structure at run-time
Based on bytecode ...

JIT : LLVM example

Compiler infrastructure

- Keep program structure until run-time
- Code generation at start-time
- High level intermediate representation

Slow code generation

Classical optimization based on code structuration

Interpretor with JIT

Tamarin `http://www.mozilla.org/projects/tamarin/` JIT javascript en C++

Rhino JS `http://www.mozilla.org/rhino/` JIT javascript en JAVA

Parrot `http://www.parrot.org/` Machine virtuelle à registre

Interpretors does not have global view of program, by definition.
Very fast (bad) code generation, but faster than line per line interpretation
Very fast starting time
Most deployed code generators !

GPU

Highly parallel architecture (ILP) (CG, OpenCL, OpenGL, ...)
Multiple isa (host, target GPU)

Static time Compilation

- **Host** from high level to binary
- **GPU** from high level to assembler (text)

Starting time GPU from assembler to binary GPU

Starting time : too early to take essential information into account
Very specific programming model (massively parallel)

Exemple of time distance

Phase	GCC	Java
Algo		
Programm	C ansi	Java standard
IR	GIMPL/RTL	
Assembly	Target specific	Byte code
Binary		Target specific
Runnable	Loaded code	

Run time Static compile time

Classification Attempt

Time / Tool	IR	Install	Thinking	Static	Starting	Running
gcc	Gimple / TREE /RTL	✔		✔		
tamarin						✔
GPU/OpenCL	3 Addr Vect. Assembly code			✔	✔	
Qemu	Bin. Native code	✔				✔

Let's start from the low level

BECAUSE

- this is the main constraint (which come from industry)
- they are many optimization opportunities
- it give the programming model (which is the main constraint for compiler)
- "The advent of just-in-time compilation for languages (such as Java) blurs the distinction between compile time and runtime, opening up new opportunities for program optimization based on dynamically computed program values. As parallel client-side applications emerge, runtime dependence checking and optimization are likely to be essential for optimizing programs that manipulate dynamic data structures" (**Compiler Research : the next 50 Years** ; Février 2009 ; Mary Hall, David Padua and Keshav Pingali ; Communication of the ACM

What do I need

Objective

- Mix “constant” data & binary program
- Full ISA description (multimedia instructions, baroque arithmetics, vector instructions)
- Portable code generation (mix multiple ISA)

Tool Needed

- FAST architecture description
- FAST code generation (9 machine cycle / instruction)

How to write a dynamic code generator ?

```
unsigned char code[] =
  {
   0x78, 0x81, 0x01, 0x83,      /* mpy $3,$3,$4 */
   0x35, 0x00, 0x00, 0x00       /* bi  $0 */
  };
  ../..
  typedef int (*pfiii)(int, int);
  pfiii Sad = (pfiii) code;
  int result = Sad(2, 21);
```

Documentation example

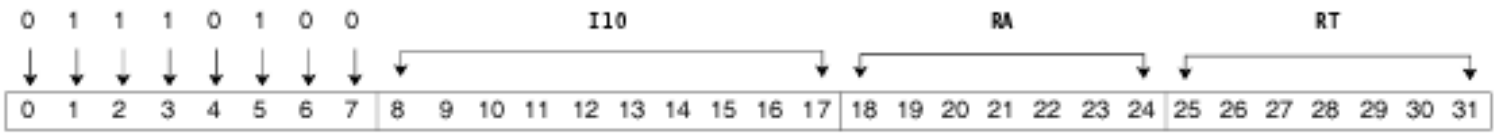
IBM

Instruction Set Architecture

Synergistic Processor Unit

Multiply Immediate

mpyi **rt, ra, value**



For each of four word slots:

- The signed value in the I10 field is multiplied by the value in the rightmost 16 bits of register RA.
- The resulting product is placed in register RT.

What did I do ? (Thinking time)

- Describe the architecture. Example cell-spu :

```
cell 32
../..
# Integer and Logical  Instructions
 01111000100      r3_7    r2_7  r1_7       | mpy    r1, r2, r3
 01111001100      r3_7    r2_7  r1_7       | mpyu   r1, r2, r3
 01110100         i1_9-0  r2_7  r1_7       | mpyi   r1, r2, i1
 01110101         i1_9-0  r2_7  r1_7       | mpyui  r1, r2, i1
 1100           r1_7   r3_7  r2_7  r4_7 | mpya   r1, r2, r3, r4
../..
```

Easy to do from the processor documentation

What did I do ? (Install time)

- Parse the ISA description and generate
  - the macro instruction generator :

```
#define mpyi_iRRI(r1,r2,i1)
    ADDINSN((((( LENOK(116, 8) )<< 10
        |  LENOK((i1 & 0x3ff), 10) )<< 7
        |  LENOK(r2, 7) )<< 7
        |  LENOK(r1, 7) ))
```

  - the function instruction generator :

```
void mpyi_iRRI    (int r1,int r2,int i1){
    ADDINSN((((( LENOK(116, 8) )<< 10
        |  LENOK((i1 & 0x3ff), 10) )<< 7
        |  LENOK(r2, 7) )<< 7
        |  LENOK(r1, 7) ));
#ifdef ASM_DEBUG
printf("%p : %s%s 0x%X\n", asm_pc, "mpyi", "_iRRI", *(asm_pc-1))
#endif /* ASM_DEBUG */
}
```

Write a “compilette” (thinking time)

```
pifi multiplyCompile(int multiplyValue)
{
 insn *code= (insn *)_malloc_align(1024, 7);
 (void) printf("Code generation for multiply value %d\n", mu
  #[
    .org    code
    mpyi    $3, $3, (multiplyValue)
    bi $lr
  ]#;
  (void) printf("Code generated\n");
  return (pifi)code;
}
```

Use the "compilette"

```
/* Generate binary code */
multiplyFunc = multiplyCompile(atoi(argv[1]));
for (i = 1; i < 11; ++i)        printf("%3d ", i);
printf("\n");
for (i = 1; i < 11; ++i)      printf("%3d ", multiplyFunc(i
printf("\n");
```

Generate the code generator (Static compile time)

```
hpbcg simple-multiply-cell.hg > simple-multiply-cell.c
cc -Wall ../.. simple-multiply-cell.c -o simple-multiply-cel
spu-gcc  ../.. -o simple-worker-cell simple-worker-cell.c
```

Run the code generator (Run time)

```
turner:simple-multiply/>./simple-multiply-cell 6
Code generation for multiply value 6
Code generated
  1   2   3   4   5   6   7   8   9  10
  6  12  18  24  30  36  42  48  54  60

turner:simple-multiply/>./simple-multiply-cell 7
Code generation for multiply value 7
Code generated
  1   2   3   4   5   6   7   8   9  10
  7  14  21  28  35  42  49  56  63  70

turner:simple-multiply/>./simple-multiply-cell 14
Code generation for multiply value 14
```

Ask for the program

- `http://hpcbg.org/`
- Support for :

power	power4, altivec, cell, FP2
itanium	full isa but not scheduling
arm	preliminary

- Samples codes
- BSD like licence

Code specialization

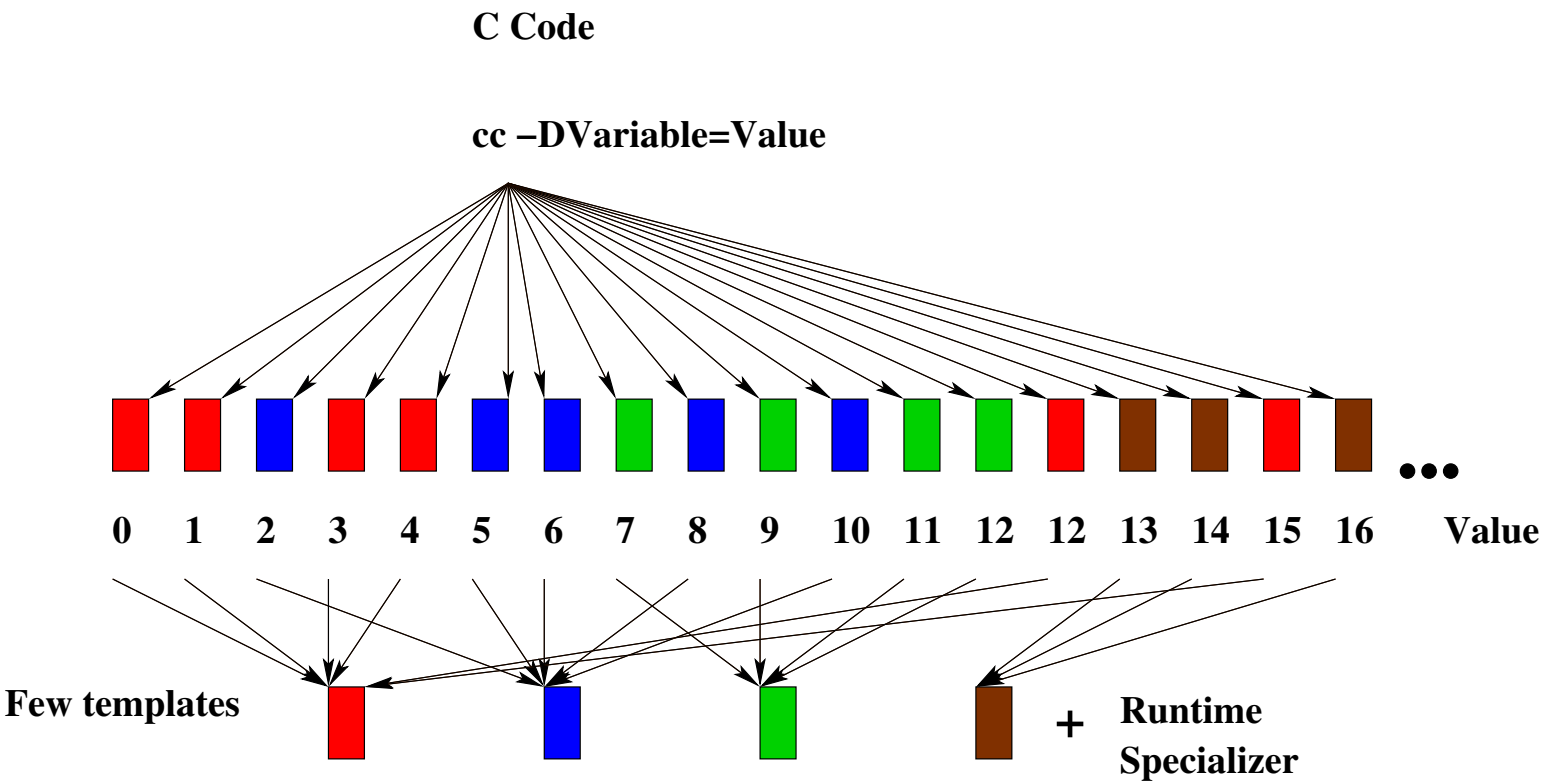
Let's remove one function parameter :

```
void Function(float *A, float *B, int size /*, int stride*/)
{     int i;
    for (i = 0; i < size; i++)
        A[i*stride] = B[i] + A[i];;
}
```

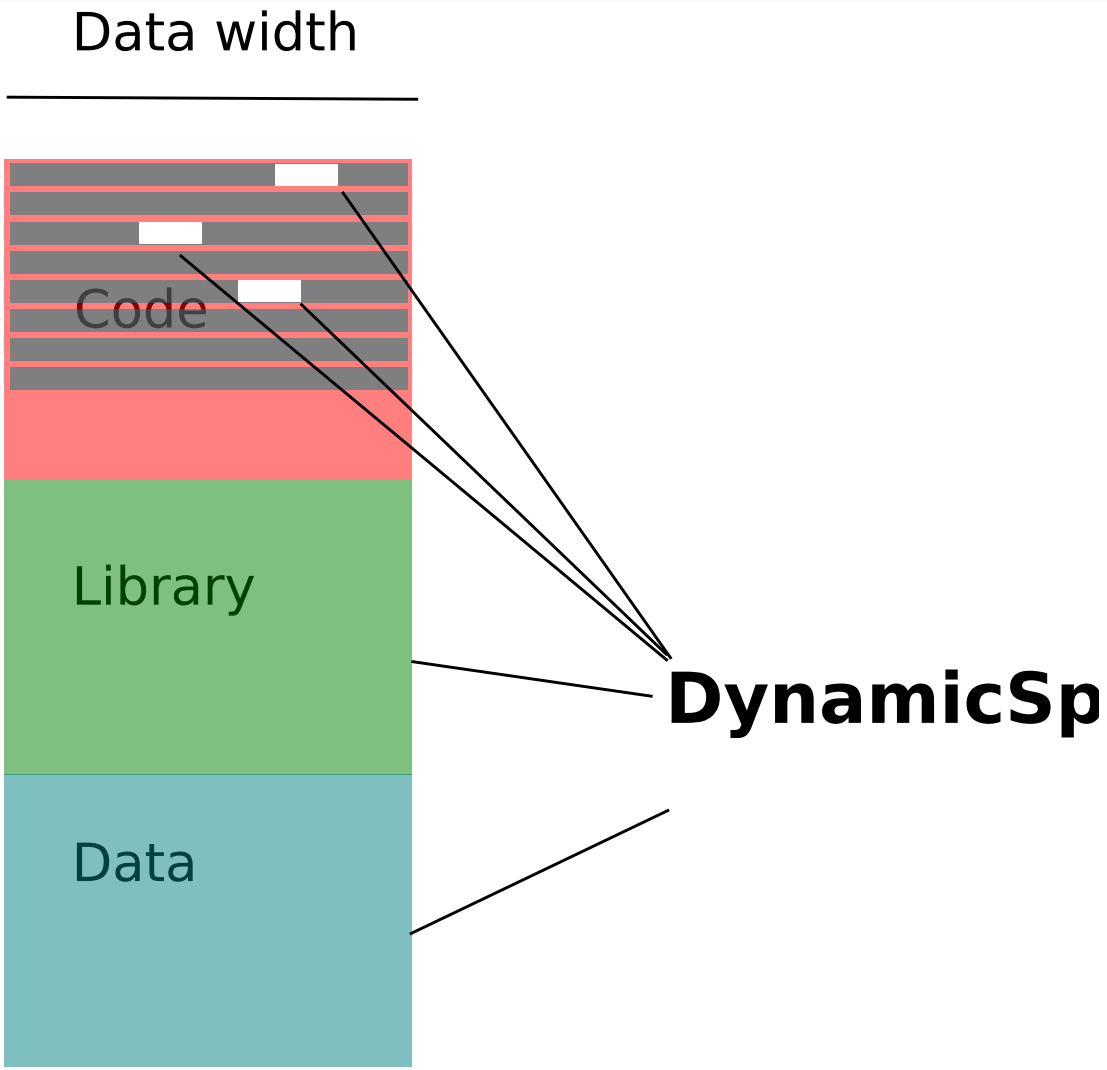
Then compile with :

```
../..
icc -Dstride=5 -S -O Example.c -o Example.s.5
icc -Dstride=6 -S -O Example.c -o Example.s.6
icc -Dstride=7 -S -O Example.c -o Example.s.7
icc -Dstride=8 -S -O Example.c -o Example.s.8
../..
diff Example.s.[57]
```

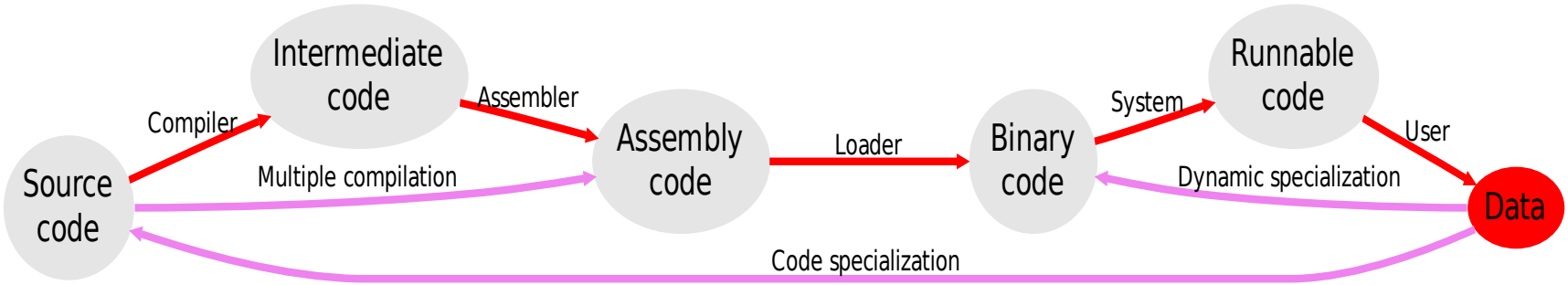
Specialization scheme : compile time



Specialization scheme : run-time



Specialization scheme : compile-time



Using specialization

Static Generate all differents version + a planner (The most often used version and the fallback version)

Dynamic Generate 1 template + a dynamic specializer

Hybrid Generate multiple templates + a cache + a specializer

Compilettes multimedia

Un exemple : 20 loads, 4 stores, 16 mult., 12 add.
Calculs flottants ou entiers

```
#define T 4
void dotProduct(Matrix m, Vector src, Vector dest)
{
  int i;
  for (i= 0;  i < T;  ++i)
    {
      int j;
      dest[i]= 0;
      for (j= 0;  j < T;  ++j)
        dest[i] += src[j] * m[i][j];
    }
}
```

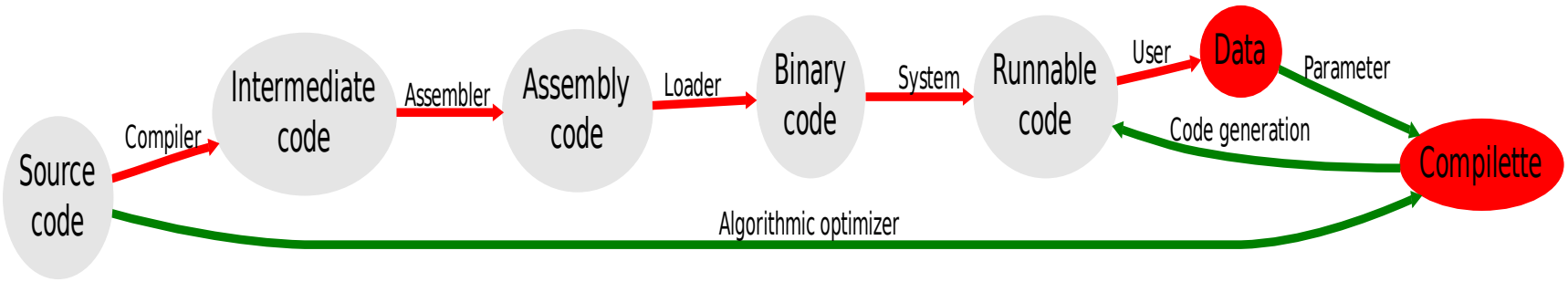
Compilettes multimedia

4 loads, 4 stores, 3 mult., 3 add.

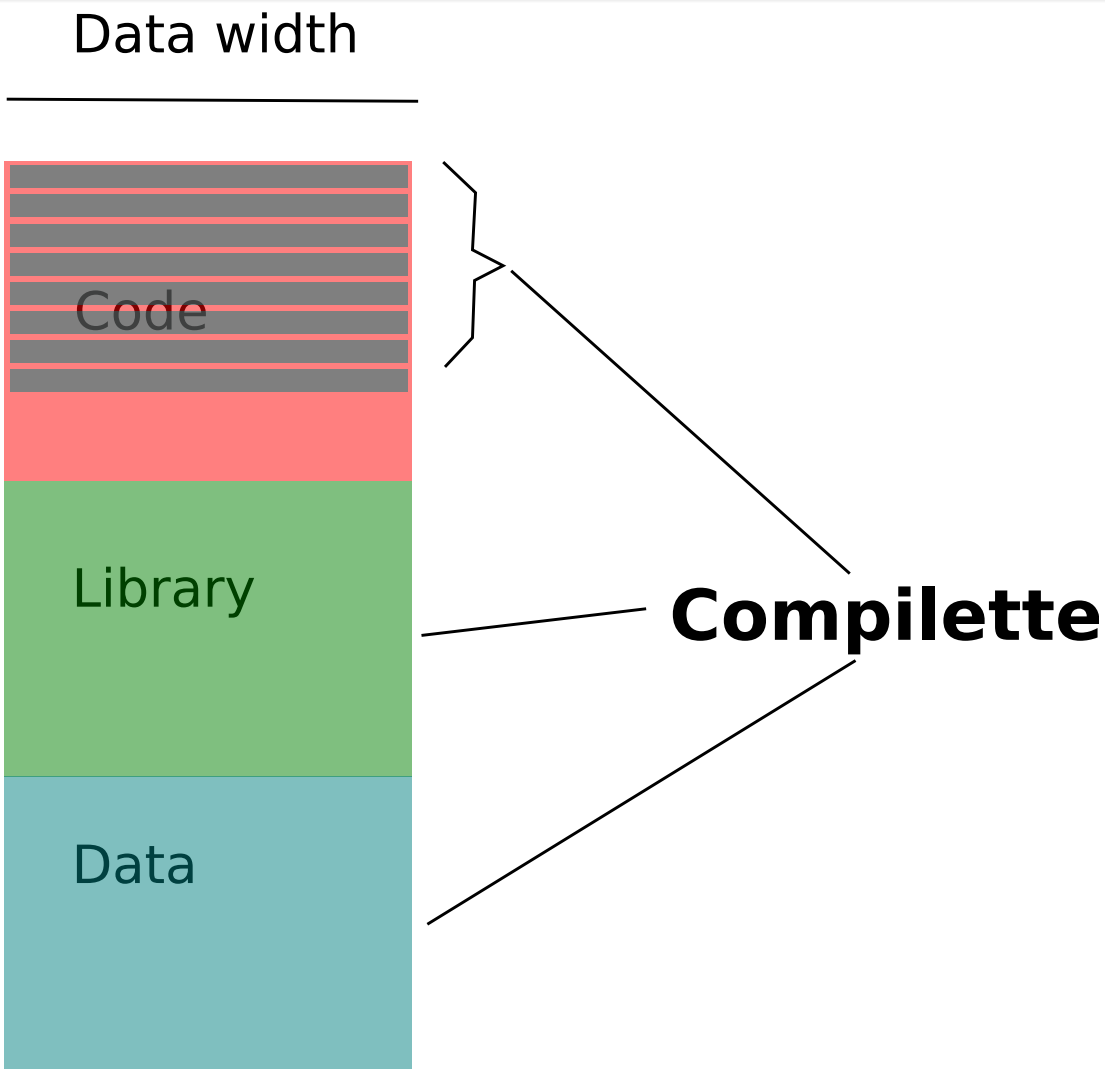
```
/* Translation matrix :
 * 1   0   0   0
 * 0   1   0   0
 * 0   0   1   0
 * Tx Ty Tz   1 */

void dotProduct(Vector src, Vector dest)
{
 dest[0]= src[0];
 dest[1]= src[1];
 dest[2]= src[2];
 dest[3]= src[0] * Tx + src[1] *Ty + src[2] * Tz + src[3];
}
```

“Compilettes”



Principe de foncionnement



Classification Attempt + compilette

Time / Tool	IR	Install	Thinking	Static	Starting	Running
gcc	Gimple/Tree /rtl	✔		✔		
tamarin						✔
GPU/OpenCL	3 Addr Vect. Assembly code			✔	✔	
Qemu	Bin. Native code	✔				✔
Compilette	+/- Arch dep		✔	✔		✔

Compilettes usage

Code specialization as seen before

Run time ISA choose depending on data quantity choose the more adapted ISA.

Run time ISA specialization depending on processor generation specialize the code

Algorithmic optimizer e.g. :

- generate an optimal matrix x matrix depending on matrix values or aligments
- generate a multimedia kernel using specialized instructions

Algoritmic hardware adapter Link high level algoritmic to run-time hardware