



DEVOXX FRANCE

Accès au hardware en Java

pour le développement d'applications industrielles

Accès au hardware en Java pour le développement d'applications industrielles

Thomas Lorblanchès



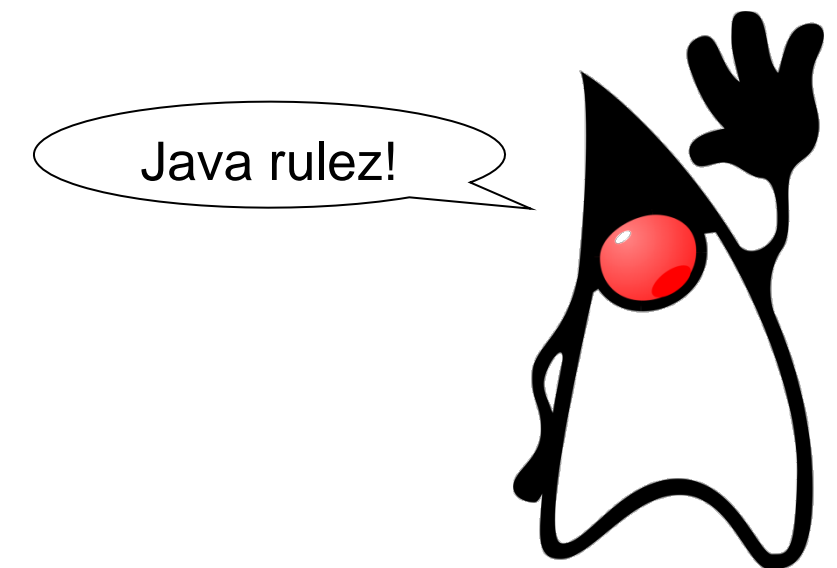
DEVOXX
FRANCE

27 au 29 mars 2013



Java dans le monde industriel

- Pilotage de cartes sur bus PCI/cPCI/PCI-e
 - Cartes d'entrées/sorties
 - Cartes de mesure/acquisition de données
 - Cartes de génération de signaux
 - Cartes d'interface pour bus spécifiques (1553, ...)
- Monde de la « mesure industrielle » dominé par Microsoft
 - Bibliothèques C, .NET, Labview pour Windows
 - Drivers Linux rarement disponibles (sauf dév. spécifique)



JNI : la solution « standard »... à éviter !

Inclus dans le JDK. Solution performante mais complexe :

- Créer l'interface Java enveloppe :

Fonctions « *public static final native* » pour mapper les fct natives

Constructeur static : `System.load("my_lib")`

- Générer le fichier d'en-têtes C/C++ :

`javah nom.du.package.MyClass`

- Coder l'interface Java → fonctions natives en C++

- Compiler la bibliothèque native avec un compilateur C/C++

Ajouter « `${JDK_HOME}/include` » et « `${JDK_HOME}/include/win32` »

- Prier

Exemple de fonction JNI « simple »

```
JNIEXPORT jint JNICALL Java_jni_ApiDll_GetPCIAgentID(JNIEnv *env, jclass theClass,
    jint pciAgentIndex, jobject pciSlotDescription)
{
    SPCISlotDescription cPciSlotDescription;
    int errCode = GetPCIAgentID(pciAgentIndex, &cPciSlotDescription);

    jclass pciSlotClass = env->GetObjectClass(pciSlotDescription);
    if (pciSlotClass == 0)
    {
        printf("GetObjectClass returned 0\n");
        return -1;
    }
    jfieldID pciSlotField1 = env->GetFieldID(pciSlotClass, "vendorID", "I");
    jfieldID pciSlotField2 = env->GetFieldID(pciSlotClass, "deviceID", "I");
    jfieldID pciSlotField3 = env->GetFieldID(pciSlotClass, "busID", "I");
    jfieldID pciSlotField4 = env->GetFieldID(pciSlotClass, "devID", "I");
    jfieldID pciSlotField5 = env->GetFieldID(pciSlotClass, "fctID", "I");

    env->SetIntField(pciSlotDescription, pciSlotField1, cPciSlotDescription.VendorID);
    env->SetIntField(pciSlotDescription, pciSlotField2, cPciSlotDescription.DeviceID);
    env->SetIntField(pciSlotDescription, pciSlotField3, cPciSlotDescription.BusID);
    env->SetIntField(pciSlotDescription, pciSlotField4, cPciSlotDescription.DevID);
    env->SetIntField(pciSlotDescription, pciSlotField5, cPciSlotDescription.FctID);
    return errCode;
}
```

JNA : JNI pour les nuls (et les pressés)

Solution 100% Java :

- Créer l'interface Java enveloppe :

Fonctions « *public static final native* » pour mapper les fct natives

Constructeur static : *Native.register("my_lib")*

- ... C'est tout !

```
class NativeLib
{
    static
    {
        Native.register("my_lib");
    }
    public native int InitDriver(int vendorId, int deviceId)
    public native int GetPCIAgentCount(IntByReference pciAgentCount);
    public native int MemoryRead32(int pciAgentIndex, int barIndex, int dwOffset, int dwSize, int[] rdBuffer);
    public native int InitializeInterrupt(int pciAgentIndex, PciInterruptTransferStruct pciIntTransfer, CallbackIntHandler);
}
```

Pointeur sur int

Pointeur sur tableau de int

Structure

Callback C → Java



ASTRIUM
AN EADS COMPANY

Mapping types JNA ↔ types natifs

Type C	Taille	Type Java
char / short / int	Entier 8/16/32 bits	byte / short / int
long	Entier 32 ou 64 bits	NativeLong
enum	Enumération	int
float / double	Flottant 32/64 bits	float / double
struct* / struct	Structure	Structure / Structure.ByValue
X[]	Pointeur sur tableau	X[], Buffer (NIO)
X*	Pointeur sur un élément	XByReference
typedef X (*callback)(...)	Pointeur sur fonction	Callback

Types natifs non signés : utiliser le mapping « signé »
(généralement pas un problème).

Trucs et astuces JNA

String / char*

- String **immutable** en Java → utilisable en « entrée » de fonction mais pas en « sortie »
- Si c'est une variable modifiée par la fonction :

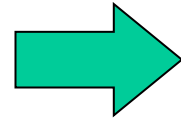
```
public String getDriverVersion()
{
    byte version[] = new byte[20];
    lib.GetDrvVer(version);
    return Native.toString(version);
}
```


Trucs et astuces JNA

Boolean : n'existe pas en C !

- « bool » C++ → 1 octet (en général), donc « byte »
- Types windows : `typedef BYTE BOOLEAN` → byte
`typedef int BOOL` → boolean

```
typedef struct
{
    unsigned char NumBits;
    bool Skip;
    bool LsbFirst;
} bufwordattribstruct;
```



```
class BufWordAttribStruct extends Structure
{
    public byte numBits;
    public byte skip;
    public byte lsbFirst;
}
```

- En paramètre de fonction, extension à 32bits sur la pile → boolean
- Attention à l'alignement !

`#pragma pack(1)` → `setAlignType(Structure.ALIGN_NONE)`

Trucs et astuces JNA

Les versions récentes de JNA obligent à déclarer l'ordre des champs des structures (ordre obtenu par réflexion non prédictif).

```
public static class BufferDescriptionStruct extends Structure
{
    public int byteCount;
    public long userAddr;
    public long physicalAddr;
    public long kernelAddr;
    @Override
    protected List getFieldOrder()
    {
        return Arrays.asList(new String[]
            { "byteCount", "userAddr", "physicalAddr", "kernelAddr" });
    }
}
```

Trucs et astuces JNA

Chemins de recherche des librairies

- Chemins par défaut:

Sous Windows : variable d'environnement « PATH », « . », ...

Sous Linux : variable d'environnement *LD_LIBRARY_PATH*



« . » généralement non inclus

- Déclaration explicite dans le code

```
NativeLibrary.addSearchPath("my_lib", ".");  
Native.register("my_lib");
```

- Propriété system *jna.library.path* (*java.library.path* pour JNI)

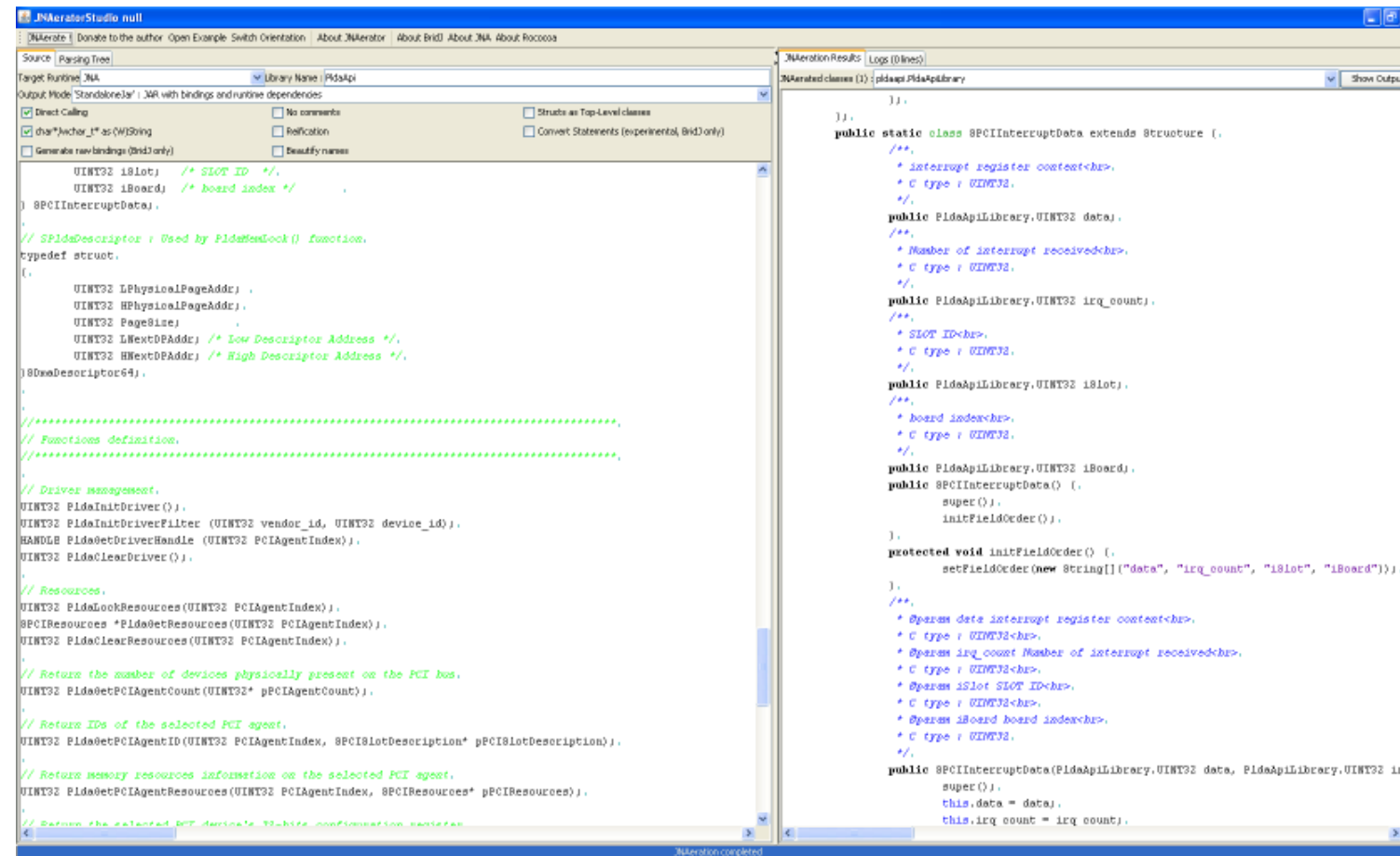
- Bibliothèque insérée dans le JAR :

Pull request <https://github.com/twall/jna/pull/120> (intégrée à future version 3.6)

JNAerator

Generation automatique des interfaces Java JNA/BridJ à partir des fichiers en-tête C/C++.

Pour chaque méthode C, crée plusieurs méthodes Java avec différents mappings possibles pour les pointeurs.



The screenshot displays the JNAeratorStudio application interface. The left pane shows the source C code with various function definitions and comments. The right pane shows the generated Java code, which includes a public static class `SPCIInterruptData` extending `Structure`, and a `public static class PdaApiLibrary` containing several methods that map the C functions to Java. The status bar at the bottom indicates "JNAeration completed".

```
UINTE32 iSlot; /* SLOT ID */
UINTE32 iBoard; /* board index */
} SPCInterruptData;

// SPIDescriptor : Used by PdaApiLibrary() function.
typedef struct {
    UINTE32 iPhysicalPageAddr;
    UINTE32 iPhysicalPageAddr;
    UINTE32 iPageLine;
    UINTE32 iLowDescriptorAddress; /* Low Descriptor Address */
    UINTE32 iHighDescriptorAddress; /* High Descriptor Address */
} SPIDescriptor64;

// Functions definition.
// Driver management.
UINTE32 PdaInitDriver();
UINTE32 PdaInitDriverFilter (UINTE32 vendor_id, UINTE32 device_id);
HANDLE PdaGetDriverHandle (UINTE32 iPCIAgentIndex);
UINTE32 PdaClearDriver();

// Resources.
UINTE32 PdaLockResources(UINTE32 iPCIAgentIndex);
SPCResources *PdaGetResources(UINTE32 iPCIAgentIndex);
UINTE32 PdaClearResources(UINTE32 iPCIAgentIndex);

// Return the number of devices physically present on the PCI bus.
UINTE32 PdaGetPCIAgentCount(UINTE32* piPCIAgentCount);

// Return IDs of the selected PCI agent.
UINTE32 PdaGetPCIAgentID(UINTE32 iPCIAgentIndex, SPCISlotDescription* piSlotDescription);

// Return memory resources information on the selected PCI agent.
UINTE32 PdaGetPCIAgentResources(UINTE32 iPCIAgentIndex, SPCResources* piResources);

// Return the selected PCI device's 32-bit configuration register.

}

}

public static class SPCInterruptData extends Structure {
    /**
     * Interrupt register content<hr>
     * C type : UINTE32.
     */
    public PdaApiLibrary.UINTE32 data;
    /**
     * Number of interrupt received<hr>
     * C type : UINTE32.
     */
    public PdaApiLibrary.UINTE32 irq_count;
    /**
     * SLOT ID<hr>
     * C type : UINTE32.
     */
    public PdaApiLibrary.UINTE32 iSlot;
    /**
     * board index<hr>
     * C type : UINTE32.
     */
    public PdaApiLibrary.UINTE32 iBoard;
    public SPCInterruptData() {
        super();
        initFieldOrder();
    }
    protected void initFieldOrder() {
        setFieldOrder(new String[]{"data", "irq_count", "iSlot", "iBoard"});
    }
    /**
     * @param data Interrupt register content<hr>
     * C type : UINTE32<hr>
     * @param irq_count Number of interrupt received<hr>
     * C type : UINTE32<hr>
     * @param iSlot SLOT ID<hr>
     * C type : UINTE32<hr>
     * @param iBoard board index<hr>
     * C type : UINTE32.
     */
    public SPCInterruptData(PdaApiLibrary.UINTE32 data, PdaApiLibrary.UINTE32 irq_count, PdaApiLibrary.UINTE32 iSlot, PdaApiLibrary.UINTE32 iBoard) {
        super();
        this.data = data;
        this.irq_count = irq_count;
    }
}
```

BridJ : le challenger

« Ultimate Java / native interoperability »

- Interopérabilité avec C, C++, ObjectiveC
- Plus rapide que JNA (en général)
- Java 5+ : utilisation de génériques et annotations

Classe centrale : `Pointer<X>`

- Compatible JNAerator (même auteur)
- Extraction des lib embarquées dans le JAR

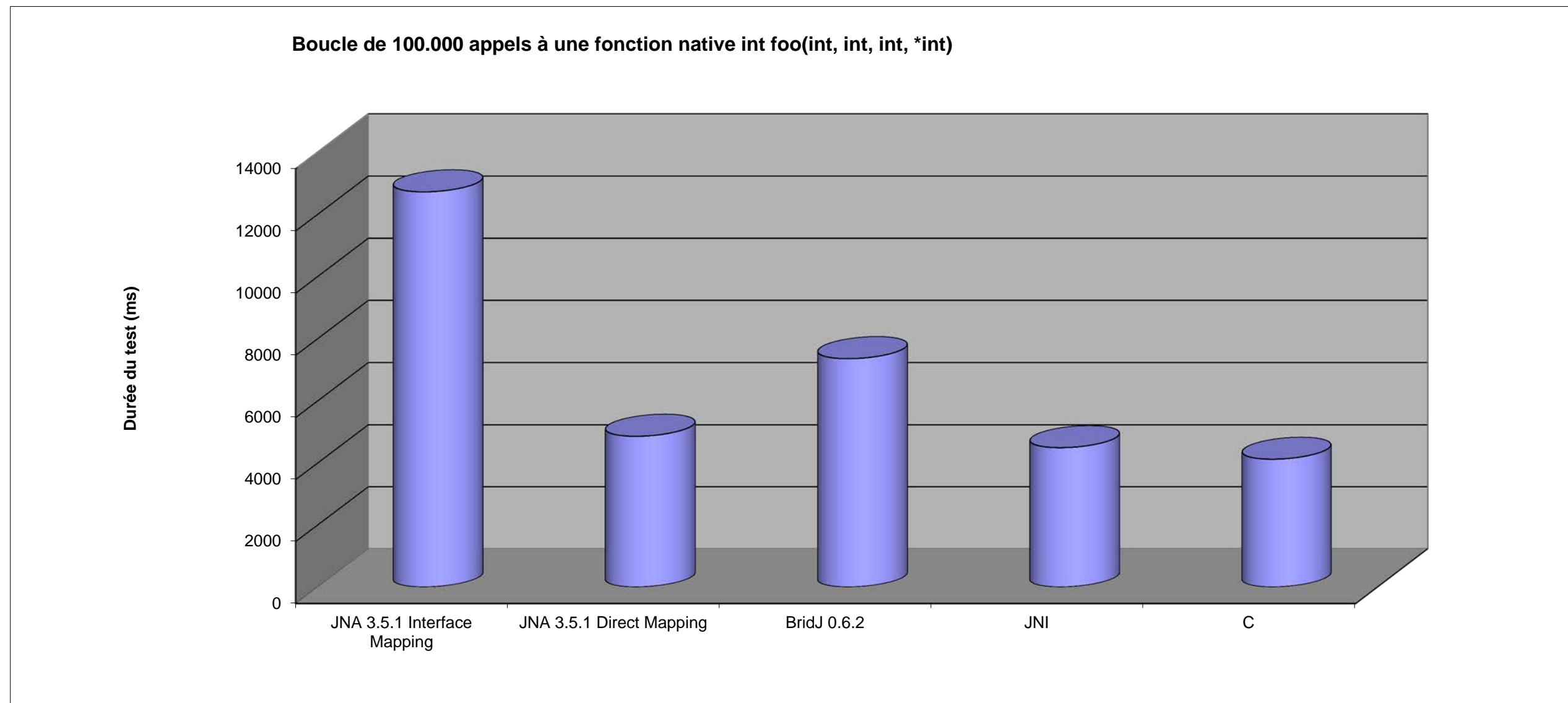
Limitations :

- Pas de passage de structure par valeur
- Projet « jeune » (2010)
- L'utilisation de génériques / annotations ne rend pas toujours le code plus simple / clair que JNA

Benchmark 1

Fonction de lecture mémoire PCI

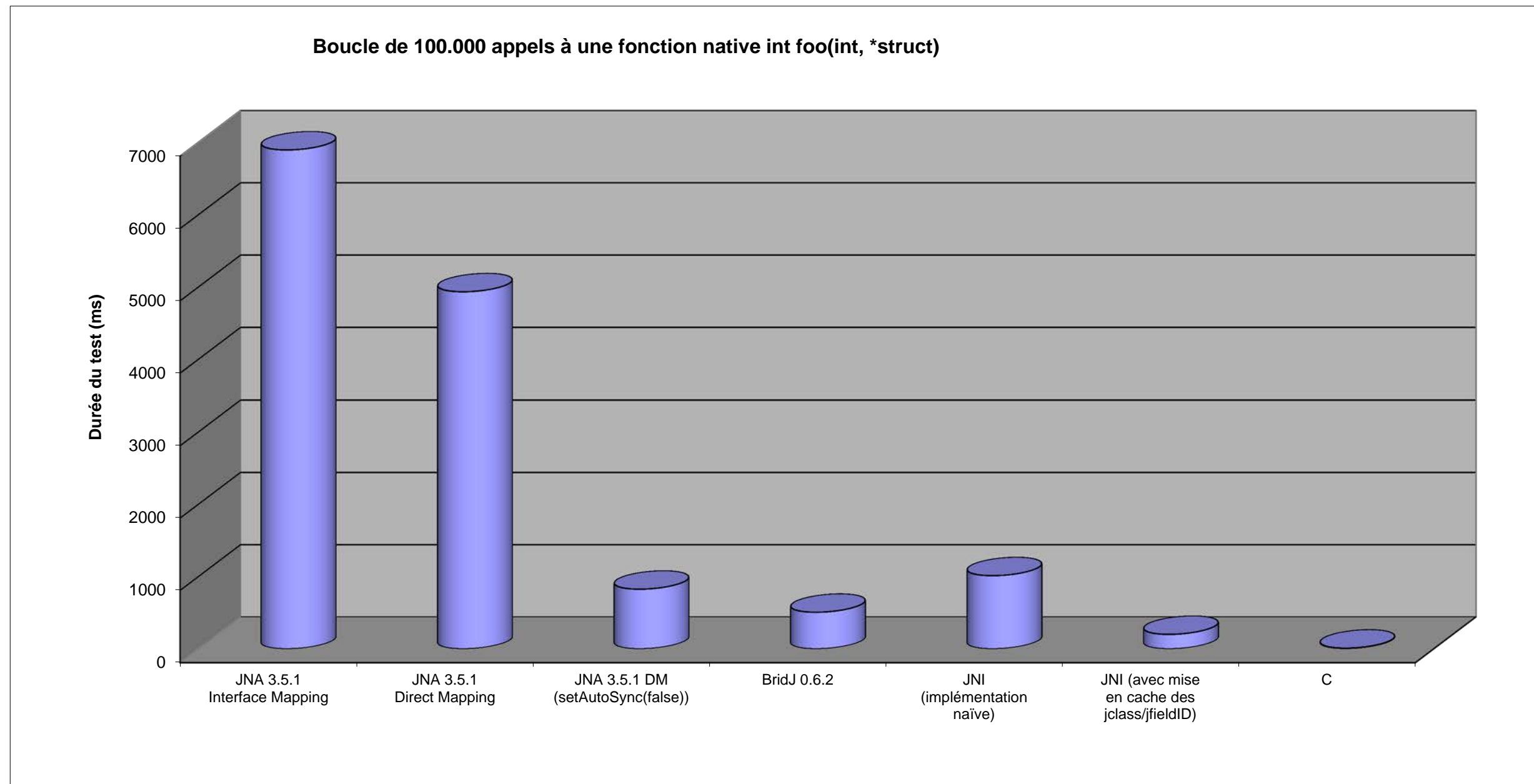
```
int memoryRead32(int pciAgent, int barIndex,  
                int dwOffset, int* data)
```



Benchmark 2

Fonction de récupération d'infos sur carte PCI

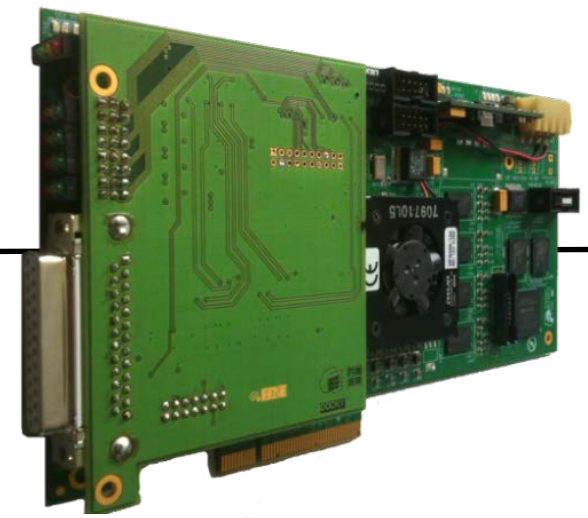
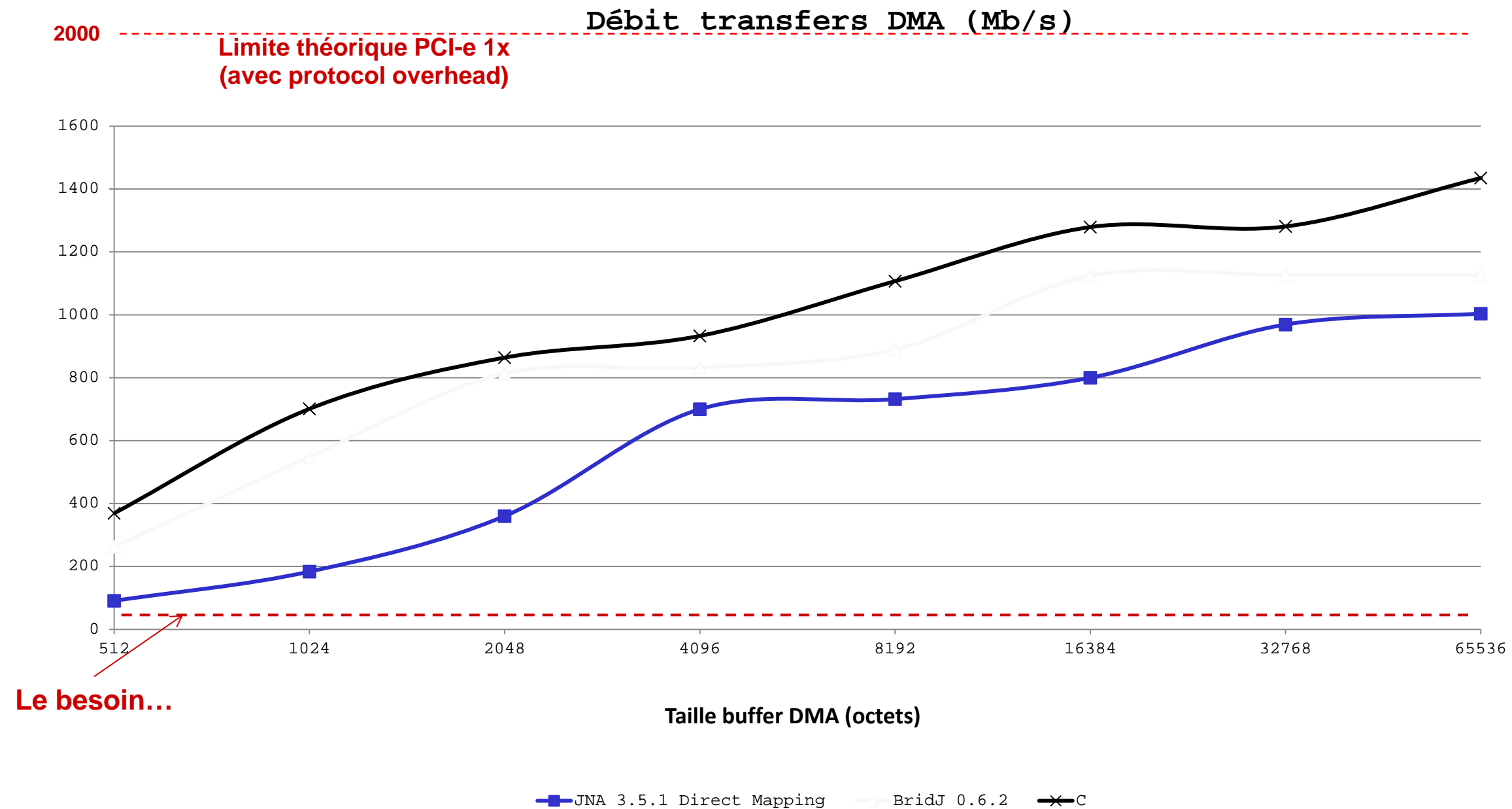
```
int getPciAgentId(int agentIndex, struct* pciSlotDescription)
```



Benchmark 3

Récupération d'un flux de télémesure par transferts DMA

– Mode de test « débit max »



Carte de réception de
télémesure lanceur
IRIG/CCSDS conçue par
Astrium

Conclusion

- Les appels natifs en Java c'est simple !
- La plupart du temps, la perte de performance (par rapport au C) est négligeable.
- Nouvelle API « à la P/Invoke » dans Java 10 ?
 - « From Java SE, 2012, to Java 12: Java SE Roadmap », Oracle OpenWorld 2011: « Calls between Java and Native without JNI boilerplate »

