SISTEMA BASADO EN OPEN SOURCE HARDWARE PARA LA MONITORIZACIÓN DEL CONSUMO DE UN COMPUTADOR

González Rincón, Juan Diego

GRADO EN INGENIERÍA DEL SOFTWARE. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería del Software

Madrid, 19 de junio de 2015

Directores:

Piñuel Moreno, Luis
Igual Peña, Francisco

Autorización de difusión

González Rincón, Juan Diego

Madrid, a 19 de junio de 2015

El abajo firmante, matriculado en el Grado de Ingeniería del Software de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: "SISTEMA BASADO EN OPEN SOURCE HARDWARE PARA LA MONITORIZACIÓN DEL CONSUMO DE UN COMPUTADOR", realizado durante el curso académico 2014-2015 bajo la dirección de Luis Piñuel Moreno y la codirección de Francisco Igual Peña del Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

"Si piensas que vales lo que sabes, estás muy equivocado. Tus conocimientos de hoy no tienen mucho valor más allá de un par de años. Lo que vales es lo que puedes llegar a aprender, la facilidad con la que te adaptas a los cambios que esta profesión nos regala tan frecuentemente."

José M. Aguilar

Agradecimientos

Gracias a Carlos Roa Romero del departamento de Arquitectura de Computadores y Automática por su colaboración y ayuda durante el prototipado.

Índice general

Índice de figuras

Resumen

A día de hoy nuestra sociedad es plenamente consciente de lo que supone el ahorro energético para el medio ambiente. Sabemos que en los centros de procesamiento de datos el consumo energético es realmente elevado y la solución no siempre pasa por obtener una mayor capacidad de computación, sino que en ocasiones puede ser beneficioso sacrificar parte de esta capacidad en favor de una mayor eficiencia energética, con el consecuente ahorro de energía.
El primer paso a dar hasta alcanzar dicha eficiencia, sin embargo, comienza por desarrollar procesos, metodologías y mecanismos que permitan realizar mediciones de consumo fiables, reproducibles y detalladas. Existen en el mercado una amplia variedad de equipos de medición, siendo muy pocos los asequibles económicamente para la gran mayoría de empresas, grupos de investigación o usuarios particulares que requieren de su funcionalidad.

El objetivo principal de nuestro trabajo es proveer de una plataforma precisa, fiable y económica a todos aquellos que quieran ser capaces de medir en tiempo real la potencia eléctrica consumida por un determinado *hardware* de un equipo, principalmente de altas prestaciones, y poder optimizar de esta manera sus procedimientos de cálculo.
Para conseguir nuestro propósito de accesibilidad, la base principal de desarrollo será una plataforma *Open Hardware* de bajo coste, que nos permitirá disponer en nuestro entorno de múltiples dispositivos de manera simultánea sin que ello suponga un gran desembolso.

La combinación de un entorno preciso y de bajo coste nos brindará la posibilidad de, por aproximadamente 150€ , instrumentar código de una manera fácil y muy eficaz.

Palabras clave

Medidor de consumo, instrumentación de código, open source hardware, conversor analógico-digital, sensor de corriente, muestreo de señales, monitorización energética, optimización de algoritmos, análisis de consumo eléctrico.

Abstract

Nowadays, our society is completely conscious of the impact of power saving on the natural environment. As of today, massive datacenters consume a considerable amount of power, and the solution to this problem is not always to attain higher computational throughput, but to trade off part of these computing capabilities for a higher energetic efficiency, with the corresponding energy savings.

The first step towards attaining that efficiency is, however, to develop processes, methodologies and mechanisms that provide reliable, reproducible and precise energy consumption measurements. There exist a wide variety of measurement equipment in the market, but only a few are affordable for small companies, research groups, or even individuals requiring their functionality.

The main goal of our project is to provide a precise, reliable and affordable platform to the community, with the ability to provide real-time power measurements of a given hardware, mainly focusing on high-performance computing facilities, in order to optimize energy efficiency.

To attain our goals in terms of ease of access, the main parts of the project will be based on low-cost *Open Hardware*, which will allow us to attain a powerful measurement environment at a low acquisition cost. The combination of a precise and low-cost environment will give us the possibility of instrumenting a given code in an easy and effective way for roughly 150€.

Keywords

Power measurement, code instrumentation, open source hardware, Analog-to-digital converter, current sensor, signal sampling, energy monitoring, algorithmic optimization, energy consumption analysis.

Capítulo 1

Introducción

1.1. Motivación

En la actualidad, entender el comportamiento de una arquitectura no sólo desde el punto de vista de rendimiento puro sino también atendiendo al consumo energético resulta fundamental para poder conseguir mayores capacidades de cálculo. Este análisis desde el punto de vista de potencia consumida será en gran medida lo que nos permitirá alcanzar y superar la barrera del *exaflop*[1].

Muchos de los entornos hardware actuales permiten por sí mismos la monitorización del uso energético en tiempo real, pero no tienen en cuenta que incluso ese propio análisis a su vez supone un consumo energético que podría alterar los resultados, por pequeño que sea. Además no tiene en cuenta otra serie de factores externos como podría ser la refrigeración activa, sino que se centran exclusivamente, en la mayoría de los casos, en el consumo de la unidad de procesamiento.

Otro dato a tener en cuenta es que pese a la heterogeneidad de las arquitecturas actuales, se producen diferencias drásticas en cuanto a consumo y rendimiento entre unas unidades y otras.

Para mitigar estas carencias, en el mercado nos encontramos con diferentes entornos

[1] **FLOP**: Operaciones en coma flotante por segundo

que nos permiten determinar de una manera precisa y fiable, en tiempo real, el consumo energético de un componente hardware.

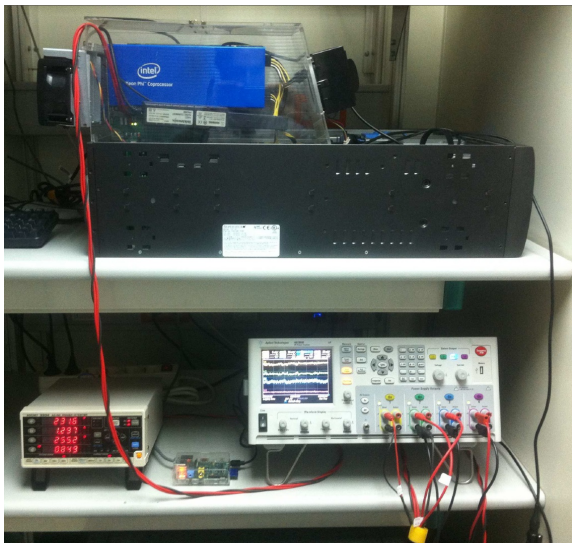


Figura 1.1: *Ejemplo de entorno de medición de potencia en sistemas heterogéneos [1].*

El problema principal con el que nos encontramos con este tipo de entornos *comerciales* es su elevado coste. Además, el gran volumen que ocupan es algo que también debemos de tener en cuenta en entornos de producción (por ejemplo, grandes centros de datos), donde los diferentes componentes están ensamblados a conciencia para optimizar los pequeños espacios de los que se dispone, y que por otro lado nos limitan el número de mediciones simultáneas que podemos realizar. La portabilidad que nos permiten estos dispositivos comerciales es también muy limitada en la mayoría de ocasiones.

Como ejemplo de equipamiento podemos nombrar el analizador de consumo *Agilent N6705B*, utilizado en el presente proyecto como validación del entorno desarrollado, cuyo precio actual en el mercado asciende a los 5686€[2].

Otro ejemplo es el medidor de potencia *Hioki AC/DC HiTester 3334*, utilizado también en nuestro laboratorio y que supera los 1400€[3].

[2] http://es.rs-online.com/web/p/power-quality-analysers/7110012/

[3] http://www.tester.co.uk/hioki-3334-power-meter

Añadir que el ámbito de uso principal de las *herramientas comerciales* es principalmente para empresas y grupos de investigación, lo que dificulta la utilización en un escenario doméstico durante el desarrollo de proyectos menores.

1.2. Objetivos

El objetivo principal es disponer de una plataforma precisa, fiable y económica capaz de medir en tiempo real la potencia eléctrica consumida por un determinado *hardware* de un equipo.

Como hemos visto en el anterior apartado 1.1, la utilización de equipos comerciales conlleva una serie de inconvenientes, algunos de los cuales se solventarán tras la finalización del presente trabajo.
Podemos enumerar los siguientes objetivos de nuestro proyecto:

- **Bajo Coste:** podemos considerarlo el objetivo principal del proyecto, sin por ello tener que sacrificar la calidad en los resultados de la monitorización. Para ello basaremos la plataforma en un dispositivo *Open-Source Hardware*[2]

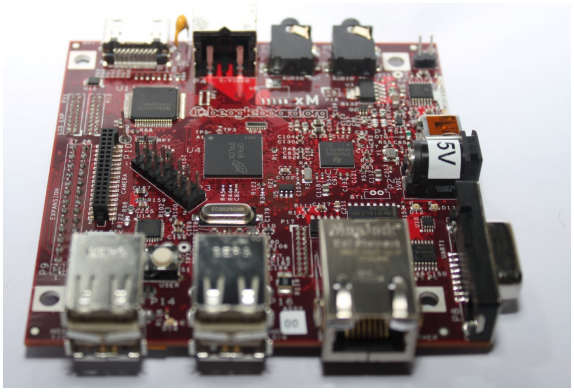


Figura 1.2: *Modelo BeagleBoard xM utilizado en pruebas.*

- **Tamaño reducido:** buscaremos reducir en la medida de lo posible el tamaño del dispositivo, con el fin de hacerlo portable y que pueda ser "replicado" en entornos

reducidos donde las necesidades del número de canales de medición superen a la capacidad de muestreo del dispositivo.

- **Precisión y fiabilidad:** de nada serviría desarrollar un dispositivo que no devuelva datos con una precisión y fiabilidad acorde a las necesidades de los entornos donde será utilizado. Para validar las mediciones nos apoyaremos en entornos ya probados que nos garantizarán la calidad de las medidas.

- **Portabilidad:** además de un tamaño reducido, damos valor a la idea de que sea un dispositivo autónomo y *Plug and play* [3], con el número de conexiones imprescindibles, y que permita ser montado y desmontado sin complicaciones, facilitando su movilidad.

- **Facilidad de uso:** queremos que el entorno sea de fácil manejo, que no requiera grandes conocimientos específicos de la plataforma y el periodo de aprendizaje sea corto, lo cual nos permita realizar mediciones rápidamente.

Para llegar a la solución final iremos evolucionando el proyecto partiendo de entornos probados y fiables que iremos modificando y adaptando con el fin de cumplir nuestros objetivos. Además, nos apoyaremos en la solución software *PowerMeter daemon* (`pmlib` [4]) para integrar todo el desarrollo y que veremos en detalle en la sección 2.3.

Capítulo 2

Entorno experimental

Durante el presente capítulo describiremos el entorno experimental que hemos diseñado para la medición de los consumos durante el proceso de muestreo, cuyo objetivo es medir de forma precisa el consumo energético de aplicaciones ejecutadas sobre plataformas heterogéneas equipadas con uno o múltiples aceleradores *PCI-Express*. De cualquier modo, cabe destacar que el entorno descrito es lo suficientemente modular y extensible como para adaptarse a otro tipo de plataformas actuales o futuras en el ámbito de la computación de altas prestaciones.

2.1. Arquitectura Objetivo

En nuestro caso, la arquitectura objetivo de la medición será un computador heterogéneo equipado con un coprocesador Intel Xeon Phi [5]. Actualmente existen pocos trabajos que analicen el consumo de energía de esta tarjeta y realicen pruebas utilizando sistemas de monitorización [6]. Además, por su relativo poco tiempo en el mercado, resulta de interés conocer el perfil de consumo de este tipo de plataforma.

Intel Xeon Phi es un coprocesador multinúcleo, con hasta 61 unidades de procesamiento (*cores*) con arquitectura x86, y unidades vectoriales extendidas de 512 bits. Puede soportar hasta 8 controladores de memoria, cada uno de ellos con dos canales *GDDR5*. Todo ello conectado al equipo anfitrión mediante un bus *PCIe Gen2*.

Tres son las razones principales para la elección de este dispositivo en detrimento de otros:

- **Intel SMC (System Management Controller):** nos permite obtener información de consumo mediante los sensores que incorpora la propia placa y que pueden ser accedidos a través de *I2C*. Nos devuelven medidas aisladas de la alimentación PCIe y los conectores externos de 12V, lo cual nos permite validar los datos obtenidos con nuestro propio dispositivo. Este nivel de detalle no es posible en otras plataformas como por ejemplo NVML de nVidia.

- **Independiente:** a diferencia de otras plataformas como las GPUs, Intel Xheon Phi puede funcionar de manera independiente al resto del sistema, lo cual nos permite realizar los experimentos únicamente utilizando los recursos del propio coprocesador, algo muy importante para nuestros objetivos de validación.

- **Investigación:** actualmente existen muy pocos trabajos de análisis sobre el consumo de esta plataforma en aplicaciones de Computación de Altas Prestaciones o HPC (High Performance Computing). Esto nos anima a querer contribuir con la investigación de posibles alternativas en lo que a HPC se refiere.

Concretamente, utilizaremos el modelo **5110P**, el cual es alimentado a través de dos vías. Por un lado lo alimentaremos mediante dos lineas PCIe, la primera de 12V y la segunda de 3.3V, capaces de aportar en conjunto hasta 75W de potencia. Por otro lado suministraremos el resto de energía a través de dos fuentes externas de 12V cada una, hasta un máximo de 225W. Por tanto, el pico de consumo de la Intel Xeon Phi 5110P puede ser de hasta 300W; en la práctica, y según la documentación de Intel, la tarjeta puede disipar hasta un máximo de 245W.

Nuestro dispostivo será capaz de monitorizar todas las vías de alimentación simultáneamente con la precisión y frecuencia necesarias.

2.2. Entorno genérico para medición de consumo. Hardware.

El entorno de medición contará con tres partes claramente diferenciadas que nos permitirán ir modificando y adaptando el conjunto hasta llegar a nuestro objetivo. Para ello debemos diseñar un entorno completamente modular que nos permita elegir cada componente en función de las necesidades del momento –precisión, frecuencia de muestreo, número de canales a monitorizar, etc.–, sin que el cambio de un componente suponga una alteración en todo el entorno.

Las partes principales son las siguientes:

- **Unidad Central de Medición** (*Central Measurement Unit* [**CMU**]). Descrita en la Sección 2.2.1.
- **Fuente de Medición** (*Measurement Source* [**MS**]). Descrita en la Sección 2.2.2.
- **Interfaz de Medición** (*Measurement Interface* [**MI**]). Descrita en la Sección 2.2.3.

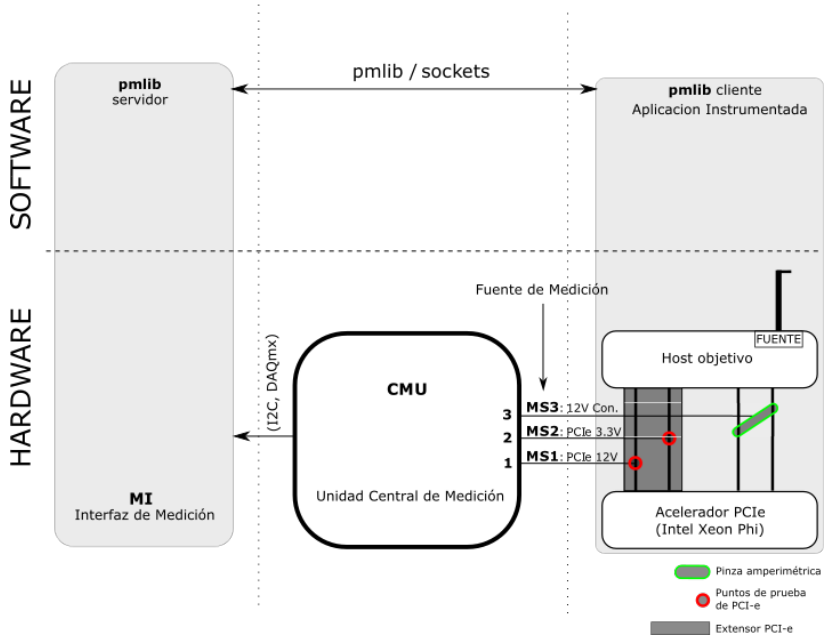


Figura 2.1: *Entorno EnaHPC.*

2.2.1. Central Measurement Unit (CMU)

Es un dispositivo capaz de registrar mediciones de cada una de las fuentes que se desea analizar. Se trata de la parte principal de nuestro entorno. De esta parte dependen factores como la precisión y frecuencia de las mediciones. Resulta ser el componente más costoso de la solución, por lo que es el objetivo principal para reducir el precio del dispositivo final.

Partiendo de la arquitectura descrita en la sección 2.1, podemos acotar más las características que definen nuestro CMU, es decir, para una única Intel Xeon Phi 5110P necesitaremos *al menos 4 lineas* para la toma de mediciones: 2 para las entradas PCIe y otras 2 para las entradas de alimentación externa de 12V.

En cualquier caso, y como queremos tener mayor versatilidad, buscaremos que el número de líneas de medición pueda ser aún mayor, al menos el doble de las necesarias para un único coprocesador en nuestras pruebas.

2.2.2. Measurement Source (MS)

Componente –o parte de un componente– que nos provee de una medida eléctrica que podremos interpretar en términos de potencia. Es otra parte muy importante del entorno, ya que se trata del nexo de unión entre nuestra unidad de medición y la energía real consumida por el componente a analizar.

En el entorno que estamos describiendo, dispondremos de dos fuentes de medición:

- *PCIe*: las actuales tarjetas PCIe utilizan las lineas de 3.3V, 3.3V Aux. y 12V de alimentación. Utilizaremos el extensor *PCIeEXT16HOT* de FURAXA[1], el cual nos proporciona puntos de medición de consumo instantáneo en las tres líneas de alimen-

[1] http://www.furaxa.com/PCIeBusExtenders.htm

tación. La medición es dada en términos de 1 Voltio por Amperio, lo que permite determinar el consumo de cada una de las líneas en un instante determinado.

Recordar que, como comentamos en la sección 2.1, la Intel Xeon Phi únicamente se alimenta de las líneas de 3.3V y 12V del PCIe.

- *Alimentación externa*: necesitaremos monitorizar también las fuentes de alimentación externa de la Xeon Phi. Para ello, en nuestro entorno inicial haremos uso de pinzas amperimétricas, cuyo funcionamiento se basa en el conocido *Efecto Hall*[7]. Estas pinzas nos permiten una gran precisión en la medida y además nos facilitan la portabilidad del módulo.

  Posteriormente intentaremos sustituir estos elementos de elevado coste por otros mucho más económicos que nos sirvan para nuestro propósito.

  Como ejemplo podemos mencionar el modelo disponible en el laboratorio: *Tektronix A622*[2] con la precisión suficiente –desde 50mA hasta 100A de pico–, y 100kHz de frecuencia de muestreo. La problemática es una vez más su coste, que asciende a 683€[3].

2.2.3. Measurement Interface (MI)

Dispositivo que nos permite integrar las unidades de medida en un entorno simplificado para el usuario final. El objetivo es utilizar una plataforma *Open-Source Hardware* de bajo coste que nos permita cumplir este propósito. Además, como explicamos en la sección 2.3, la solución será accesible a través de red, por lo que la plataforma seleccionada deberá disponer de dicha interfaz.

La comunicación entre la **MI** y la **CMU** dependerá de los protocolos que ambos dispositivos soporten, siendo el estándar SCPI el más común, definido en la especificación *IEEE 488.2*[8].

[2] http://www.tek.com/datasheet/current-probe/a621-a622
[3] http://es.farnell.com/tektronix/a622/sonda-ac-dc-current-100a-100khz/dp/7986874

2.3. Entorno genérico para medición de consumo. Software.

Teniendo como objetivo principal la precisión y calidad de las mediciones, no podemos obviar la necesidad de proveer al usuario final de un entorno sencillo, de fácil utilización, que le permita, de la forma más transparente posible, la obtención de resultados. Estos resultados podrán ser capturados con la frecuencia y precisión que el usuario requiera en cada momento, sin para ello tener que modificar el entorno de muestreo.

Para conseguir toda esta funcionalidad descrita, optaremos por una arquitectura cliente-servidor. Con este propósito utilizaremos el *framework* **pmlib**[4].

2.3.1. pmlib

Es una aplicación software desarrollada en *Python* que permite la monitorización del consumo energético y la instrumentación de código en diferentes arquitecturas. Sus desarrolladores, inicialmente, utilizaron soluciones ad-hoc y medidores comerciales para estimar el consumo de una serie de plataformas determinadas.
Uno de los grandes atractivos de **pmlib** es su capacidad de extensión para la adición de nuevos módulos que soporten la comunicación con las distintas **CMU**. Esto sin duda amplía nuestras posibilidades de encontrar soluciones de bajo coste, debido a que, mediante una capa software que realice el "enlace" entre el framework y el dispositivo de medición, podemos proveer al usuario de un entorno nuevo para el muestreo.

Su estructura consta de dos partes diferenciadas:

- **Servidor**: será la parte del software que se ejecute en el **MI** de nuestro entorno y es la encargada de realizar el muestreo en las diferentes **CMU**. Es capaz de soportar

distintos **CMU** de manera simultánea, no sólo de la misma naturaleza, sino también fuentes de completamente distintas.

Nos provee de una API que nos facilita la tarea de instrumentación del código que queremos rastrear.

- **Cliente**: puede ser ejecutado en cualquier entorno con acceso mediante red *Ethernet* al servidor de **pmlib**. En nuestro realizaremos peticiones a la *API* mediante *Python* y *C*, según las necesidades de cada momento.

El único inconveniente que hemos encontrado hasta el momento con **pmlib** es que, dada su implementación, es requisito imprescindible que se ejecute en un sistema *Linux*, algo que por otro lado, en nuestro caso, no es una barrera insalvable debido a que prácticamente la totalidad de plataformas *Open-Source Hardware* funcionan bajo estos sistemas.

Capítulo 3

Implementación del entorno

En este capítulo se explica con todo detalle la evolución del entorno, partiendo del que se describe en la sección 2.2, hasta llegar a la última evolución llevada a lo largo de este proyecto.
Se informa además del ventajas y desventajas en la utilización de una modificación u otra en nuestro entorno de trabajo, así como los inconvenientes o problemas que hayan podido surgir durante esta etapa.

3.1. Entorno de medición inicial

El trabajo inicial parte del entorno elaborado por miembros del Dacya, cuyos resultados ya fueron contrastados, validados y publicados [1].

La **CMU** del entorno es el modelo *N6705B* de *Agilent*, cuyo coste actual en el mercado asciende hasta los 5686€[1], lo cual hace de este componente nuestro primer objetivo.
Como es de esperar, las características del *N6705B* son difícilmente igualables por dispositivos de un coste muy inferior:

- *Voltímetro*: precisión de hasta 0,016 %, 18 bits.
- *Amperímetro*: precisión de hasta 0,016 %, 18 bits.

[1] http://es.rs-online.com/web/p/power-quality-analysers/7110012/

- *Muestreo*: 50kHz con una precisión de 18 bits.

Sin embargo, además del inconveniente de su precio, también tiene otra limitación básica, y es el número de canales que puede monitorizar, 4 en total. Es algo que queremos mejorar en la medida de lo posible.

Además del *N6705B*, el entorno inicial se complementa con el medidor de corriente alterna *Hioki AC/DC HiTester 3334* que permite medir el consumo global del sistema.

3.2. Utilización de un DAQ como CMU

La primera alternativa con la que hemos trabajado ha sido un **dispositivo de adquisición de datos** o DAQ (*Data Acquisition*) de la compañía *National Instruments*, referente en este tipo de dispositivos.
El modelo elegido es un **NI USB-6218**[2], con un rendimiento, en principio, excepcional y suficiente para nuestro propósito:

- 32 entradas analógicas.
- 16 bits de precisión en cada entrada analógica.
- 250.000 muestras por segundo.

Para llevar a cabo la primera toma de contacto decidimos instalar el entorno **DS-5**[3] de **ARM**, el cual es compatible con Microsoft Windows®y Linux, y permite la captura de datos utilizando dispositivos DAQ de National Instruments. Su versión *Ultimate* nos da la posibilidad de, mediante un módulo instalado en el dispositivo que queremos monitorizar, capturar los procesos y el uso de recursos del sistema de manera sincronizada con el consumo eléctrico.

[2] http://sine.ni.com/nips/cds/view/p/lang/es/nid/203484
[3] http://ds.arm.com/

ARM pone a disposición de los usuarios una serie de tutoriales y documentación para la iniciación con su entorno DS-5, los cuales también han sido útiles para nuestra iniciación con el DAQ.

Siguiendo su documentación [4], hemos procedido a realizar el primer montaje[5] sobre una placa de prototipado y el propio DAQ:

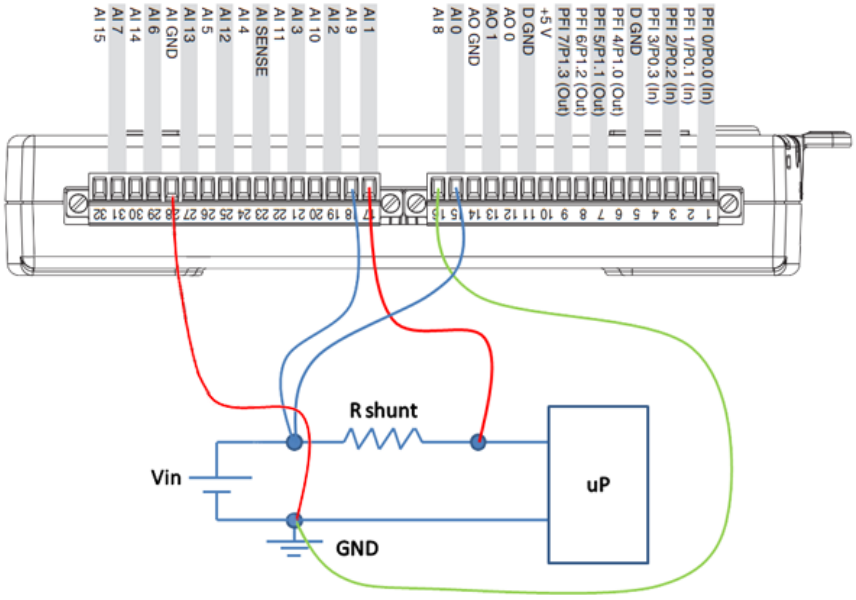


Figura 3.1: *Conexiones NI USB-6218.*

Optamos por utilizar una placa Open-Source Hardware como sistema objetivo de medidas, la conocida **BeagleBoard xM *Rev B***[6] que tiene como característica para nuestro propósito la de disponer de una resistencia interna de *shunt* de alta precisión, mediante la cual podremos calcular el consumo de la placa.

Otra característica muy interesante de la placa es la posibilidad de monitorizar el consumo mediante acceso I2C al integrado de *Texas Instruments* **TPS65950** que incorpora la propia placa.

[4] https://www.youtube.com/watch?v=Dqzh1N3RGFQ

[5] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0482q/ric1362074310560.html

[6] http://beagleboard.org/beagleboard-xm

Figura 3.2: *BeagleBone xM utilizada en las pruebas.*

3.2.1. Instalación de DS-5 y su demonio *gator*

Elegimos la versión *DS-5 Ultimate* con una licencia de evaluación de 30 días, la cual nos permite utilizar todas las funcionalidades del entorno; en concreto utilizamos *DS-5 Streamline*.
Elegimos la versión para Microsoft Windows®.

Una vez instalado el entorno en el equipo, necesitamos instalar los controladores del *NI USB-6218*[7]. National Instruments pone a disposición de los usuarios dos controaldores:

- **DAQmx**: funcionalidad completa con los dispositivos DAQ.
- **DAQmx Base**: funcionalidad reducida (por ejemplo *DAQ Assistant*), con la ventaja de ser compatible con Linux y Windows®.

Para nuestro propósito ambas versiones son plenamente compatibles, por lo que bajo un entorno Windows® optaremos por la versión completa.

[7] http://www.ni.com/downloads/ni-drivers/

Con ésta configuración ya podemos realizar las primeras pruebas de medición para comprobar que el DAQ es reconocido por el equipo anfitrión. Utilizamos la propia herramienta de prueba que se instala con el controlador de National Instruments.

Como podemos observar, se realizan mediciones de tensión de manera diferencial, con valores aparentemente razonables, teniendo en cuenta la tensión de funcionamiento de la Beagle.

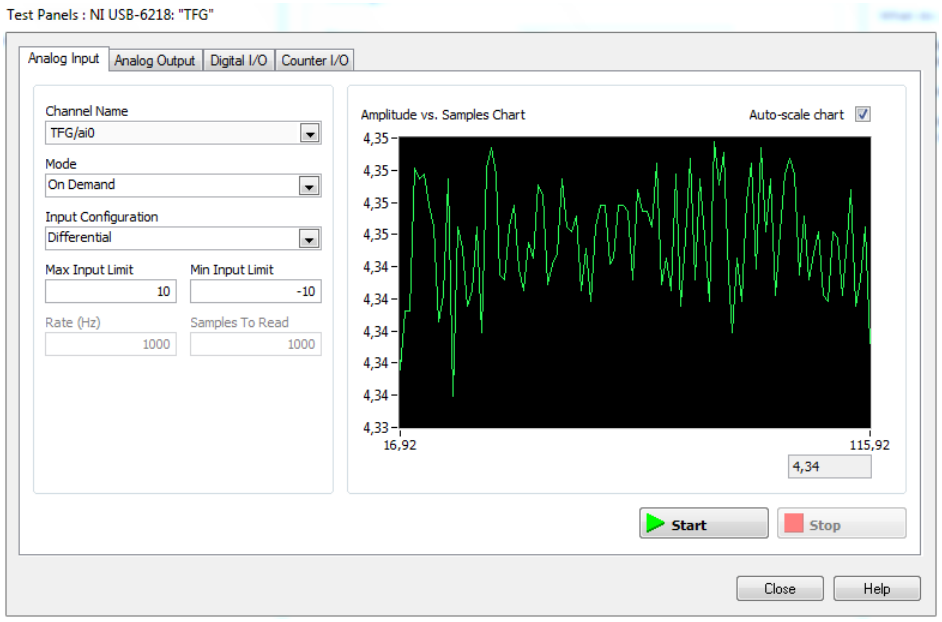


Figura 3.3: *Primera captura de datos con NI USB-6218.*

La siguiente tarea que nos ocupa es la de seleccionar una distribución Linux compatible con nuestra plataforma objetivo, la *BeagleBoard xM*. Para ello probamos varias, como consecuencia de los diferentes problemas que nos reporta cada una de ellas:

- **Angstrom**: antigua y sin prácticamente soporte por parte de la comunidad. Los repositorio están inaccesibles.

- **Debian** (distintas versiones): una versión reducida de consola, muy liviana, pero da problemas de red y no inicia correctamente los servicios del arranque.
- **Fedora** (distintas versiones): no funciona con la *Rev B* que tenemos.
- **Ubuntu**: Al igual que la anterior, no es compatible con *Rev B*.

Finalmente optamos por una versión **Debian** de Junio de 2014, por ser la más "estable" y completa de las probadas. Tras distintas pruebas, conseguimos configurar y solventar los problemas que nos causaba en la primera toma de contacto.

Para obtener comparativas del consumo con el uso de recursos de la *BeagleBoard xM*, necesitamos instalar el demonio **gatord**. Si queremos una funcionaildad completa, también necesitamos compilar e instalar un módulo en el *Kernel*[9] del sistema objetivo.

Debido al poco soporte y actualización de la distribución *Debian* elegida –en parte por la antigüedad de la versión de la placa–, resulta compleja la compilación del módulo, principalmente por la falta de versiones de las librerías para arquitecturas *ARM*.
Finalmente conseguimos que funcione correctamente el módulo del Kernel **gator.so**, que nos permitirá el acceso completo al sistema mediante el *Streamline*.

El método de acceso al sistema objetivo es mediante interfaz de red. Simplemente necesitaremos conocer su IP para introducirlo en el entorno DS-5, y podremos empezar a tomar mediciones.

Con el fin de comprobar que los datos que capturamos con el DAQ son fiables, decidimos realizar una serie de pruebas que hagan uso de los recursos de la *BeagleBoard xM*, lo cual necesariamente provocará un cambio en su consumo, y que, si todo funciona correctamente, podremos observar.

Antes de visualizar los resultados de las pruebas, hay que indicar que la resistencia de *shunt* de la placa es de **0.1 Ohmios**, lo que nos indica que la caída de tensión en ella es de 0.01 Voltio por cada 100 miliamperios de consumo.

En primer lugar realizamos unas pruebas de estrés, las cuales hacen uso principalmente de la memoria RAM, la unidad de almacenamiento y el procesador de la Beagle.

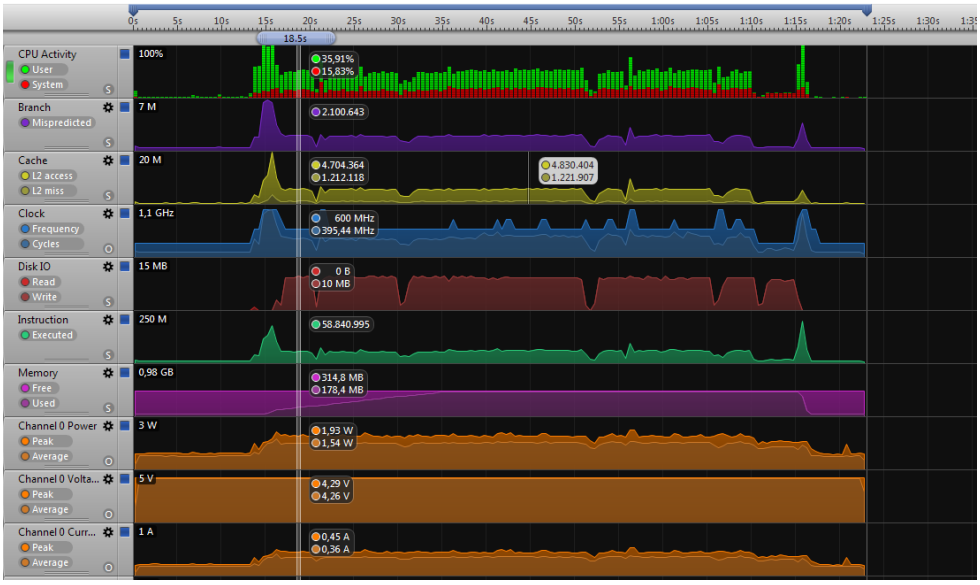


Figura 3.4: *Resultados de medidas con el NI USB-6218 en DS-5 de ARM.*

Posteriormente lanzamos otro test, en esta ocasión optimizado para procesadores *Cortex*; nuestra placa utiliza el procesador *Cortex A8*. El software elegido es BLIS[8][10].

Todos los datos obtenidos nos indican que nuestro nueva **CMU** con el *DAQ NI-USB6218* funciona correctamente, pero para cerciorarnos realizaremos una prueba más.

[8] https://github.com/flame/blis

Figura 3.5: *Montaje BeagleBone xM con NI USB-6218.*

3.2.2. Integrado TPS65950 como medidor de consumo

Además del *jumper* J2 con la resistencia de *shunt*, la *BeagleBoard xM* dispone de un integrado capaz de monitorizar el consumo de su procesador, se trata del **TPS65950**[9] de *Texas Instruments*.

Las lecturas con el integrado **TPS65950** se realizan a través de comunicación por el bus I2C[11], en concreto por el bus número 0 –I2C_0– de la *BeagleBoard xM*.
La lectura se lleva a cabo mediante los pines analógicos ADCIN3 y ADCIN5 del propio integrado. Para poder acceder al valor leído en estas dos entradas debería ser necesario con activar el ***MADC***[10] y leer los registros correspondientes a cada una de las dos entradas analógicas. Deberemos tener en cuenta que la conversión de la lectura analógica a la salida digital acarrea un retardo cuya duración dependerá de la frecuencia de reloj que gobierne el ***MADC***.

[9] http://www.ti.com/product/tps65950
[10] Subsistema del TPS65950 que consiste en un conversor analógico-digital de 10 bits con 16 entradas multiplexadas

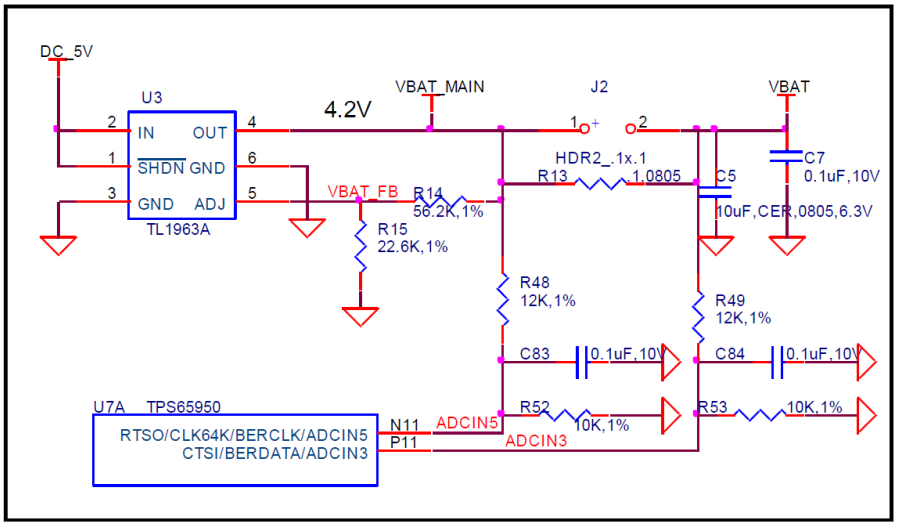


Figura 3.6: *Medidor de consumo del procesador de la BeagleBoard xM.*

Con todo conectado nos disponemos a realizar las lecturas utilizando la librería I2CTools [11], desde la propia *BeagleBoard xM*. Tras diversos intentos comprobamos que siempre se obtienen lecturas nulas –pese a producirse correctamente la comunicación–, por lo que intuimos que hay algo que no está funcionando como debería.

Empezamos una serie de pruebas con el fin de detectar el posible fallo. Para ello nos centramos en el manual técnico del integrado [12] revisando los registros involucrados en la operación que queremos efectuar.

Lo primero que observamos es que en el registro **CTRL_SW1** del banco con *dirección0x40* el bit **EOC_SW1** –número 1– permanece siempre con valor 0, lo que nos indica que la operación de conversión del **MADC** no se ha terminado. Además, el bit BUSY –número 0– del mismo registro se encuentra permanentemente a 1, lo que nos indica que hay una operación en curso y el dispositivo está ocupado.

[11] http://www.lm-sensors.org/wiki/I2CTools
[12] http://www.droid-developers.org/images/2/21/Tps65950_TRM.pdf

Finalmente y tras muchas revisiones y pruebas comprobamos que el reloj del conversor analógico-digital está desactivado, pese a que en la documentación del integrado referente a éste registro indica que está activado. La solución consiste en activar los bits **MADC_HFCLK_EN** y **DEFAULT_MADC_CLK_EN** –números 7 y 4 respectivamente– en el registro **GPBR1** en el banco con *dirección 0x49*.

Con el conversor ya activo, verificamos que se producen lecturas. En este punto optamos por desarrollar un *script* de captura en **BASH**[12] que nos permite indicar la frecuencia y el número de muestras. Véase *Apéndice A*.
Al igual que en pruebas anteriores, lanzamos procesos que hacen uso de recursos y que nos permiten ver alteraciones en el consumo.

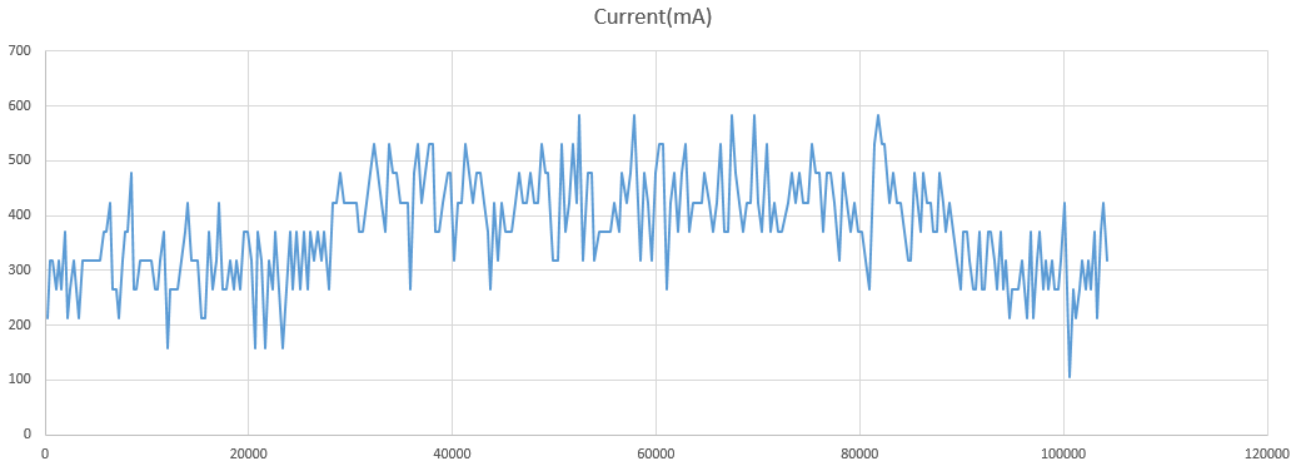


Figura 3.7: *Gráfica de consumo obtenida mediante el integrado* ***TPS65950****.*

Si comparamos los datos con los obtenidos (Figura 3.8) con el *DAQ* (Figura 3.4) en pruebas anteriores, podemos observar que son coherentes.

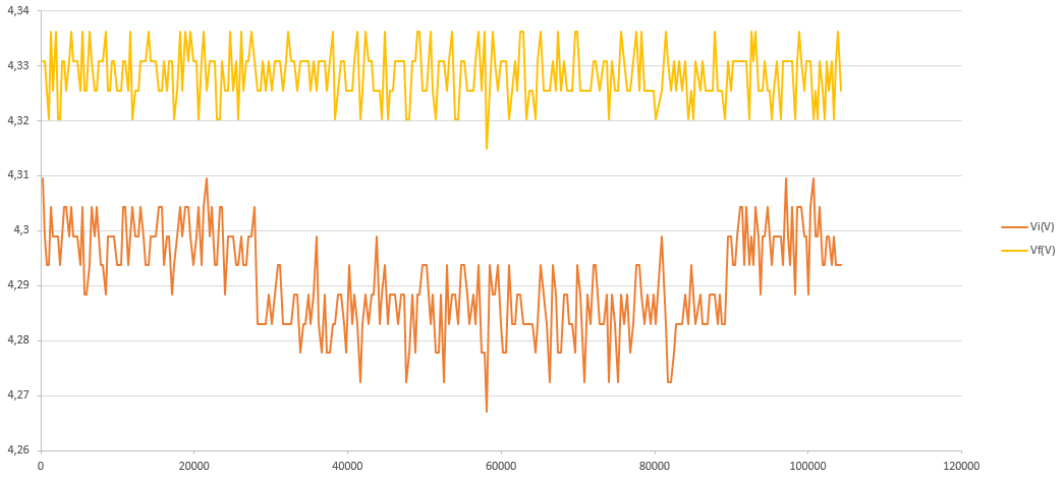


Figura 3.8: *Gráfica de tensiones obtenida mediante el integrado* ***TPS65950****.*

3.2.3. Módulo pmlib para NI USB-621X

Habiendo sustituido el **CMU** por el *DAQ NI-USB6218*, el siguiente paso consiste en integrarlo en nuestra *interfaz de medición* para que nos permita realizar muestreos mediante peticiones a la *API* de **pmlib**.
Con este propósito hemos desarrollado un módulo para **pmlib**.

Como hemos explicado en la sección 2.3.1, es requisito imprescindible correr el **MI** en un entorno *Linux*. Según la documentación de *National Instruments*, las versiones de este sistema soportadas por su controlador son:

- *openSUSE 12.3 y 13.1*
- *Red Hat Enterprise Linux WS 6*
- *Scientific Linux 6.x*

Elegimos **Scientific Linux 6** para correr el servidor de **pmlib** y desarrollar el módulo que soporte el *DAQ NI-USB6218*.

Con la instalación del controlador del dispositivo de *National Instruments*, tenemos acceso a su la *API* utilizando las librerías desarrolladas en lenguaje C. El problema que esto nos plantea es la integración con **pmlib**, que recordemos, está desarrollado en *Python*.
Nos planteamos las siguientes opciones:

- Desarrollar una aplicación en C que permita acceder al dispositivo y que sea accedida mediante un módulo *Python* de **pmlib**.
- Ejecutar código C en el propio módulo de *Python*.
- Utilizar una solución intermedia haciendo uso de la interfaz **PyDAQmx**[13].

Optaremos por la solución que nos proporciona **PyDAQmx** por considerarla la más versátil y con mayores posibilidades de integración. Por contra la documentación de esta librería es escada, por lo que vemos obligados a hacer uso del manual de referencia de la *API* de C de *National Instruments*[13].

Para cerciorarnos que el acceso a la *API* funciona correctamente, como primera toma de contacto, antes de desarrollar el módulo, realizamos una serie de pruebas. Primeramente en un entorno Windows® con el controlador **NI DAQmx Base**, *Python 2.7* y el módulo **PyDAQmx** instalado, realizamos un primer intento de toma de datos, el cual resulta fallido. Tras revisar la documentación de **PyDAQmx** comprobamos que en la versión de Windows® se hace uso de la librería **nicaiu**, que se instala únicamente con la versión completa del controlador, **NI DAQmx**. Solventado este inconveniente, comprobamos que la captura de datos simple en un único canal se produce correctamente.

A continuación realizamos pruebas en el entorno con *Scientific Linux 6*. Haciendo uso de los ejemplos en lenguaje C que podemos encontrar en la documentación del driver, comprobamos que la comunicación con el *DAQ* se realiza correctamente y se realizan lecturas.

[13] https://pypi.python.org/pypi/PyDAQmx

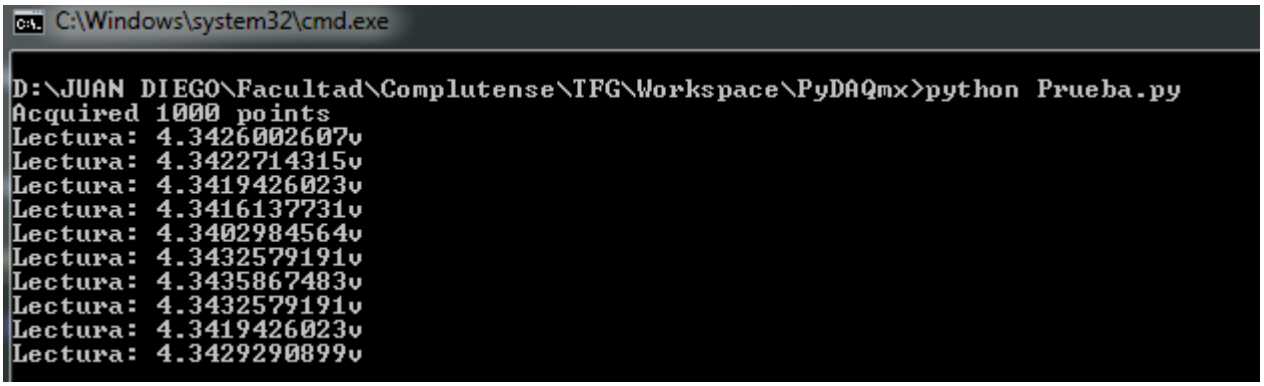


Figura 3.9: *Primera toma de medidas en un canal con PyDAQmx y el DAQ NI-USB6218.*

Posteriormente ejecutamos el ejemplo *Python* que probamos en el entorno Windows® y comprobamos que efectivamente se realizan lecturas.

Con esta serie de comprobaciones realizadas comenzamos a desarrollar el módulo para **pmlib** que realice las mediciones en distintos canales de manera simultánea. Aquí nos hemos encontrado con otra dificultad, que viene detallada en la documentación[14]; consiste en un error al realizar peticiones en un bucle infinito, ya que cada proceso de toma de medidas analógicas requiere un cierto tiempo, durante el cual el *DAQ* queda bloqueado y no admite más peticiones.
Para solucionar el problema decidimos realizar medidas por lotes, es decir, en lugar de realizar una medida en cada petición, se solicita un número determinado de medidas de tal forma que el procesamiento de los valores devueltos ocupe el tiempo suficiente al servidor hasta que sea posible realizar una nueva petición al dispositivo.

A continuación indicamos los parámetros de configuración que hemos modificado para realizar las mediciones:

- Función **DAQmxCfgSampClkTiming**, parámetro *numSampsPerChan*: Nos permite indicar el número de muestras por canal en la petición en curso.

[14] http://digital.ni.com/public.nsf/allkb/4159A4241B8A24B3862570EC007AB6B7

- Función **DAQmxReadAnalogF64**, parámetro *terminalConfig*: Las lecturas las realizamos con referencia a tierra –GND–, por lo que debemos dar el valor *DAQmx_ Val_ RSE* a éste argumento.

- Función **DAQmxReadAnalogF64**, parámetro *fillMode*: Especifica si las muestras de los diferentes canales se intercalan o se muestran agrupadas. En nuestro caso la trataremos estando intercaladas, por lo que este parámetro lo pondremos con el valor *DAQmx_ Val_ GroupByScanNumber*.

Como en el resto de módulos de **pmlib**, es necesario instanciar la clase e indicar sobre que líneas se quiere realizar la medición. A continuación se muestra y se explica un ejemplo de uso del módulo **DAQmxDevice** –*Apéndice B*– desarrollado:

DAQ_msr=DAQmxDevice(name="DAQmx", computer=ARM, url="Dev1", max_frequency=1000)

- *name*: Nombre del dispositivo de medición que se muestra en **pmlib**.

- *computer*: Objeto creado en la configuración con especificaciones del dispositivo anfitrión.

- *url*: Nombre físico del dispositivo con el que es reconocido en el sistema en el que está conectado. Podemos utilizar la orden *lsdaq* en entornos *Linux* para mostrar los dispostivos *DAQ* conectados.

- *max_frequency*: Frecuencia máxima de muestreo que queremos que admita **pmlib** para nuestro dispositivo.

DAQ_msr.add_line(number=0, name="ai1", voltage=10, description="AI1")

- *number*: Número de línea añadida al dispositivo. Son números correlativos comenzando desde 0 para cada una de las líneas de muestreo.

- *name*: Nombre físico del canal. En la documentación del *DAQ NI-USB6218* vienen especificados los nombres de cada uno de sus canales.
- *voltage*: Tensión máxima que soportará el canal.
- *description*: Breve descripción del canal.

Llegados a éste punto, en el que podemos realizar mediciones en distintos canales de manera simultánea, observamos que en algunas ocasiones se perdía la comunicación con el *DAQ NI-USB6218* cuando era conectado a un sistema **Scientific Linux 6**. Para solucionarlo optamos por actualizar el *Firmware* siguiendo las instrucciones de la documentación oficial[15]. Tras esta actualización, los problemas se solventaron.

3.3. Otras vías alternativas exploradas: DAQ como CMU y Raspberry Pi como MI

Con el desarrollo del módulo **DAQmxDevice** para **pmlib**, las posibilidades de expansión pueden ser mayores. Sabemos que la *API* a la que tenemos acceso cuando instalamos el controlador **NI DAQmx** o **NI DAQmx Base** es compatible con diferentes dispositivos de adquisición de datos de *National Instruments*.
Al disponer de otro modelo de DAQ, el *NI-USB6009* decidimos crear una solución compacta que consista en utilizar este dispostivo –de menor tamaño que el *NI-USB6009*– junto con un computador de bajo coste y de pequeñas dimensiones.

Las primeras pruebas las realizamos en una *Raspberry Pi® Model B+*, última evolución de la primera versión de este dispositivo. Existe un *driver* sencillo para esta placa que soporta el *DAQ*[14] en las distribuciones más comunes de la plataforma, que viene acompañado por unos binarios que nos permiten la captura de mediciones mediante la consola de comandos.

[15] http://digital.ni.com/public.nsf/websearch/91CBCFE9E171C845862572CF0077D8F9?OpenDocument

Figura 3.10: *Dispositivo de adquisición de datos NI USB-6009.*

En este punto nos hemos planteado la posibilidad de realizar otro módulo para **pmlib** que realice las lectura haciendo uso de los binarios mencionados, pero hemos descartado esta posibilidad debido a las limitaciones que esto supone, y es que únicamente se podría monitorizar un canal de manera simultánea, y en el caso de realizar lecturas secuenciales en los distintos canales del *DAQ*, la frecuencia de muestreo obtenida sería muy limitada, impidiéndonos tener mediciones con la precisión que se requiere.

Por el motivo anterior nos planteamos la posibilidad de instalar el controlador **NI DAQmx Base** en la propia *Raspberry Pi*® . Recordamos que los sistemas soportados oficialmente son *openSUSE 12.3 y 13.1*, *Red Hat Enterprise Linux WS 6* y *Scientific Linux 6.x*. Los sistemas más comunes de *Raspberry Pi*® como Raspbian están basados en *Debian*, por lo que el driver oficial de *National Instruments* no es compatible.
Hemos probado entonces otra distribución basada en *Red Hat*, conocida como Pidora. En el momento de compilar nos hemos encontrado con un problema de versiones para estas distribuciones concebidas para dispositivos con arquitectura *ARM* y es que no hay disponibles todas las versiones actualizadas de las librerías necesarias para compilar el controlador **NI DAQmx Base**, en concreto la librería estándar de C –libstdc–, de cuya versión más actual

para Pidora encontramos la 4.4, mientras que la que necesitamos es al menos la versión 5.

Debido a la obsolescencia de la plataforma, optamos por probar la más actual, *Raspberry Pi® 2*, para la cual disponemos de más sistemas operativos basados en *Red Hat*. Instalamos **Redsleeve**[16] y procedemos a compilar el controlador del DAQ.
En ésta ocasión, tenemos más dificultades para realizar el proceso debido a discrepancias entre las versiones del *Kernel* instalado y los *headers* disponibles, además de la diferencia de versiones entre el compilador **gcc** disponible en el sistema y el utilizado para compilar la propia distribución.
Tras modificar el script de instalación del controlador, llegamos al mismo punto que en el caso anterior con Raspbian, y es que la misma librería tampoco se encuentra disponible en la versión 5 para Redsleeve.

Por ello descartamos la utilización del *DAQ* directamente en un computador de bajo coste y decidimos utilizarlo con un equipo convencional para realizar pruebas y validar los resultados de otras alternativas.

3.4. BeagleBone Black como MI-CMU

Una placa de bajo coste y *Open-Source Hardware* con muy buenas expectativas es la **BeagleBone Black**, la cual dispone de memoria interna *eMMC* con una distribución *Linux* pre-instalada, concretamente *Debian Wheezy*. Además, para nuestra solución ad-hoc, dispone de una serie de características muy interesantes, como los 7 conversores analógico-digitales y comunicación I2C.

La primera toma de contacto con la placa es para instalar y configurar el servidor de

[16] http://www.redsleeve.org/

Figura 3.11: *BeagleBone Black utilizada. Caja fabricada con impresora 3D.*

pmlib sin ningún dispositivo conectado, y asegurar que es compatible en el sistema preinstalado.

Tras comprobar que todo funciona correctamente, el objetivo pasa a ser el de utilizar los ADC de la placa, aprovechándolos para tomar medidas. Las características de estas entradas analógicas son:

- 12 bits de resolución
- 125ns de tiempo de muestreo
- Rango de medición entre 0 y 1.8V
- 2uA de consumo para cada medición

Podremos utilizar estas entradas para medir por ejemplo las tomas de la tarjeta extensora de *PCIe* de la que hablamos en la sección 2.2.2, teniendo en cuenta que las medidas que nos proporciona son de 1V por amperio, por lo que si el consumo de la *Intel Xeon Phi* en ese canal supera los 1.8A, deberemos de utilizar un divisor resistivo para reducir la tensión de entrada.

El método de lectura de las entradas analógicas de la **BeagleBone Black** se simplifica mucho gracias a que se puede realizar leyendo el contenido de unos ficheros, los cuales son

actualizados por el propio sistema operativo.

Por seguridad y compatibilidad, lo primero que realizamos es una actualización del sistema operativo, haciendo uso del gestor de paquetes que lleva incorporado con la orden *apt-get update*, ejecutada en modo super usuario o modo *root*. Aquí nos encontramos con un fallo que hemos podido solucionar rápidamente buscando información al respecto[17], reemplazando el contenido del fichero *leg_ aging.sh* por el ya corregido.

Los puertos analógicos no se encuentras activados por defecto en la placa, por lo que es algo que hemos debido de tener en cuenta para las pruebas y desarrollos. Para activarlos hay que utilizar la siguiente orden ejecutada como *root*:

echo cape-bone-iio >/sys/devices/bone_capemgr.*/slots

Una vez activados, ya podemos comenzar a realizar mediciones. Basta con leer los ficheros **AIN[0-6]** situados en el directorio */sys/devices/ocp.*/helper.*/* –nótense los asteriscos, los cuales representan un valor numérico y que difiere según la versión del sistema operativo utilizado–.
Los valores que obtenemos estarán representados en milivoltios, por lo que según las especificaciones de la placa, las lecturas deben variar **entre 0 y 1800**.

Con todas las pruebas realizadas, nos planteamos la viabilidad de crear un módulo para **pmlib** que permita acceder a las lecturas de las entradas analógicas de la **BeagleBone Black**.
Debemos tener en cuenta la circunstancia descrita anteriormente acerca de la inactividad inicial de las entradas, por lo que nuestro módulo las activará si fuera necesario.

[17] http://comments.gmane.org/gmane.comp.hardware.beagleboard.user/69174

Finalmente disponemos del módulo **BeableBoneDevice** para **pmlib** que nos permite realizar el muestreo de tensiones en las entradas analógico-digitales de la **BeagleBone Black**.

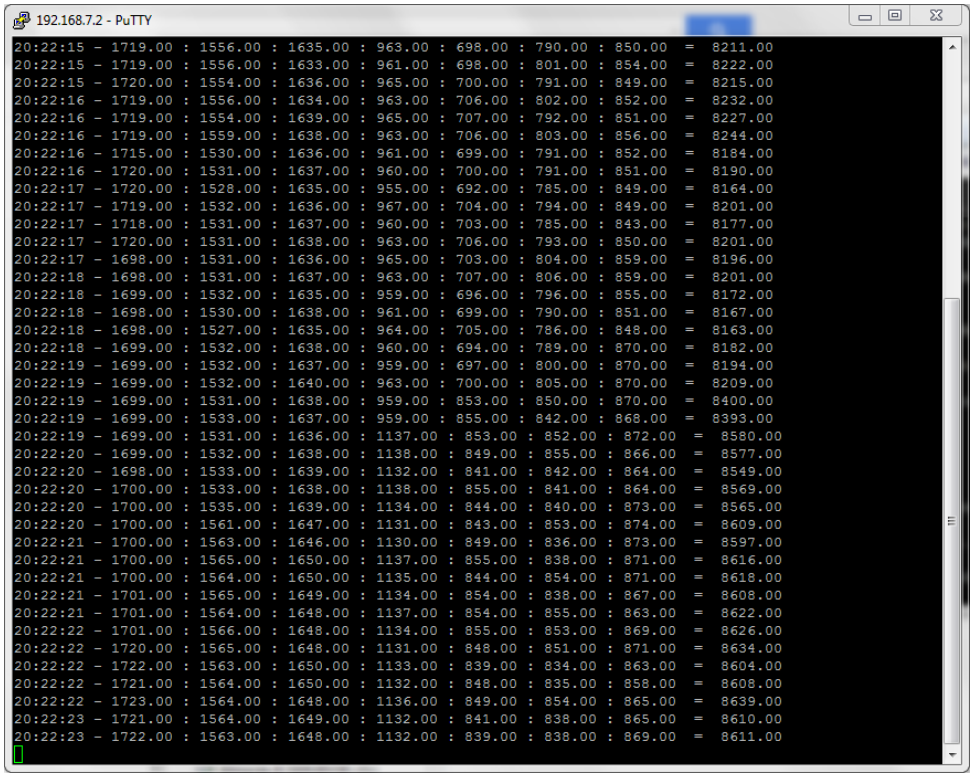


Figura 3.12: *Lectura de las 7 entradas analógicas BeagleBone Black con pmlib.*

Al igual que comentamos en la sección 3.2.3, es necesario instanciar la clase e indicar sobre que líneas se quiere realizar la medición. A continuación explicamos un ejemplo de uso módulo **BeableBoneDevice** implementado:

BBB_msr=BeagleBoneDevice(name="BeagleBoneDevice",computer=ARM,url="", max_frequency=1000)

- *name*: Nombre del dispositivo de medición que se muestra en **pmlib**.
- *computer*: Objeto creado en la configuración con especificaciones del dispositivo anfitrión.

- *url*: Dirección física del dispositivo. En este caso no se utiliza.
- *max_frequency*: Frecuencia máxima de muestreo que queremos que admita **pmlib** para las lecturas en las entradas analógicas.

BBB_msr.add_line(number=0, name="0", voltage=1.8, description="AIN0")

- *number*: Número de línea añadida al dispositivo. Son números correlativos comenzando desde 0 para cada una de las líneas de muestreo.
- *name*: Nombre físico del canal. Para el módulo los valores van desde 0 hasta 6.
- *voltage*: Tensión máxima que soportará la entrada.
- *description*: Breve descripción del canal.

Véase el *Apéndice C* con el código del módulo completo implementado.

3.5. Diseño e implementación de dispositivo de medición (MS) ad-hoc de bajo coste

Llegados a este punto, tenemos alternativas al **CMU** y al **MI**, pero seguimos dependiendo de fuentes externas de medición –**MS**–, en ocasiones con un coste elevado, como es el caso de las pinzas amperimétricas utilizadas en el entorno inicial del **EnaHPC**, concretamente el modelo *Tektronix A622*[18].

3.5.1. Elección del sensor de medición

En ocasiones dispondremos de puntos de medición, como en el caso del extensor de *PCIe PCIeEXT16HOT*, lo cual nos permitirá realizar mediciones simplemente con entradas

[18] http://www.tek.com/datasheet/current-probe/a621-a622

analógicas. En otras ocasiones necesitaremos directamente realizar nosotros esas mediciones en el raíl de alimentación del sistema objetivo de análisis.

Con la premisa de abaratar costes, barajamos diferentes tipos de sensores:

- **LEM HXS 20-NP**: Sensor de corriente AC/DC efecto Hall de hasta 20 Amperios. Coste aproximado de 12€[19]
- **Allegro ACS713**: Sensor de corriente DC de efecto Hall de hasta 30 Amperios. Coste aproximado de 4.50€[20]
- **Analog Devices ADM1191**: Medidor de corriente y tensión desde 3.5V hasta 26V. Precisión de 12 bits y comunicación por I2C. 5.50€[21] aproximadamente por unidad.
- **Texas Instruments INA219**. Medidor de corriente y tensión desde 0V hasta 26V. Precisión de 12 bits y comunicación por I2C. Precio por unidad de 2.35€[22].

Además de las características y costes de cada uno de los sensores, en el caso del **TI INA219** tenemos en cuenta la existencia de un circuito preparado por *Adafruit*, con su correspondiente placa, además de instrucciones de utilización y una librería ya desarrollada. Valorando el precio del sensor y la información disponible optamos por utilizar el **TI INA219**[15].

3.5.2. Adafruit INA219 con Arduino

Podemos encontrar en el mercado un circuito simple creado por *Adafruit* que monta el **TI INA219**[23] que utilizaremos como sensor de medida.

[19] http://es.farnell.com/lem/hxs-20-np/transductor-de-corriente/dp/9135596

[20] http://es.farnell.com/allegro-microsystems/acs713elctr-30a-t/ic-hall-effect-current-sensor/dp/1651977

[21] http://es.farnell.com/analog-devices/adm1191-2armz-r7/monitor-potencia-dgtl-alerta-10msop/dp/2074889

[22] http://es.farnell.com/texas-instruments/ina219aidcnr/current-power-monitor-120db-8sot23/dp/2295989

[23] https://www.adafruit.com/products/904

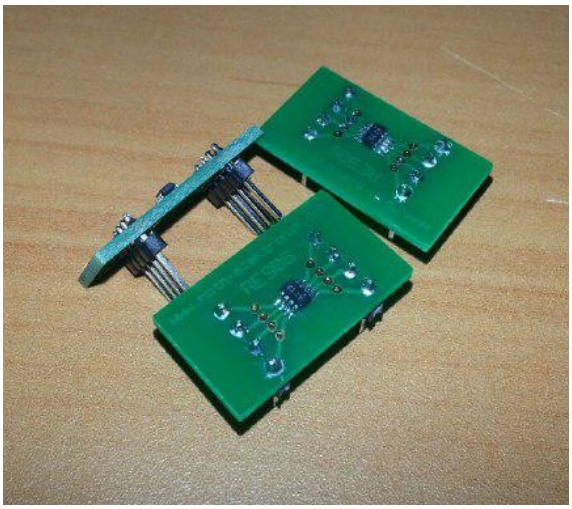


Figura 3.13: *Sensores INA219 soldados en PCB utilizados para pruebas.*

El montaje consiste en:

- Resistencia de *shunt* 0.1 Ohm y 1 % de precisión.
- Resistencias *pull-up* para SDA y SCL.
- Resistencias de direccionamiento.
- Condensador de desacoplo.

Según la hoja de especificaciones, el **TI INA219** admite hasta 16 direcciones distintas, pero la placa diseñada por *Adafruit* únicamente está preparada para soportar 4 direcciones diferentes; por defecto la dirección que está configurada es *0x40*.

Está a nuestra disposición un montaje con un *Arduino Uno*, pero utilizamos el **Arduino Duemilanove**, de similares características, para nuestras primeras pruebas.

Adafruit pone a nuestra disposición una librería con ejemplos de *Arduino*, por lo que para poder realizar las primeras pruebas unicamente se necesita obtener el *IDE* del propio *Arduino*[24], descargar la librería para poder compilar el ejemplo y a continuación cargarlo en

[24] http://www.arduino.cc/en/Main/Software

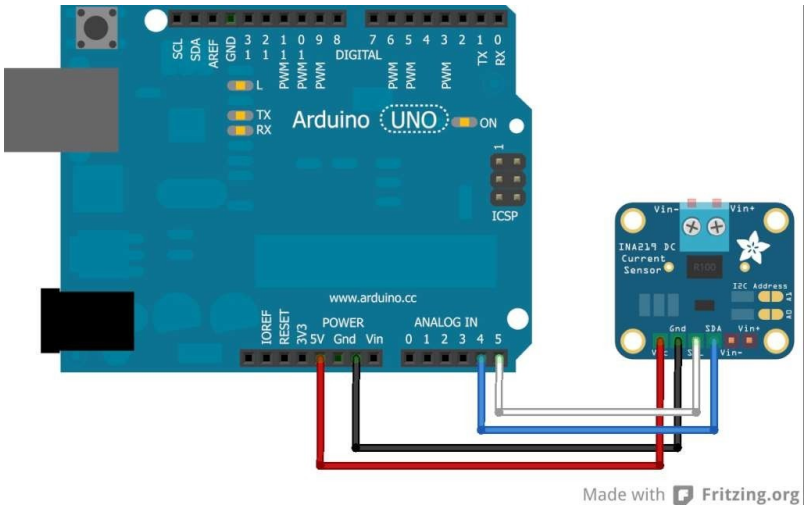


Figura 3.14: *Esquema de montaje Adafruit INA219 en Arduino Uno.*

la placa.

La librería la podemos descargar desde GitHub [25].

Realizado todo el proceso, podemos conectarnos por comunicación serie con el *Arduino Duemilanove* y obtenemos los primeros resultados, satisfactorios en cuanto a consumo, pero no así en lo que a tensiones se refiere. Posteriormente nos damos cuenta de que en el montaje nos falta unir las tierras, por lo que solucionado éste inconveniente, los resultados son correctos.

3.5.3. Comunicación con INA219 desde BeagleBone Black

Comprobada la comunicación con un Arduino, intentamos realizarla también con la **BeagleBone Black** a través de su bus I2C. Aquí hay que remarcar que pese a que en el *pinout* de la placa[26] se indique que los pines 19 y 20, correspondientes a *SCL* y *SDA* respectivamente, pertenecen al bus número 2 de la **BeagleBone Black**, en realidad pertenecen al bus 1, al menos en lo que a acceso se refiere, y así lo hemos podido comprobar y verificar.

[25] https://github.com/adafruit/Adafruit_INA219

[26] https://insigntech.files.wordpress.com/2013/09/bbb_pinouts.jpg

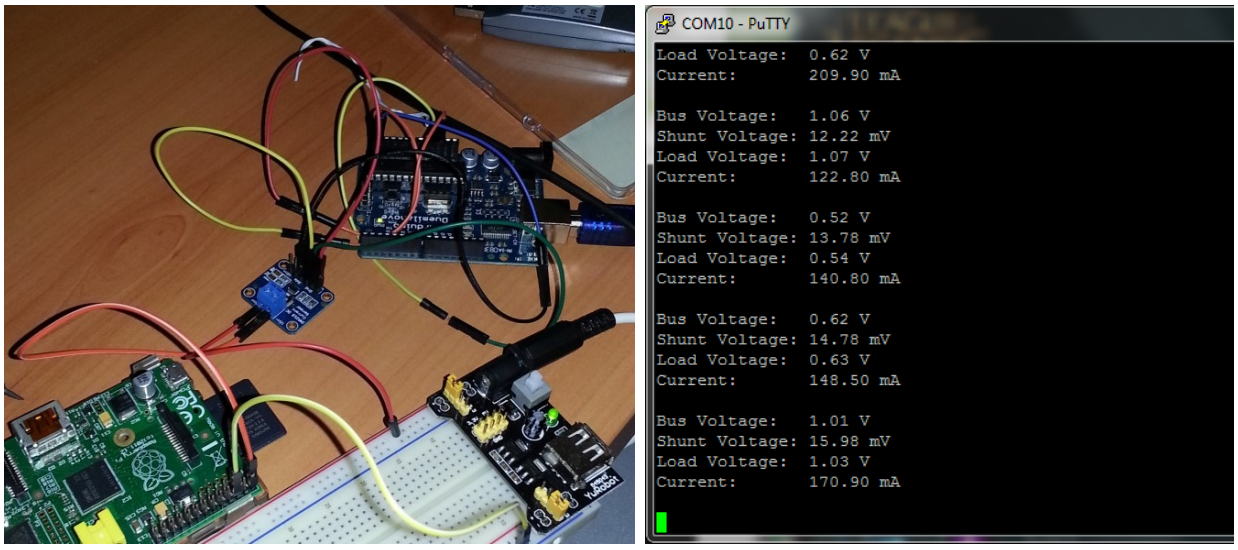


Figura 3.15: *Montaje –izquierda– y mediciones –derecha– Adafruit INA219 con Arduino Duemilanove.*

Para simplificar el acceso por I2C, *Adafruit* dispone de una librería para este tipo de comunicación desarrollada en *Python*: **Adafruit_I2C**[27].
También hay disponible una librería para Raspberry Pi® [28] que nos sirve de base para desarrollar nuestra propia librería con soporte para **BeagleBone Black**.

Con el fin de comprobar el correcto funcionamiento del sensor con la **BeagleBone Black**, realizamos un montaje sencillo colocando una pequeña carga. En primer lugar verificamos si el dispositivo es detectado en el bus I2C_1 con la dirección *0x40*; para ello ejecutamos el comando ***i2cdetect -y -r 1*** –el número *1* indica el bus– como super usuario y comprobamos que efectivamente está conectado.
Por último realizamos una serie de mediciones haciendo uso de las dos librerías en un *script* de prueba en *Python*, y comprobamos que todo funciona como se espera.

[27] https://github.com/adafruit/Adafruit-Raspberry-Pi-Python-Code/blob/master/Adafruit_I2C/Adafruit_I2C.py
[28] https://github.com/scottjw/subfact_pi_ina219

3.5.4. Módulo pmlib para TI INA219

Aprovechando las dos librerías desarrolladas en *Python* para la captura de datos de los sensores **TI INA219**, realizamos un módulo para **pmlib** que nos permita comunicarnos con varios sensores y poder así capturar mediciones de varios canales simultáneamente.
La flexibilidad del módulo nos permite añadir tantos dispositivos de medición como direcciones nos permita el propio sensor, es decir, hasta 16.

El integrado dispone de un registro de configuración en la dirección *0x00*. Para calcular los valores necesarios en nuestro entorno, hacemos uso de la aplicación de *Texas Instruments* **INA219 EVM**, tal y como se observa en la figura 3.16.

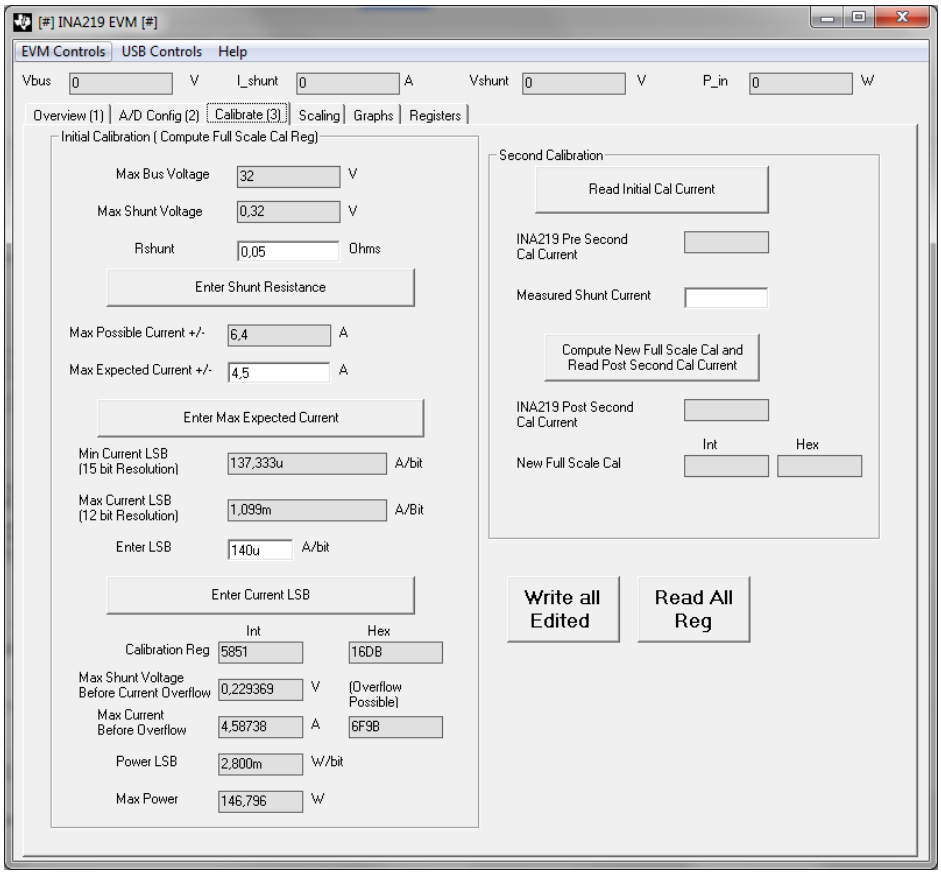


Figura 3.16: *Pantalla de Calibración INA219 EVM para nuestro entorno.*

Las características de nuestro entorno requiere que el máximo de corriente que pueda medir un **TI INA219** sea de 4,5A al menos. Con esta premisa, utilizaremos una resistencia de *shunt* con un valor de *0,05Ohm*, la cual disipará como máximo *1,0125 vatios*, por lo que deberemos elegir un componente adecuado teniendo en cuenta esta circunstancia. De este modo, el valor que deberemos configurar en el **registro de calibración** es de 5851, como se puede observar en la figura 3.16.
Además, debemos calcular los divisores, teniendo en cuenta los valores del **LSB**[29] **de corriente** y el **LSB de potencia**, cuyos valores son **140 microamperios/bit** y **2,8 vatios/bit** respectivamente.

La resolución de los sensores configurada por defecto en la librería *INA219* es de 12 bits, lo cual supone, según la hoja de especificaciones[15], un tiempo de lectura en el peor de los casos de 512us –constante *__INA219_CONFIG_SADCRES_12BIT_1S_532US*–. Si fuera necesario aumentar la frecuencia de muestreo, se puede disminuir la resolución.

Una vez más, y como ocurre con los módulos desarrollados y explicados en las secciones 3.2.3 y 3.4, es necesario instanciar la clase e indicar qué direcciones corresponden a cada uno de los sensores **TI INA219**. A continuación explicamos un ejemplo de uso módulo **INA219Device** –véase *Apéndice D*–:

INA219_msr= INA219Device(name=ÏNA219Device", computer=ARM, url="Current", max_frequency= 1000)

- *name*: Nombre del dispositivo de medición que se muestra en **pmlib**.
- *computer*: Objeto creado en la configuración con especificaciones del dispositivo anfitrión.

[29] **LSB**: Least significant bit, o bit menos significativo

- *url*: Tipo de medición deseada: tensión, corriente o potencia consumida. Los tres posibles valores son: "Voltage", Çurrentz "Power", siendo éste último el valor por defecto si el indicado no es válido.

- *max_frequency*: Frecuencia máxima de muestreo que queremos que admita **pmlib** para las lecturas en el conjunto de todos los sensores conectados.

INA219_msr.add_line(number=0, name="0x40", voltage=12, description=ÏNA Direccion 40")

- *number*: Número de línea añadida al dispositivo. Son números correlativos, comenzando desde 0, para cada uno de los sensores conectados.

- *name*: Dirección I2C del sensor.

- *voltage*: Tensión máxima que soportará la entrada del sensor.

- *description*: Breve descripción del canal.

A diferencia de otros módulos desarrollados, gracias a las capacidades del **TI INA219**, podemos realizar mediciones de tensión del bus (voltios), corriente (miliamperios) y potencia (milivatios), simplemente indicándolo en la configuración.

La combinación del módulo **BeagleBoneDevice** y **INA219Device** nos dan una capacidad de muestreo de hasta 23 raíles, 7 mediante los ADC de la **BeagleBone Black** y 16 utilizando **TI INA219**.

3.5.5. Diseño de una placa de expansión para BeagleBone Black (*CAPE*)como solución completa

Habiendo ya seleccionado la **CMU**, **MI** y **MS**, nos disponemos a realizar la integración completa en un único dispositivo de bajo coste.

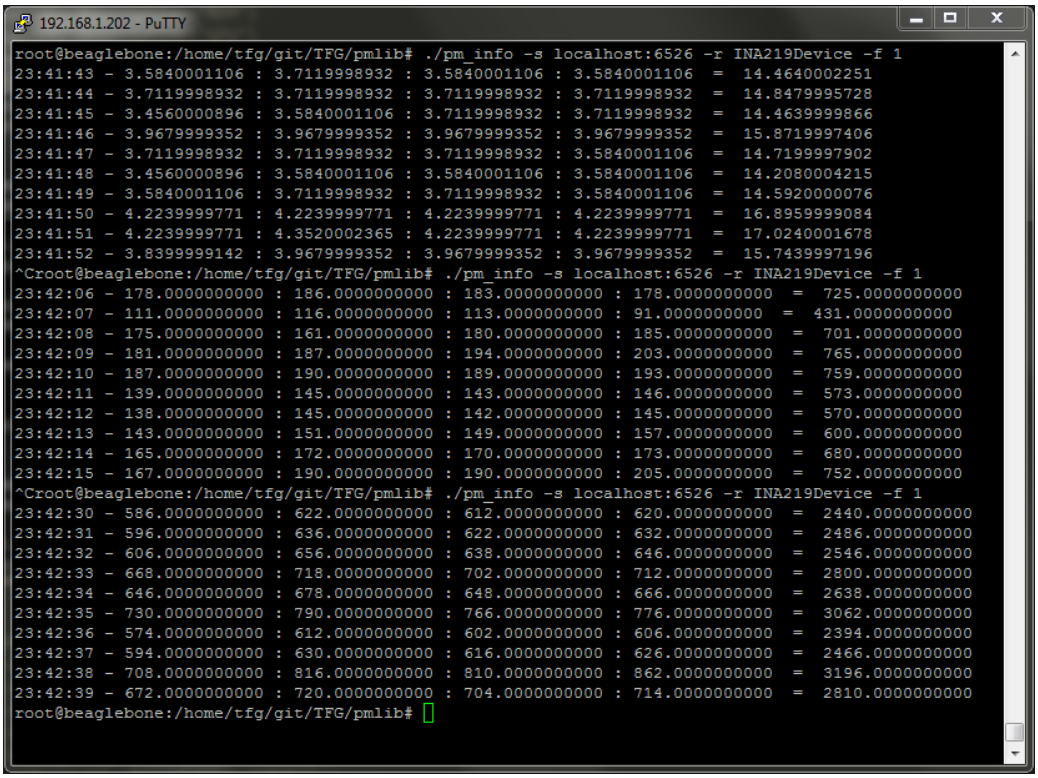


Figura 3.17: *Medidas de tensión, corriente y potencia mediante el módulo INA219Device.*

En primer lugar realizamos pruebas sobre la placa de prototipado, con el fin de verificar el diseño, para posteriormente diseñar el *PCB* y fabricarlo.

Debemos tener en cuenta que la precisión del montaje sobre placas de prototipado se ve mermada por las características del propio montaje en este tipo de entornos.

Finalmente, tras la integración en **pmlib** de los módulos **BeagleBoneDevice** y **INA219Device**, realizamos una prueba de captura de medidas, para comprobar que el total del conjunto funciona correctamente.

Como explicamos en la sección 2.1, la *Intel Xeon Phi* requiere de 4 raíles de alimentación, 2 mediante *PCIe* y otros dos a través de una fuente externa.

Gracias a la tarjeta extensora *PCIeEXT16HOT* disponemos de dos puntos de medición que conectaremos a las entradas analógicas de la *BeagleBone Black*. Estos puntos de medi-

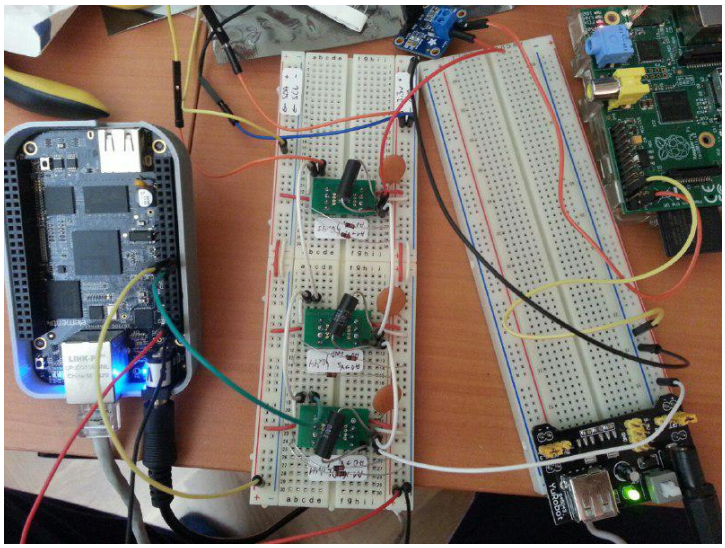


Figura 3.18: *Montaje de TI INA219 con BeagleBone Black.*

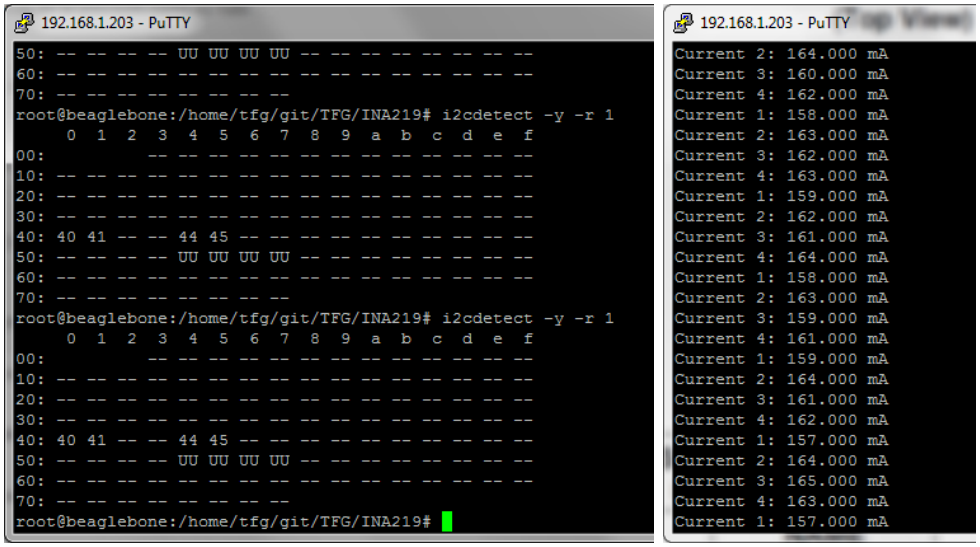


Figura 3.19: *Direcciones y mediciones de 4 TI INA219 con BeagleBone Black.*

ción nos dan una tensión de 1 voltio por cada amperio de consumo, lo cual nos limita hasta un máximo de 6.5 voltios el punto de medición. Debemos recordar que las entradas ADC de la placa admiten una tensión de entrada máxima de 1.8V, por lo que deberemos utilizar un divisor resistivo de tal modo que con una tensión de 6.5 voltios de entrada, no supere en su salida el límite del conversor.

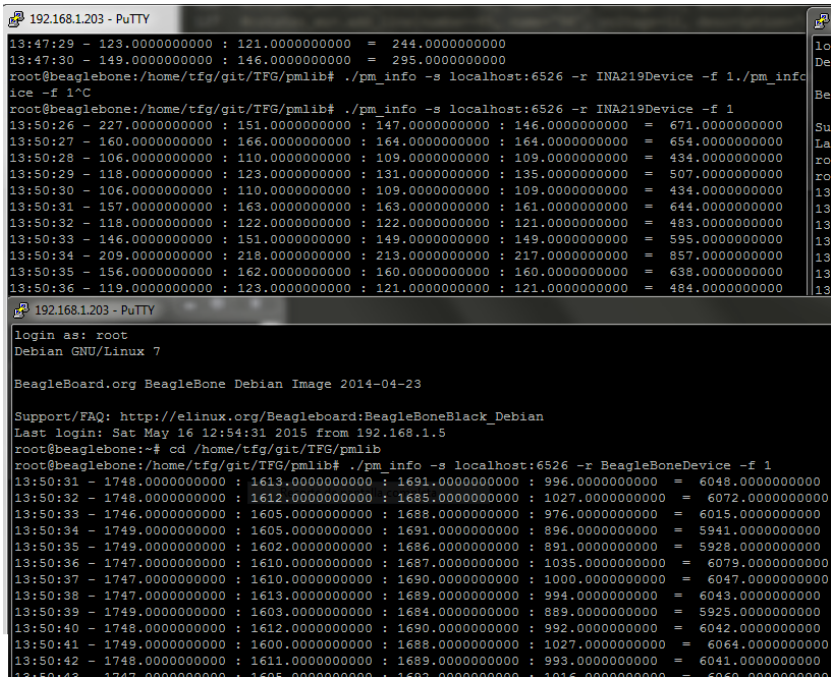


Figura 3.20: *Captura de datos simultánea de sensores TI INA219 y ADC de la BeagleBone Black.*

Por otro lado, cada uno de los dos raíles de alimentación externos de la *Intel Xeon Phi* consta a su vez de diferentes vías y cables. Uno de los conectores dispone de 4 vías –8 cables–, de las cuales se monitorizarán 3 –uno es directo–, y el otro 3 vías (6 cables). Hacen un total de 6 vías de las que es necesario tomar medidas, lo que obliga a que nuestra solución disponga de al menos 6 sensores *TI INA219*.

En definitiva, para la monitorización del coprocesador *Intel Xeon Phi* necesitamos un total de 2 ADC y 6 *TI INA219*.

Para el diseño propuesto, vamos a determinar las direcciones para los sensores, en las que indicamos las conexiones:

- *0x40*: GND (A1) - GND (A0)
- *0x41*: GND (A1) - Vs (A0)

- *0x42*: GND (A1) - SDA (A0)
- *0x43*: GND (A1) - SCL (A0)
- *0x44*: Vs (A1) - GND (A0)
- *0x45*: Vs (A1) - Vs (A0)

También consideramos la opción de aprovechar todas las entradas analógicas de la *BeagleBone Black*, debido a que apenas requiere de componentes pasivos y creemos que no habría problemas de espacio.

Tras todas estas consideraciones, diseñamos el circuito del *CAPE*[30] para la *BeagleBone Black*. Véanse los *Apéndices E* y *F* con el esquemático y la disposición de componentes respectivamente.

[30] **CAPE**: Circuito impreso que se puede conectar en la parte superior de la placa principal, permitiendo añadir funcionalidad extra a la misma.

Capítulo 4

Aplicación y validación del entorno experimental

El objetivo de la presente sección es doble. En primer lugar, ilustrar un ejemplo de aplicación del entorno experimental de medición de consumo desarrollado sobre un código real; en segundo lugar, validar la corrección de las mediciones obtenidas desde el entorno de medición, comparándolas con las mediciones de consumo ofrecidas por el fabricante de una determinada arquitectura.

4.1. Aplicación objetivo

Con el fin de evaluar la utilidad y corrección del entorno experimental de medición de consumo, se ha desarrollado una aplicación heterogénea sobre una arquitectura híbrida equipada con dos tarjetas aceleradoras Intel Xeon Phi de última generación. La aplicación desarrollada realiza un producto de matrices utilizando aritmética de doble precisión a través de invocaciones a la rutina correspondiente de la biblioteca Intel MKL (*Math Kernel Library*); dicha biblioteca posee la capacidad de distribuir automáticamente el proceso general de multiplicación entre los recursos de computación disponibles (en este caso, CPU multi-core y cada una de las dos Intel Xeon Phi), lo cual nos permitirá evaluar tanto el rendimiento como el consumo de las tres unidades de procesamiento operando en paralelo. El objetivo final será obtener perfiles detallados de consumo energético de la aplicación a través

del entorno de medición descrito y, alternativamente, integrar dichos perfiles con trazas de rendimiento, de modo que sea posible extraer conclusiones acerca de la eficiencia energética de la aplicación.

Aunque el análisis detallado del perfil de consumo de la aplicación sobre la arquitectura descrita queda fuera del alcance del presente trabajo, ilustraremos el tipo de información que puede extraerse utilizando el entorno experimental descrito.

4.2. Instrumentación y monitorización de Intel Xeon Phi con DAQ NI USB-6218 y pmlib

Nótese que, a diferencia de la arquitectura básica descrita en la Sección 2.1, en este experimento evaluamos una arquitectura con *dos* aceleradores Intel Xeon Phi, lo que requerirá instrumentación adicional de medida (básicamente, un extensor PCIexpress adicional, junto con una pinza amperimétrica adicional) para obtener datos de consumo desde ambos aceleradores. Además, utilizaremos una tercera pinza que monitorizará las líneas de 12V que alimentan la placa base, y que permitirán evaluar el consumo energético del procesador de propósito general.

Desde el punto de vista software, la instrumentación de un determinado código utilizando la API de `pmlib` se reduce a añadir un reducido número de invocaciones a la biblioteca `pmlib`; dichas funciones encapsulan la lógica de comunicación con el servidor, la selección de canales a consultar y frecuencia de muestreo, o el formato de almacenamiento de las trazas. A modo de ejemplo, el código que se muestra en el Listado 4.1 realizaría el perfilado de una invocación a la rutina `cblas_dgemm` (producto de matrices), monitorizando las líneas 0 y 7 del DAQ con una frecuencia de 400 muestras por segundo, obteniendo los resultados en formato CSV.

Listing 4.1: *Instrumentación de código a través de pmlib para adquisición de datos mediante DAQ NI USB-6218*

```
1 #include "pmlib.h"
2
3 int main( void )
4 {
```

```
5  // ...
6
7  server_t servidor;
8  counter_t contador;
9  counter_t contador2;
10 line_t lineas;
11 device_t disp;
12
13        int frequency= 400;
14        int aggregate= 1;
15
16 LINE_CLR_ALL(&lineas);
17
18 // Seleccion de lineas a monitorizar.
19 LINE_SET( 0, &lineas );
20 LINE_SET( 7, &lineas );
21
22 // ...
23 // Creacion de un contador asociado al DAQ.
24 pm_create_counter("DAQmxDevice", lineas, !aggregate,
25   frequency, servidor, &contador);
26
27 // Inicio de un contador asociado al DAQ.
28 pm_start_counter(&contador);
29
30 //// Inicio codigo a perfilar.
31 cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
32   N, M, K, alpha, A, K, B, M, beta, Ccblas, M);
33 //// Fin codigo a perfilar.
34
35 // Finalizacion de un contador asociado al DAQ.
36 pm_stop_counter(&contador);
37
38 // Obtencion de datos de perfil.
39 pm_get_counter_data(&contador);
40
41 // Seleccion de formato (CSV).
42 pm_print_data_csv( "perfil.out", contador, lineas, -1);
43
44 // Finalizacion de contador.
45 pm_finalize_counter(&contador);
46
47 // ...
48 }
```

En nuestro caso, dado que deseamos monitorizar el consumo detallado tanto de la CPU como de cada una de las dos tarjetas Intel Xeon Phi, se han monitorizado los siguientes siete canales del DAQ:

1. Xeon Phi 1. Extensor PCIe. Línea 3.3 V (Canal A9).

2. Xeon Phi 1. Extensor PCIe. Línea 12 V (Canal A1).

3. Xeon Phi 1. Extensor PCIe. Alimentación externa (Canal A18).

4. Xeon Phi 2. Extensor PCIe. Línea 3.3 V (Canal A10).

5. Xeon Phi 2. Extensor PCIe. Línea 12 V (Canal A2).

6. Xeon Phi 2. Extensor PCIe. Alimentación externa (Canal A25)

7. Alimentación placa base. Línea 12 V (Canal A17).

4.3. Resultados obtenidos

4.3.1. Ejemplo ilustrativo

Las Figuras 4.1, 4.2 y 4.3 muestran tres ejemplos de trazas de consumo energético obtenidas a partir del código anteriormente descrito e instrumentado. De nuevo, cabe destacar que el objetivo de la presente sección no es realizar un estudio detallado del perfil de consumo de una determinada combinación aplicación/arquitectura, sino ilustrar el tipo de mediciones y nivel de detalle obtenido a partir del entorno desarrollado. En los experimentos instrumentados, el reparto de carga de trabajo entre el procesador de propósito general y las tarjetas aceleradoras es fijo: en la Figura 4.1, la totalidad de la computación se realiza en el procesador de propósito general; en la Figura 4.2, el trabajo se reparte equitativamente entre el procesador de propósito general (50 %) y las tarjetas aceleradoras (25 % por cada una). en la Figura 4.3, la totalidad de la computación se realiza en los Intel Xeon Phi.

En cada figura, la gráfica superior muestra la potencia instantánea a través de cada uno de los canales analizados, mientras que la gráfica inferior agrupa dichas lecturas en función de su origen (*cores*/procesador de propósito general y aceleradores).

Resultan claramente identificables los picos de consumo producidos por la carga de trabajo asignada a cada tipo de procesador, así como la reducción en tiempo ante el uso de una distribución u otra de trabajo. Aunque no se muestra en las figuras, la integración de este tipo de trazas con trazas de rendimiento resulta trivial gracias a la integración con `pmlib`, lo que permite, con gran nivel de detalle, obtener perfiles no sólo de la aplicación completa, sino también de las distintas fases que la componen.

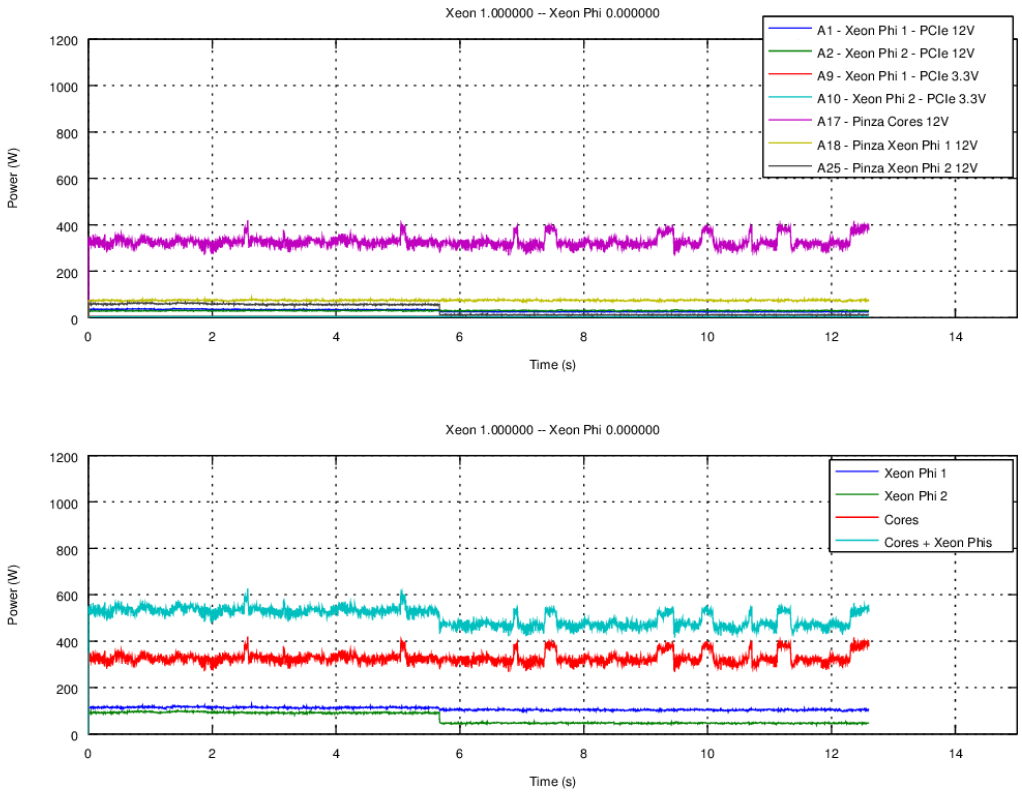


Figura 4.1: *Perfil de consumo energético para el producto de matrices. Ratio Xeon Phi/-Xeon: 0.0.*

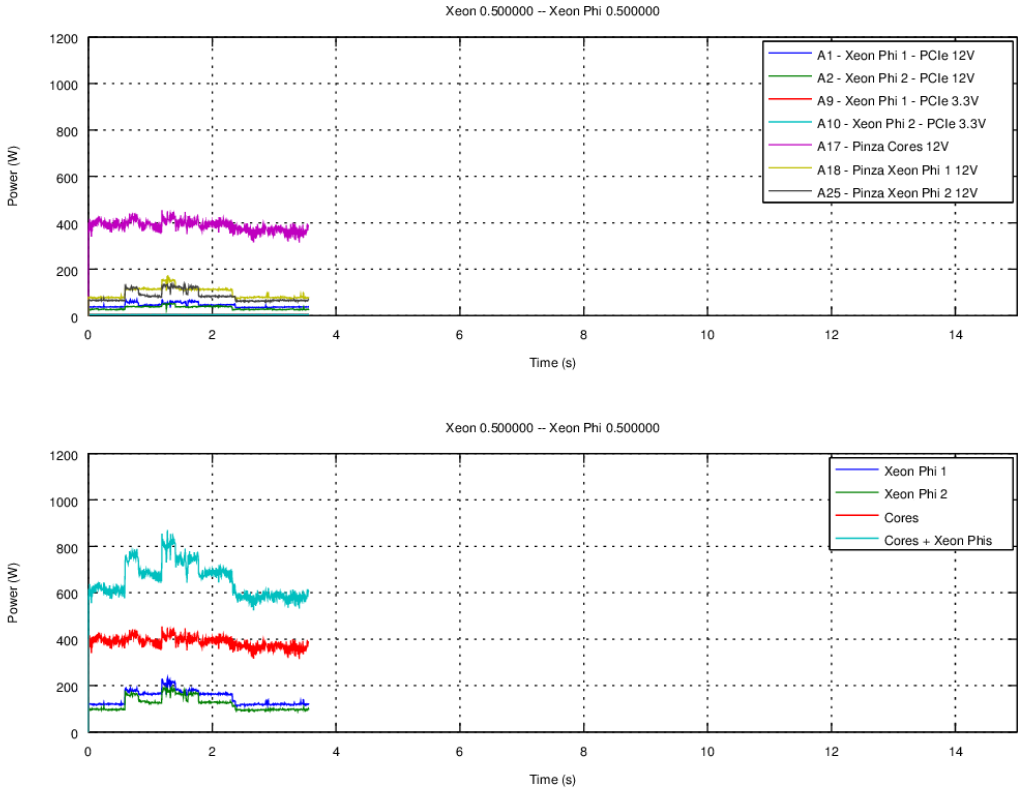


Figura 4.2: *Perfil de consumo energético para el producto de matrices. Ratio Xeon Phi/-Xeon: 0.5.*

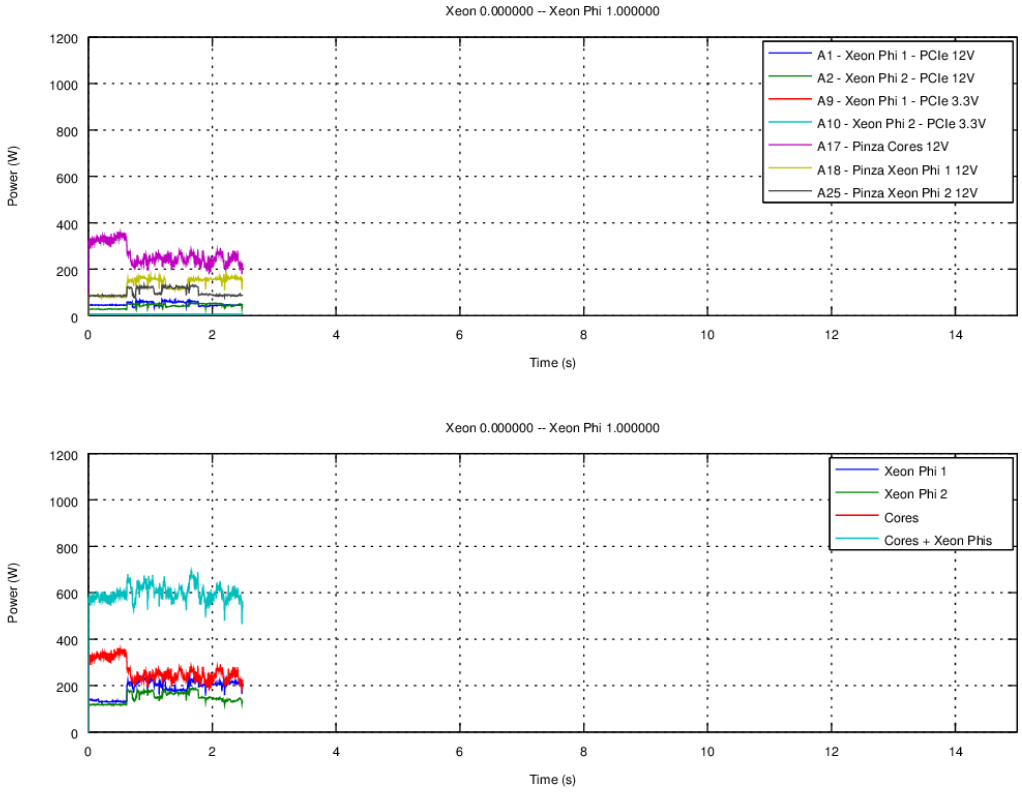


Figura 4.3: *Perfil de consumo energético para el producto de matrices. Ratio Xeon Phi/-Xeon: 1.0.*

4.3.2. Validación de resultados

Como último resultado experimental producto de la utilización del entorno desarrollado, se ilustra a continuación una comparativa entre las medidas de potencia obtenidas por el entorno experimental utilizando un DAQ NI USB-6218 actuando como CMU en conjunto con `pmlib`, y las medidas ofrecidas a través de la biblioteca MicAccess distribuida por Intel. El objetivo de esta comparativa es doble: por un lado, observar posibles desviaciones graves en los perfiles de potencia obtenida, producto por ejemplo de una mala calibración del CMU o un error en el diseño o montaje del entorno, y por otro observar la calidad de la sincronización de las muestras obtenidas a través de `pmlib`.

En este caso, el experimento desarrollado se centra en ejecutar un producto de matrices de dimensión 10240 sobre únicamente una tarjeta aceleradora Intel Xeon Phi, repitiendo dicho experimento de forma cíclica durante diez iteraciones. La Figura 4.4 muestra los perfiles de consumo obtenidos por nuestro entorno experimental (gráfica superior) y por la herramienta ofrecida por Intel (gráfica inferior). Cabe destacar que, para favorecer la legibilidad, únicamente ilustramos los valores obtenidos por ambos entornos para la alimentación recibida por la tarjeta a través del bus PCIe, aunque se han observado similares resultados para el resto de fuentes de alimentación.

Como se puede observar en la figura, tanto la sincronización de las trazas obtenidas como el perfil de potencia es muy similar en ambos casos; atendiendo a este último factor, la desviación en términos de potencia media durante toda la ejecución es menor al 3 %. Cabe destacar las ventajas de utilizar nuestro entorno experimental: la frecuencia es mucho mayor (en este caso, la frecuencia utilizada ha sido de 400 muestras por segundo, mientras que el entorno software de Intel únicamente ofrece 20 muestras por segundo); además, en el caso de no disponer de una herramienta software similar, un entorno de medición externo es la única forma de realizar medidas de consumo precisas y de gran resolución.

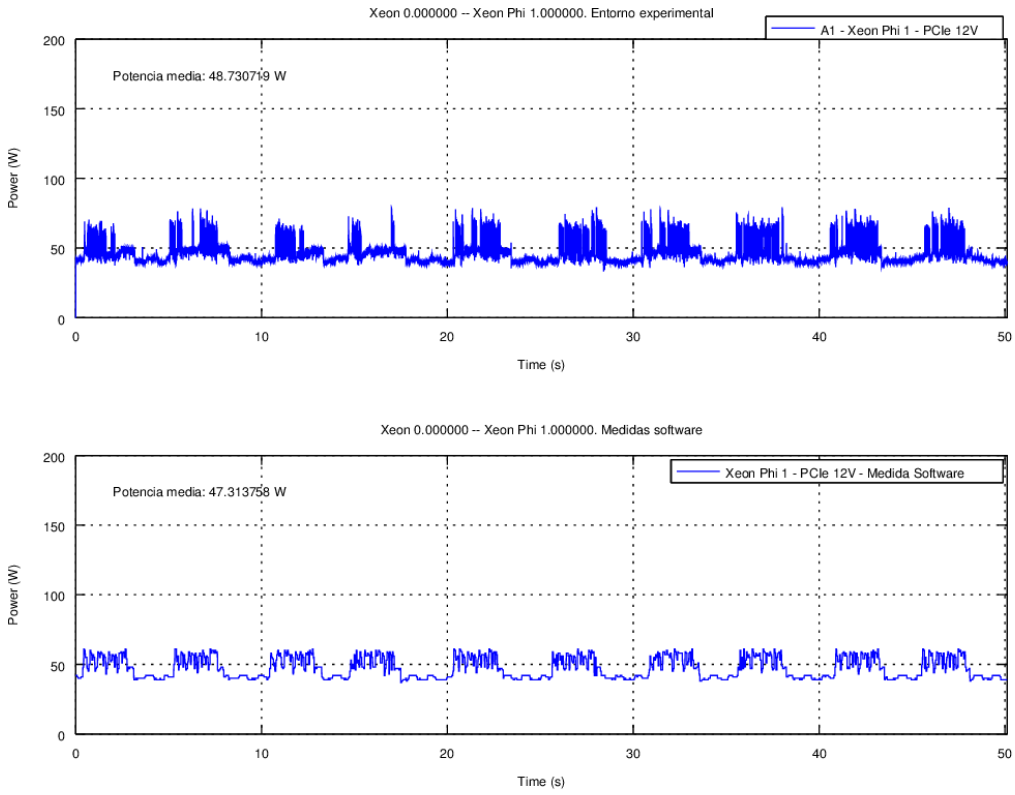


Figura 4.4: *Perfil de consumo energético para el producto de matrices. Ratio Xeon Phi/-Xeon: 1.0.*

Capítulo 5

Conclusiones

Durante la evolución del proyecto podemos considerar dos hitos principales, en los cuales definimos dos entornos diferenciados, con distintas características y posibilidades pero cumpliendo ambos los objetivos del presente trabajo.

5.1. Sustitución del Agilent DC Power Analyzer N6705B como CMU por NI DAQ USB

Uno de los objetivos principales ha sido el de poder sustituir el costoso equipo de control de mediciones *Agilent DC Power Analyzer N6705B*. Este reemplazo, ha supuesto, además, un aumento de rendimiento en otros aspectos.

- **Número de canales**: El *NI USB-6218* ofrece la posibilidad de muestrear hasta 32 canales de manera simultánea, frente a los 4 canales del *N6705B*. En el caso del *NI USB-6009*, el número de canales que permite medir es de 8, por lo que en este caso también mejoramos la capacidad de canales a monitorizar.

- **Frecuencia de muestreo**: en esta ocasión también es muy significativa la capacidad de muestreo del DAQ frente al equipo sustituido. En el caso del *NI USB-6218* disponemos de una frecuencia de muestreo de hasta 250.000 muestras por segundo frente a las 100.000 del equipo reemplazado. Con el *NI USB-6009* la frecuencia disminuye, y no supera en este caso al *N6705B*; en cualquier caso dispone de una velocidad de

muestreo suficiente y válida para la gran mayoría de entornos: 48.000 muestras por segundo.

- **Coste del dispositivo**: La diferencia más significativa es la diferencia de coste entre unos equipos de medida y otros.
  Si comparamos el *N6705B* –5686€[1]– con el *NI USB-6218* –1490€[2]–, obtenemos un ahorro de prácticamente un 75 % al utilizar el segundo, o lo que es lo mismo, tenemos un gasto de $\frac{1}{4}$ en comparación con el dispositivo del entorno inicial.
  Al comparar el *N6705B* con el otro dispositivo de *National Instruments* empleado, el *NI USB-6009* –315€[3]–, el ahorro es mucho más importante, llegando casi hasta el 95 %, o dicho de otra forma, reduciendo el coste inicial hasta $\frac{1}{18}$.

5.2. Solución BeableBone Black con CAPE a medida como conjunto MI-CMU-MS de bajo coste

Finalmente, tras las diversas alternativas probadas, llegamos a un sistema que nos permite monitorizar prácticamente cualquier tipo de entorno. Además, la opción elegida de *BeagleBone Black* con un *CAPE* reduce considerable los costes y nos da una gran versatilidad, siendo posibles modificaciones de una manera sencilla.

Son muchas las ventajas que podemos conseguir gracias al entorno aquí planteado:

- **Número de canales**: Al tratarse de un diseño personalizado, hemos optado por la creación de un dispositivo capaz de monitorizar las 4 vías de alimentación de la *Intel Xeon Phi* del entorno de partida del proyecto –ver sección 2.1 para más información–. Según lo explicado en la sección 3.5.5, en total necesitamos tomar medidas de 8 vías, pero aprovechando el espacio y las posibilidades de los conversores analógico-digitales

[1] http://es.rs-online.com/web/p/power-quality-analysers/7110012/
[2] http://sine.ni.com/nips/cds/view/p/lang/es/nid/203484
[3] http://sine.ni.com/nips/cds/view/p/lang/es/nid/201987

de la *BeagleBone Black*, hemos optado por incluir 3 entradas de medición más, haciendo un total de 11 –6 *TI INA219* y 5 ADC de la *BeagleBone Black*– los canales de los que podemos capturar información.

Recordar que la *BeagleBone Black* dispone de **7 ADC**, los cuales sumados a los posibles **16** *TI INA219* que podemos conectar en una misma red de datos, harían un total de **23 líneas objetivo de análisis**. Si fuera necesario, podríamos ampliar aún más el número de dispositivos *TI INA219* realizando multiplexación, lo que nos da unas posibilidades que difícilmente nos brinda otro tipo de dispositivo.

Si todo ello lo comparamos con los 4 canales que podíamos monitorizar con el *N6705B* en el entorno inicial, vemos que es una opción muy versátil.

- **Menor coste**: como ocurre con la alternativa de la sección 5.1, el coste se ve drásticamente reducido. El coste de nuestra plataforma de base, la BeagleBone Black, apenas supera los 50€[4] de coste. Si a ello le sumamos los componentes utilizados en el *CAPE* desarrollado durante el presente trabajo, en conjunto no superamos la cifra de 150€–precio aproximado del prototipo, incluyendo fabricación de la *PCB*–, apenas $\frac{1}{38}$ –o poco más de 2,5 %– del coste del *N6705B*. Estos costes se podrían reducir si se realiza una producción mayor de placas del circuito impreso.

- **No requiere pinzas amperimétricas**: gracias a que el *TI INA219* nos permite monitorizar directamente la corriente que circula por el raíl, no necesitamos el uso de pinzas amperimétricas. En nuestro entorno inicial se hacen uso de dos de ellas para ambas fuentes de alimentación externa, por lo que reduciríamos el coste en algo más de más de 1300€.

- **Medición de voltaje**: otra ventaja de los *TI INA219* es que permite la medición de la tensión de las líneas de alimentación. Estos datos nos dan la posibilidad de ajustar el consumo ya que **no partimos de la premisa de una tensión constante**.

[4] http://es.rs-online.com/web/p/kits-de-desarrollo-de-procesador-y-microcontrolador/7753805/

- **Potencia instantánea**: una de las grandes características que podemos destacar de los integrados *TI INA219* es la medición de la potencia consumida en un instante de tiempo. Como hemos mencionado anteriormente, ya nos permitía la captura de tensión y corriente, por lo que en principio podría parecer innecesaria esta prestación. En nuestro caso es realmente útil, ya que para leer la tensión y la corriente necesitaríamos realizar dos peticiones mediante I2C al integrado, lo cual conlleva un retardo, y esto supondría que la medida se desvirtuara, algo que no ocurre si el integrado nos devuelve ya la potencia calculada en un momento concreto.

Por otro lado, también nos hemos encontrado con algún tipo de desventaja. Para el funcionamiento del *TI INA219* es necesaria una resistencia de *shunt* de precisión, con un valor realmente bajo –0,05 Ohm en nuestro montaje–, pero que supone una pequeña distorsión en el canal medido. Pese a que esta variación apenas supone un 0,01 % de la medición, en caso de ser necesario, se puede calcular el consumo de la propia resistencia y corregirlo posteriormente en nuestro **MI** –sección 2.2.3–.
Otra inconveniente es el de la limitación de los canales ADC, ya que los mismos sólo admiten 1.8v, y además sólo monitorizan tensión. Aún así, modificando la configuración del divisor resistivo podríamos variar la capacidad de la tensión que podemos medir.

Siguiendo la filosofía *Open Source Hardware*, en la se apoya el presente trabajo con la utilización de la *BeagleBone Black*, hemos puesto a disposición de toda la Comunidad los esquemáticos y fuentes que permitan a cualquier persona modificar el desarrollo inicial según sus necesidades. Para ello, el proyecto ha sido publicado en *Open Hardware Repository*, con el nombre de **Fine grain powermeter for HPC accelerators**[5] y bajo una licencia CERN OHL v1.2[6].

[5] http://www.ohwr.org/projects/cape-pwr-12b-6ch
[6] http://www.ohwr.org/attachments/2388/cern_ohl_v_1_2.txt

Bibliografía

[1] Francisco D Igual, Luis M Jara, José I Gómez-Pérez, Luis Piñuel, and Manuel Prieto-Matías. A power measurement environment for pcie accelerators. *Computer Science-Research and Development*, pages 1–10, 2014.

[2] Wikipedia - Open-Source Hardware. 2015. http://en.wikipedia.org/wiki/Open-source_hardware.

[3] Wikipedia - Plug and Play. 2015. http://en.wikipedia.org/wiki/Plug_and_play.

[4] S. Barrachina, M. Barreda, S. Catalán, M. F. Dolz, G. Fabregat, R. Mayo, and E. S. Quintana-Ortí. An integrated framework for power-performance analysis of parallel scientific workloads. *3rd Int. Conf. on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 114–119, 2013.

[5] Intel Corporation. *Intel Xeon Phi Coprocessor*, June 2015. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[6] Yakun Sophia Shao and David Brooks. Energy characterization and instruction-level energy model of intel's xeon phi processor. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 389–394. IEEE Press, 2013.

[7] Wikipedia - Hall Effect. 2015. http://en.wikipedia.org/wiki/Hall_effect.

[8] IEEE Instrumentation and Measurement Society. *488.2-1992 - IEEE Standard Codes, Formats, Protocols, and Common Commands for Use With IEEE Std 488.1-1987, IEEE Standard Digital Interface for Programmable Instrumentation*, 2011. http://standards.ieee.org/findstds/standard/488.2-1992.html.

[9] Wikipedia - Linux Kernel. 2015. http://en.wikipedia.org/wiki/Linux_kernel.

[10] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 2013. Accepted.

[11] Wikipedia - I2C. 2015. http://en.wikipedia.org/wiki/I%C2%B2C.

[12] Wikipedia - Bash (Unix shell). 2015. http://en.wikipedia.org/wiki/Bash_%28Unix_shell%29.

[13] National Instruments. *NI-DAQmx C Reference Help*, July 2012. http://zone.ni.com/reference/en-XX/help/370471W-01/.

[14] National Instruments. *Low Cost USB DAQ Driver for use with Raspberry Pi*, 2012. https://decibel.ni.com/content/docs/DOC-25806.

[15] Texas Instruments. *INA219 - Bi-Directional Current/Power Monitor with I2C Interface*, September 2011. https://decibel.ni.com/content/docs/DOC-25806.

Apéndice A

Script bash i2c BeagleBoard xM para lectura de tensiones

Listing A.1: *Lectura de tensiones BeagleBoard xM por i2c*

```
#!/bin/bash
#set -x

# Read Vi and Vf and return: Vi Vf I
function readValues() {
    # Active GPBR1[7,4] (MADC_HFCLK_EN, DEFAULT_MADC_CLK_EN) bank 0x49
    i2cset -y -f 0 0x49 0x91 0x90

    # Turn on MADC and connect the ADC pins...
    i2cset -y -f 0 0x4a 0x00 0x01 # CTRL1: Active MADC
    i2cset -y -f 0 0x48 0xbb 0x08 # CARKIT_ANA_CTRL: Pins MCPC set to analog

    # Set which ADCs to read/average
    i2cset -y -f 0 0x4a 0x06 0x28 # SW1SELECT_LSB: Bits 3 and 5 with value 1
    i2cset -y -f 0 0x4a 0x07 0x00 # SW1SELECT_MSB: All bits with value 0
    i2cset -y -f 0 0x4a 0x08 0x28 # SW1AVERAGE_LSB: Bits 3 and 5 with value 1
    i2cset -y -f 0 0x4a 0x09 0x00 # SW1AVERAGE_MSB: All bits with value 0

    # Start read and get results
    i2cset -y -f 0 0x4a 0x12 0x20 # CTRL_SW1: Bit 5 (SW) with value 1
    CHAN3=`i2cget -y -f 0 0x4a 0x3d  w`   # GPCH3_LSB: Result from channel 3
    CHAN5=`i2cget -y -f 0 0x4a 0x41  w`   # GPCH5_LSB: Result from channel 3

    # Conversion to decimal, divided by 64 (register is 10-bits res, so we remove
    # 6 least significant bits). Then divide by 1024.
    CHAN3=`printf "%d\n" $CHAN3`
    CHAN5=`printf "%d\n" $CHAN5`

    # Divided by 64 (register is 10-bits res, so we remove 6 least significant bits)
    # Then divide by 1024 and multiplied by 2.5 for the read voltage
    # Finally, how this value is 46% of the real value, we adjust it
    CHAN3=`bc -l <<< $CHAN3/64/1024*2.5*100/46`
    CHAN5=`bc -l <<< $CHAN5/64/1024*2.5*100/46`
    CURRENT=`echo "($CHAN5-$CHAN3)*10" | bc -l`

    TMP=`echo "$CHAN3 $CHAN5 $CURRENT"`
    if [ -z "$1" ]
    then
        return 0
    else
```

```
41          eval "$1='$TMP'"
42      fi
43  }
44
45
46  function default() {
47      readValues TMP
48
49      CHAN3=`cut -d " " -f 1 <<< $TMP`
50      CHAN5=`cut -d " " -f 2 <<< $TMP`
51      CURRENT=`cut -d " " -f 3 <<< $TMP`
52
53      echo "Voltage Initial reading: `echo "scale=5; $CHAN3/1" | bc`v"
54      echo "Voltage Final reading: `echo "scale=5; $CHAN5/1" | bc`v"
55
56      # Current consumption. R13 is 0.1ohm, so... (mA scale)
57      echo "Current consumption: `echo "scale=0; $CURRENT*1000/1" | bc`mA"
58  }
59
60  if [ -z "$1" ]
61  then # Default
62      default
63  else
64      # Sampling
65      # $1 - number os samples
66      # $2 - delay between sample (seconds)
67      if [ -n "$2" ]
68      then
69          TSTART=$(($(date +%s%N)/1000000))
70          echo -e "Time(ms)\tVi(V)\tVf(V)\tCurrent(mA)"
71          for i in `seq 1 $1`
72          do
73              readValues TMP
74              echo -ne $((($(date +%s%N)/1000000)-$TSTART))
75              echo -ne "\t`echo "scale=5;$(cut -d " " -f 1 <<< $TMP)/1" | bc`"
76              echo -ne "\t`echo "scale=5;$(cut -d " " -f 2 <<< $TMP)/1" | bc`"
77              echo -e "\t`echo "scale=0;$(cut -d " " -f 3 <<< $TMP)*1000/1" | bc`"
78              sleep $2
79          done
80      else
81          echo "Syntaxis: $0 NumberSamples DelaySample"
82      fi
83  fi
```

Apéndice B

Módulo DAQmxDevice para pmlib

Listing B.1: *Módulo pmlib para adquisición de datos mediante NI DAQ*

```
# -*- coding: utf-8 -*-
#=============================================================
# DCDevice class
#=============================================================
import Device
import time
import struct
from PyDAQmx import * # DAQmx
import numpy # for PyDAQmx

## A DCMeter device description
class DAQmxDevice(Device.AttachedDevice):

    ## Creates a Cstates device description and adds it to the
    ## devices dictionary
    #
    # @param [in] name           The device name (used for identification, unique)
    # @param [in] computer       The computer the device is attached to
    # @param [in] url            The url of this device
    # @param [in] max_frequency  The maximum sample frequency of the device
    #
    def __init__(self, name, computer, url, max_frequency):
        # Declaration of variable passed by reference
        self.taskHandle = TaskHandle()
        self.firstTime = True # First time
        self.n_lines = 0 # Initially 0 lines
        self.url = url
        self.max_frequency = max_frequency
        self.index = 0

        try:
            # DAQmx Configure Code
            DAQmxCreateTask("",byref(self.taskHandle))

        except DAQError as err:
            print "DAQmx Error: %s"%err

        super(DAQmxDevice, self).__init__(name, computer, url, max_frequency)

    #  Reads data from DCDevice, pyserial package is needed in order to run
    def read(self):
    ##  Measuring device reads all channels DC
        read=int32()
```

```
44          self.data = numpy.zeros((self.max_frequency * self.n_lines,),
45                                  dtype=numpy.float64)
46          completed = False
47
48          # Initiallize the array
49          channel_scale = []
50          for i in range(0, self.n_lines):
51              channel_scale.append(1.0)
52
53          power = [[0] for i in range(0, self.n_lines) ]
54          t_muestra = self.max_frequency ** -1
55
56          while self.running:
57              # Check channel list changes
58              if self.firstTime: # First execution
59                  # DAQmx Active Channels
60                  for n in range(0, self.n_lines):
61                      DAQmxCreateAIVoltageChan(self.taskHandle,
62                                               self.url + "/" + self.lines[n].name,
63                                               "",
64                                               DAQmx_Val_RSE,
65                                               0.0,
66                                               10.0,
67                                               DAQmx_Val_Volts,None)
68
69                  # DAQmx Reconfigure Code
70                  DAQmxCfgSampClkTiming(self.taskHandle,
71                                        "",
72                                        10000.0,
73                                        DAQmx_Val_Rising,
74                                        DAQmx_Val_FiniteSamps,
75                                        self.max_frequency)
76                  self.firstTime = False
77              else:
78                  if not completed:
79                      try:
80                          # DAQmx Start Code
81                          DAQmxStartTask(self.taskHandle)
82                          # DAQmx Read Code
83                          DAQmxReadAnalogF64(self.taskHandle,
84                                             self.max_frequency * self.n_lines,
85                                             10.0,
86                                             DAQmx_Val_GroupByScanNumber,
87                                             self.data,
88                                             self.max_frequency * self.n_lines,
89                                             byref(read),None)
90
91                      except DAQError as err:
92                          print "DAQmx Error: %s"%err
93                      finally:
94                          if self.taskHandle:
95                              # DAQmx Stop Code
96                              DAQmxStopTask(self.taskHandle)
97                              completed = True
98
99                  else:
100                     for channel in range( 0, self.n_lines ):
101                         power[channel] = self.data[self.index]
102                         self.index += 1
103
104                     if self.index >= (self.max_frequency * self.n_lines - 1):
105                         completed = False # New data adquire
106                         self.index = 0
107
108                     yield power
```

Apéndice C

Módulo BeagleBoneDevice para pmlib

Listing C.1: *Módulo pmlib para adquisición de datos mediante entradas analógicas de la BeagleBone Black*

```
# -*- coding: utf-8 -*-
#=============================================================
# BeagleBoneDevice class
#=============================================================
import Device
import time
import struct
import os
import fnmatch


## A BeagleBoneDevice device description
#
class BeagleBoneDevice(Device.AttachedDevice):

    ## Creates a Cstates device description and adds it to the
    ## devices dictionary
    #
    # @param [in] name           The device name (used for identification, unique)
    # @param [in] computer       The computer the device is attached to
    # @param [in] url            The url of this device
    # @param [in] max_frequency  The maximum sample frequency of the device
    #
    def __init__(self, name, computer, url, max_frequency):
        self.path = ""
        self.n_lines = 0 # Initially 0 lines
        self.total_lines = 7 # Analog lines of BeagleBone Black [AIN0..AIN6]

        # Search for Analog Ports
        for root, dirnames, filenames in os.walk('/sys/devices/'):
            for filename in fnmatch.filter(filenames, '*AIN*'):
                self.path = root + "/"
                break # Only first needed

        if (self.path == ""): # ADC is not actived
            self.exists = False
            dirname = fnmatch.filter(os.listdir("/sys/devices/"),
                                     "bone_capemgr.*")[0] # Directory
            import subprocess
            # Activate ADC
            subprocess.Popen("echo cape-bone-iio > /sys/devices/" + dirname + "/slots",
                             shell=True)
```

```
43              print "\nADC is now active"
44
45
46          super(BeagleBoneDevice, self).__init__(name, computer, url, max_frequency)
47
48
49      ## Read function
50      #
51      #  Reads data from BeagleBoneDevice, pyserial package is needed in order to run
52      #
53      def read(self):
54      ##  Measuring device reads all channels DC
55          power       = [0] * self.n_lines
56          powerTmp    = [0] * self.total_lines
57
58          t_muestra = self.max_frequency ** -1
59
60          while self.running:
61              t1 = time.time()
62
63              if (self.path != ""):
64                  # Read all lines
65                  for i in range(self.total_lines):
66                      powerTmp[i]=float(open(self.path + "AIN" +
67                                             str(i),"rb").read().replace('\n', ''))
68
69                  # Return only selected
70                  for i in range(self.n_lines):
71                      power[i]=powerTmp[int(self.lines[i].name)]
72              else:
73                  # Take the path
74                  for root, dirnames, filenames in os.walk('/sys/devices/'):
75                      for filename in fnmatch.filter(filenames, '*AIN*'):
76                          self.path = root + "/"
77                          #print "ADC Directory %s" %(root + "/")
78                          break # Only first needed
79
80              yield power
81
82              w = t_muestra - (time.time()- t1 )
83              if (w > 0): time.sleep(w)
```

Apéndice D

Módulo INA219Device para pmlib

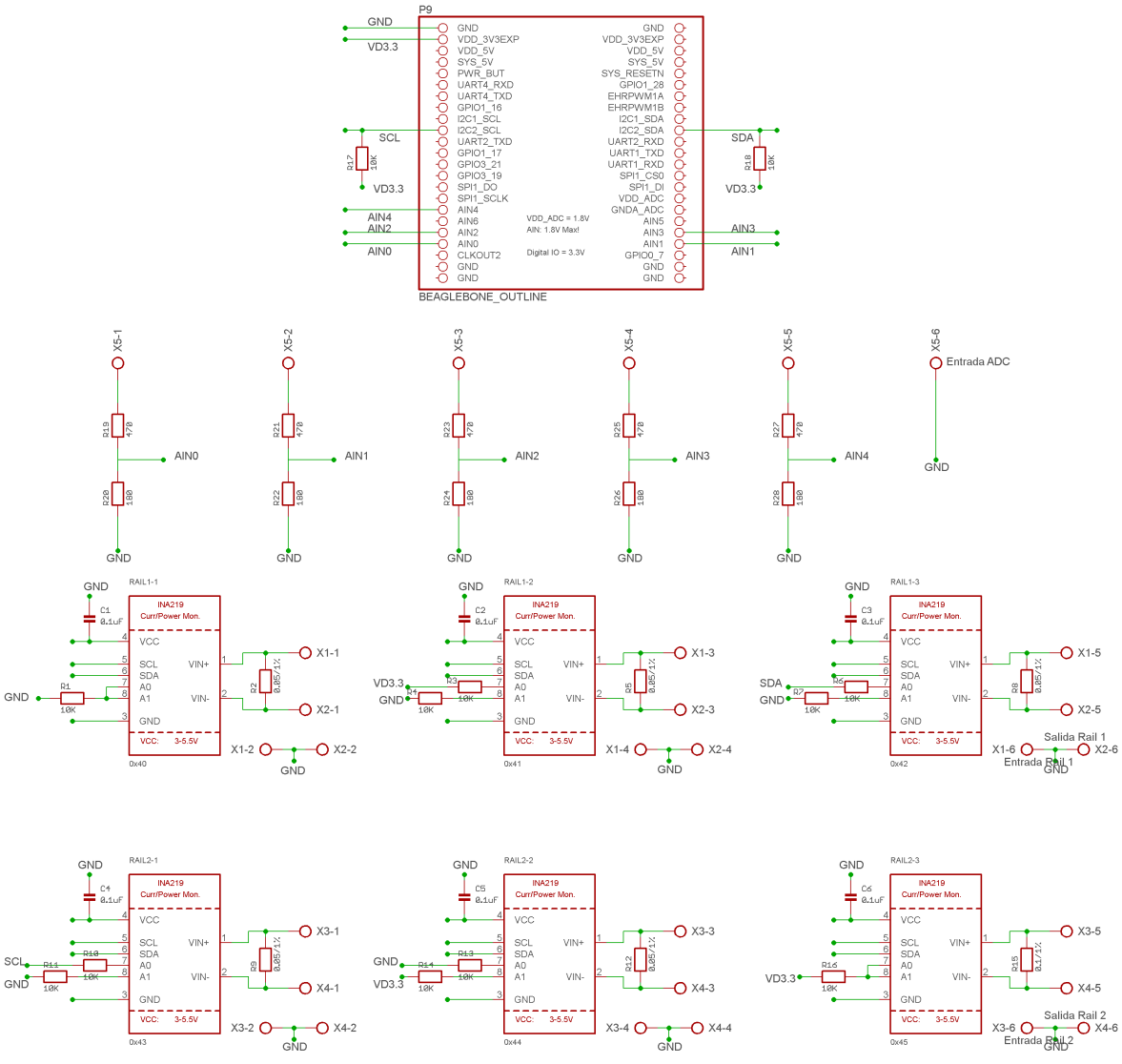
Listing D.1: *Módulo pmlib para adquisición de datos mediante INA219 de Texas Instruments*

```
# -*- coding: utf-8 -*-
#=============================================================
# INA219Device class
#=============================================================
import Device
import time
import struct
import os
import fnmatch
from INA219 import INA219


## A INA219Device device description
# The parameter "name" is used for the I2C address
class INA219Device(Device.AttachedDevice):

    ## Creates a Cstates device description and adds it to the
    ## devices dictionary
    #
    # @param [in] name           The device name (used for identification, unique)
    # @param [in] computer       The computer the device is attached to
    # @param [in] url            Type of measurement.
    #                            Values: 'Voltage', 'Current', 'Power'
    # @param [in] max_frequency  The maximum sample frequency of the device
    #
    def __init__(self, name, computer, url, max_frequency):
        self.n_lines = 0 # Initially 0 lines
        self.objects = []

        super(INA219Device, self).__init__(name, computer, url, max_frequency)



    ## Read function
    #
    #  Reads data from BeagleBoneDevice, pyserial package is needed in
    #  order to run
    #
    def read(self):
    ##  Measuring device reads all channels DC

        power       = [0] * self.n_lines
```

```
43          t_muestra = self.max_frequency ** -1
44
45          while self.running:
46              t1 = time.time()
47
48              if (len(self.objects) == 0): # First execution, create objects
49                  for i in range(self.n_lines):
50                      # Create object for this line
51                      self.objects.append(INA219(int(self.lines[i].name,16),False))
52              else:
53                  for i in range(len(self.objects)):
54                      if self.url == "Current": # Read current from this line
55                          power[i] = self.objects[i].getCurrent_mA()
56                      elif self.url == "Voltage": # Read voltage from this line
57                          power[i] = self.objects[i].getBusVoltage_V()
58                      else: # Default, read power from this line
59                          power[i] = self.objects[i].getPower_mW()
60
61
62              yield power
63
64              w = t_muestra - (time.time()-t1)
65              if (w > 0): time.sleep(w)
```

Apéndice E

Esquemático CAP BeagleBone Black



Apéndice F

Distribución de componentes CAPE BeagleBone Black

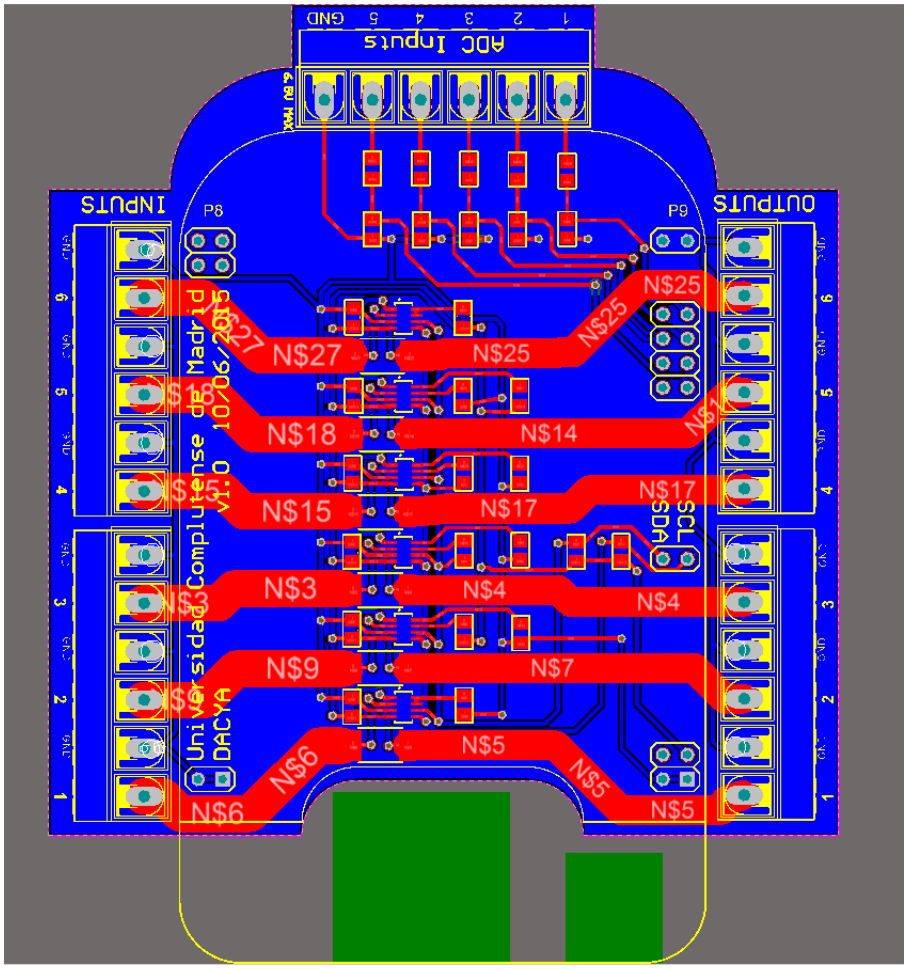


Figura F.1: *Vista general de la placa.*

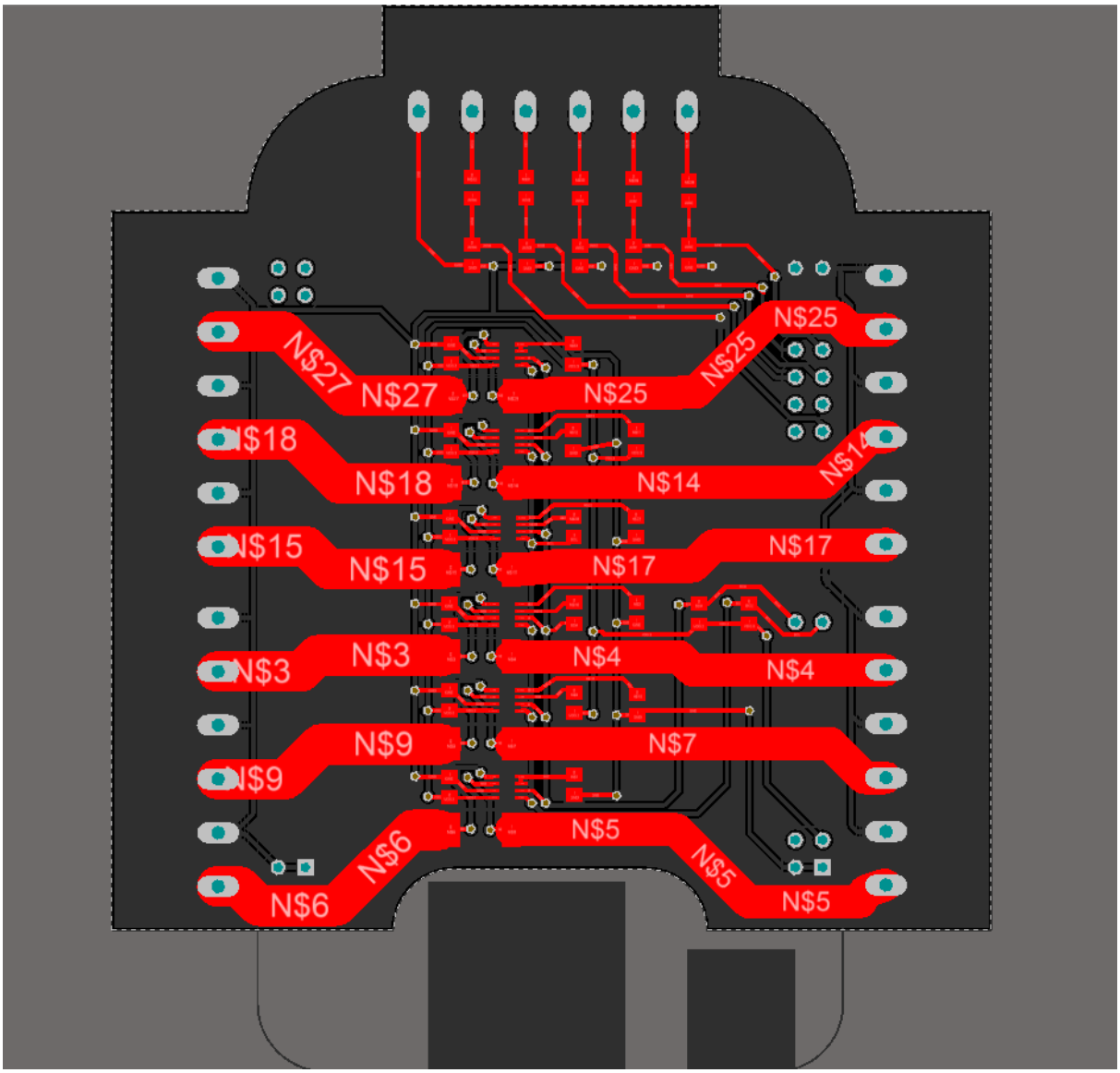


Figura F.2: *Detalle de la capa superior de la placa.*

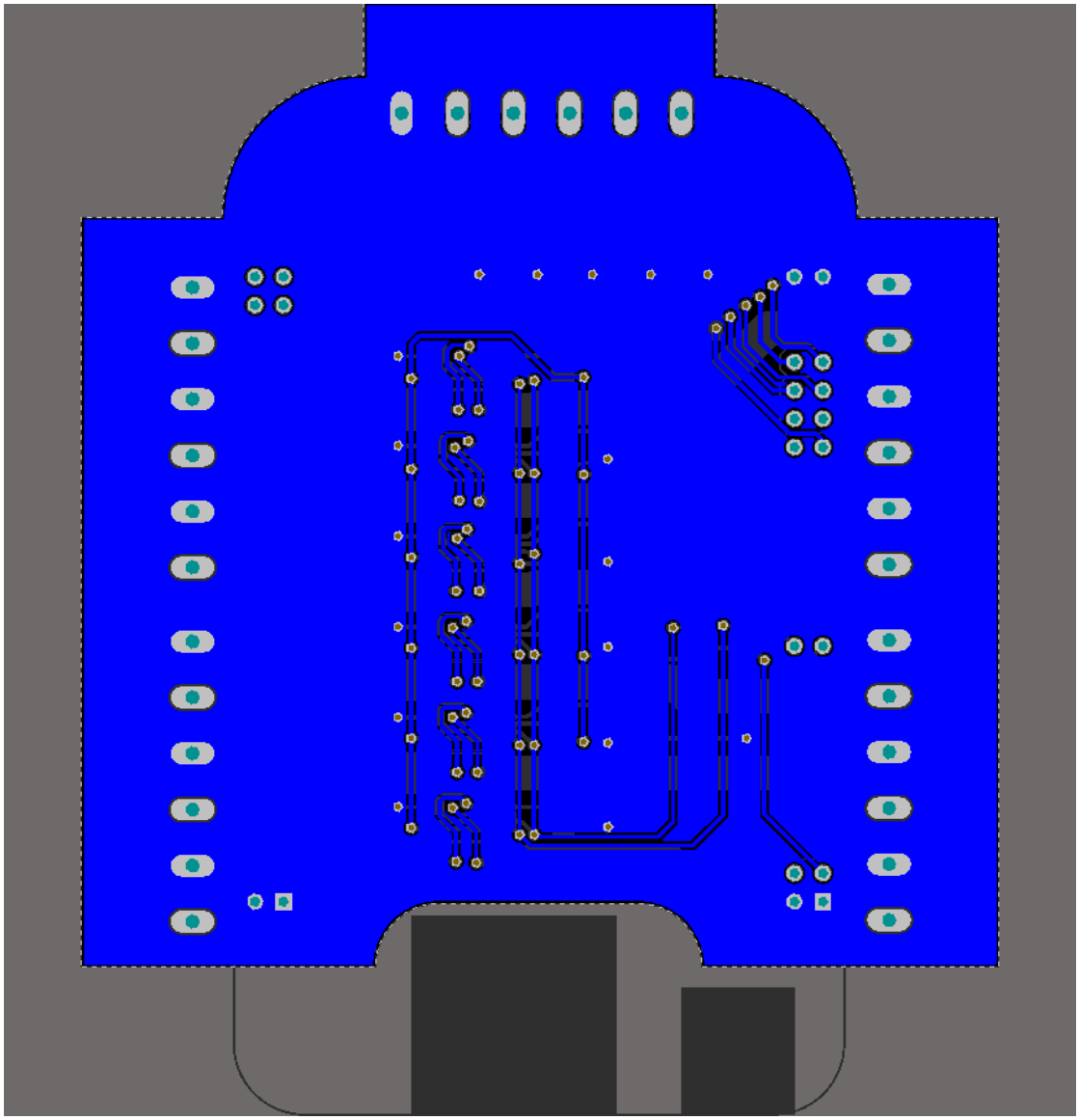


Figura F.3: *Detalle de la capa inferior de la placa.*

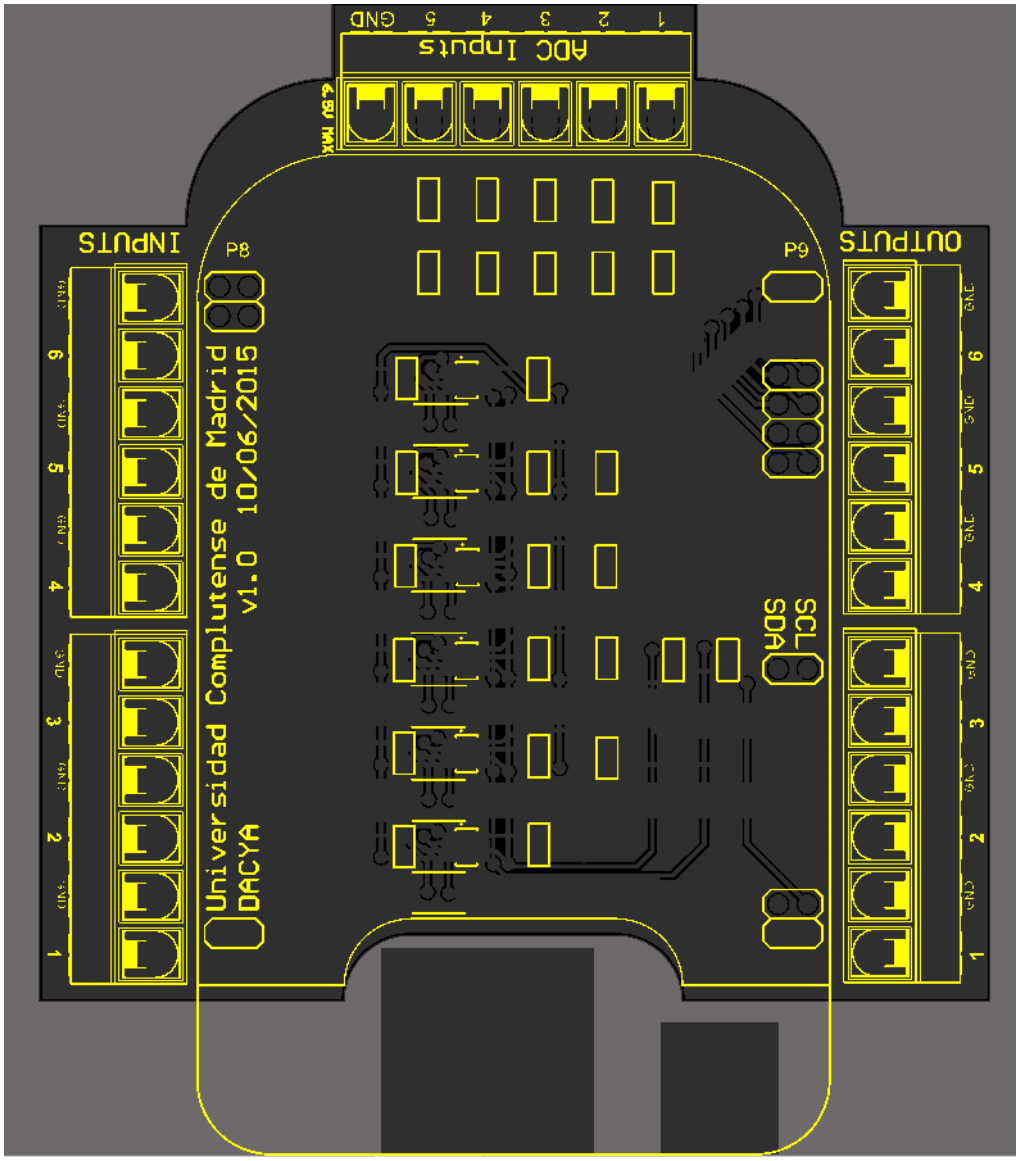


Figura F.4: *Detalle de la serigrafía de la placa.*

Apéndice G

Introduction

G.1. Motivation

Nowadays, a correct a understaning of the behavior of a given architecture not only from the performance perspective, but also considering energy consumption, becomes fundamental to attain higher compute capabilities at a exchange of a reduced power consumption. This type of analysis will drive both industry and scientific community towards the *exaflop* barrier[1].

Many of the hardware environments available nowadays allow a correct real-time energy monitoring, but do not take into account the fact that the analysis procedure itself yields an extra power consumption that can dramatically modify the actual results. In addition, they usually do not consider external factors such as active cooling, but are exclusively focused on CPU consumption. This problem is even more dramatic on heterogeneous architectures, in which differences in performance and power consumption between architectures is even more relevant.

In order to alleviate those pitfalls, there exist a number of measurement environments that can extract energy consumption measurements in a robust and precise way.

The main problem with this type of *commercial* environments is acquisition cost. In

[1] **FLOP**: Floating point operations per second.

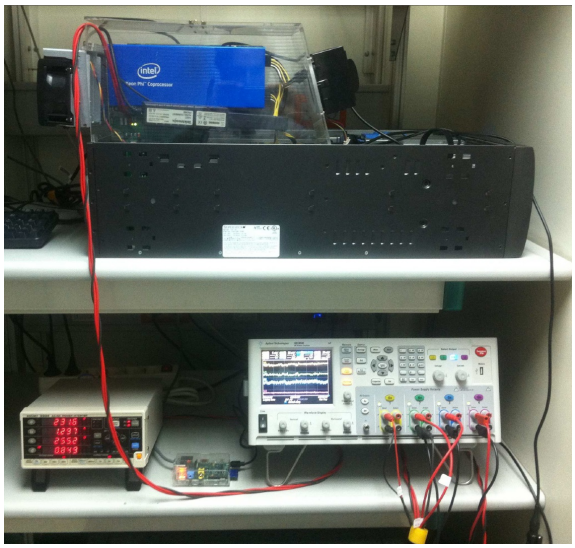


Figura G.1: *Example of power consumption measurement environment [1].*

addition, their huge volume is usually a common problem towards their integration into production environments (e.g. datacenters), where the facility is usually designed and integrated in a compact way; additionally, these environments typically limit the amount of of simultaneous measurements that can be carried out. Finally, portability to different hardware setups is usually another problem in many measurement environments.

As an example, the *Agilent N6705B* power analyzer, used in this project as a validation environment, features an acquisition cost of 5686€[2]. Another example is the *Hioki AC/DC HiTester 3334*, also used as a measurment tool in many environments, with an acquisition cost of 1400€[3].

G.2. Main goals

The main goal of the project is to provide a precise, reliable and affordable power measurement platform.

As said in Section 1.1, the use of commercial solutions usually involves a number of problems, many of them solved after the development of the project.

[2] http://es.rs-online.com/web/p/power-quality-analysers/7110012/
[3] http://www.tester.co.uk/hioki-3334-power-meter

More precisely, the specific goals of the project are:

- **Low acquisition cost:** This can be considered as the main goal of the project, without trading off measurement quality. To accomplish this goal, we will base the development on an *Open-Source Hardware*[2] device.

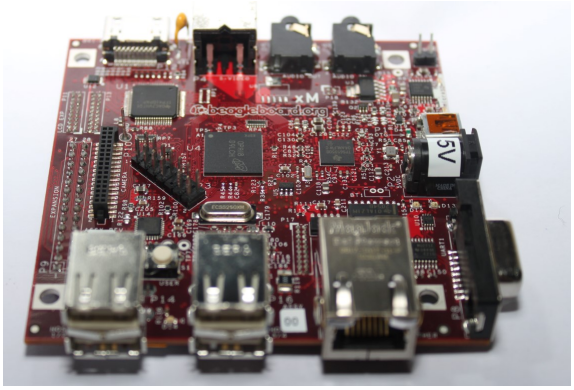


Figura G.2: *BeagleBoard xM used in our tests.*

- **Compact form factor:** We will look for a reduction in the form factor of the device, in order for it to be portable and easily replicated in reduced environments where the necessities in terms of measurement channels (**no entiendo la frase...**)

- **Reliability and precision:** In our development process, we will validate our environment comparing the attained results with those obtained from thoroughly tested hardware setups, or proprietary software tools.

- **Portability:** In addition to the compact form factor, we will pursue the idea of developing an autonomous device, with *Plug and play* [3] capabilities.

- **Ease of use:** Our environment will be easy to use, with no requirements of knowledge of the underlying platform and a smooth learning curve.

In order to arrive to a final solution, we will start our development from tested environments, that will be evolved and adapted in order to meet the aforementioned requirements.

In addition, we will base our software solution on an existing tool the *PowerMeter daemon* (`pmlib` [4]) that will be detailed in Section 2.3.

Apéndice H

Conclusions

After the development of the project, we can consider two main milestones, each one defining a different environment and featuring different characteristics and possibilities, but both accomplishing the main goals of the project.

H.1. Replacement of Agilent DC Power Analyzer N6705B as CMU for NI DAQ USB

One of the main goals has been the possibility of replacing the expensive *Agilent DC Power Analyzer N6705B* as the main measurement device. This replacement has carried out, additionaly, an increased performance in other aspects, namely:

- **Number of channels**: The *NI USB-6218* provides the ability to simple up to 32 channels simultaneously, compared to the 4 channels of *N6705B*. Regarding to *NI USB-6009*, the number of channels that allows to measure is 8, so in this case we effectively increase the number of channels to monitor.

- **Sampling frequency**: in this case is also very significant the sampling rate of the DAQ against the replaced equipment. The *NI USB-6218* exhibits a sampling frequency of 250.000 samples per second versus 100.000 of the replaced equipment. With the *NI USB-6009* the frequency decreases, and in this case does not overcome the *N6705B*;

in any case has a sufficient speed sampling and valid for most of the environments: 48.000 samples per second.

- **Device cost**: The most important difference is the difference between one measurement equipment and others.

  Comparing the *N6705B* –5686€[1]– with the *NI USB-6218* –1490€[2]–, we obtain a budget saving of almost a 75 %, which is the same as having an expense of $\frac{1}{4}$ compared to the initial environment device. Comparing the *N6705B* with the other device used of *National Instruments*, the *NI USB-6009* –315€[3]– the saving is much more important, reaching almost to 95 %, or in other words, minimizing the initial cost up to $\frac{1}{18}$.

H.2. BeableBone Black solution with CAPE custom-built as low cost MI-CMU-MS set

Finally, after considering a number of choices, we reached a system that allows to monitor virtually any kind of environment. Moreover, the *BeagleBone Black* equipped with an ad-hoc *CAPE* reduces costs and gives a great versatility, being possible to make changes in a very simple way.

There are many advantages that we can get through the environment posed here:

- **Number of channels**: Being a custom-designed, it has been chosen the possibility of create a device capable of monitoring the 4-way power of *Intel Xeon Phi* device of the starting project environment –see section 2.1 for more information–.

  As explained in section 3.5.5, it is needed to monitor a total of 8 tracks, but taking advantage of the space and the possibilities of the analog-digital converters of *BeagleBone Black*, it has been selected the possibility of including 3 more measuring inputs,

[1] http://es.rs-online.com/web/p/power-quality-analysers/7110012/
[2] http://sine.ni.com/nips/cds/view/p/lang/es/nid/203484
[3] http://sine.ni.com/nips/cds/view/p/lang/es/nid/201987

for a total of 11 –6 *TI INA219* and 5 ADC of *BeagleBone Black*–.

It should be remembered that *BeagleBone Black* has **7 ADC**, which added to the potential **16** *TI INA219* that we can connect in a single data network, for a total of **23 lines of analysis goals**. If necessary, it can be further expanded the number of devices *TI INA219* performing multiplexing, which gives us a number of possibilities that hardly give another kind of device.

If it is compared all of this with the 4 channel that could be monitored with *N6705B* in the initial environment, it is shown that is a very versatile option.

- **Lower cost**: as with the alternative given in section 5.1, the cost is drastically reduced. The cost of our base platform, *BeagleBone Black* one, barely exceeds 50€[4]. If we add the used components in the developed *CAPE* during this project, together is not reached the cost of 150€–approximately prototype price, including the *PCB* manufacture– barely $\frac{1}{38}$ –or less than 2,5 %– from the cost of *N6705B*. These costs could be cutted down if it further production of printed wiring boards are made.

- **Does not require Clamp Ampmeters**: thanks to the *TI INA219* that allows us to monitorize directly the current that moves along through the rail, it is not needed the use of clamps ampmeters. In our initial environment it has been used two of them for two external power supplies, which means that we can reduce the cost in more than 1300€.

- **Voltage measurement**: another advantage of *TI INA219* is that it allows us measure the voltage supply lines. These data gives us the ability of adjusting consumption as **we do not start from the premise of constant voltage**.

- **Instantaneous power**: one of the great features that it can be highlighted of integrated *TI INA219* is the measurement of the consumed power in an instant of time.

[4] http://es.rs-online.com/web/p/kits-de-desarrollo-de-procesador-y-microcontrolador/7753805/

As mentioned above, it allowed us the voltage and current capture, so in principle this provision may seem unnecessary. In our case is really useful because to read the voltage and the current it should be needed to make two requests by the integrated I2C, which carries a delay and this would bring that the measure would deface, something that does not happen it the integrated returns us the calculated power at a particular time.

On the other hand, it has been also found some sort of disadvantage. For the working of *TI INA219* is needed a precision *shunt* resistor, with a very low value –0,05 Ohm in our assembly–, but that entails a small distortion in the measured channel. Although this variation represents only 0.01 % of the measurement, if necessary, the consumption of the own resistance can be calculated and corrected later in our **MI** –section 2.2.3–.

Another drawback is the limitation of ADC channels, since they only support 1.8v, and they also monitorize voltage. Even so, modifying the configuration of resistive divider, we could vary the tension capacity to be measured.

Following the philosophy *Open Source Hardware*, on which is supported this project with the use of *BeagleBone Black*, it has been made available to the Community all the schematics and sources that allow anyone to modify the initial development as needed. For this, this project has been published in *Open Hardware Repository*, under the name of **Fine grain powermeter for HPC accelerators**[5] and licensed under a CERN OHL v1.2[6].

[5] http://www.ohwr.org/projects/cape-pwr-12b-6ch
[6] http://www.ohwr.org/attachments/2388/cern_ohl_v_1_2.txt