Desarrollo de aplicaciones embebidas de control en robots móviles

Universidad Politécnica de Valencia
Máster en Automática e Informática Industrial

Autor
Jorge Coronado Vallés

Director
Ángel Valera

Codirector
Jose Ignacio Cazalilla Morenas

Valencia, Septiembre 2013

Agradecimientos

Son tantas las personas que influyen de una forma u otra en la vida de una persona, que es prácticamente imposible dedicar el trabajo realizado a todos sin olvidar algún nombre.

Personas que ya no están, personas que están y espero sigan estando por muchos años, mucha gente que ha ayudado de una forma u otra a que este trabajo se complete, pues no deja de ser una fase más de tantos y tantos años de estudio y esfuerzo, durante los cuales mucha gente ha tenido que aguantar mis preguntas, mis frustraciones, mis horas de abstracción en las que he olvidado hasta comer... incluso la felicidad de un resultado positivo, que cuando es explicado de forma entusiasta estas personas han sonreido y asentido con la cabeza como diciendo "¿de qué me está hablando ahora?"

Por eso no hace falta nombrar a esos familiares, a esa compañera, a esos amigos o a esos director y codirector (seguro que acabas dirigiendo cientos de grandes trabajos, pero tengo el orgullo de saber que "yo fui el primero"). Ellos saben que están ahí y que, sin ellos, nada de esto habría sido posible.

Índice

1. Introducción, estado del arte, objetivos y justificación

Como se sabe, la robótica es un área multidisciplinar que combina mecánica, informática, ingeniería de control, electrónica e inteligencia artificial. Debido al rápido avance de la robótica, han ido apareciendo nuevos requerimientos, distintos a los que estaban anteriormente.

En los inicios de la robótica, el sistema de control era, típicamente, una especie de "caja negra" desde la que se tenía acceso a los motores del robot (por ejemplo, el Puma 500). Este sistema de control solía tener un tamaño considerable, por lo que, era una gran desventaja, por ejemplo, para poder controlar un robot móvil. Con el paso del tiempo, este sistema de control se fue sustituyendo por PC Industriales, siendo más accesible y más cómodo de manejar. Sin embargo, debido a su peso, tamaño y requerimientos energéticos, su uso se limita a robots estáticos o prototipos lo suficientemente grandes como para poder albergar un PC Industrial.

De ahí, dentro del área de la robótica aparecen los sistemas empotrados. Estos sistemas, nacieron aproximadamente en la década de los 70 y su objetivo principal es ejecutar tareas de control. Aunque estos sistemas nacieron en el ámbito industrial, "la tendencia es incluirlos de forma masiva en dispositivos y sistemas de uso general y particular, aumentando de forma continua su potencia, mientras que el precio disminuye". [1]

Precisamente, ya forman parte de nuestra vida cotidiana y los podemos encontrar en el control de muchas aplicaciones de automoción (frenos ABS, controles de inyección...), en la industria (control de motores y robots) así como en electrónica de consumo, como electrodomésticos, o dispositivos móviles.

Estos sistemas cumplen unas restricciones [2] muy estrictas:

- De tiempo. A menudo deben reaccionar en un periodo de tiempo muy corto
- Cada vez deben ser más ligeros, portables y baratos
- El consumo de energía debe ser muy reducido

Probablemente, en los últimos años su evolución ha sido todavía mayor gracias a la creciente demanda de dispositivos móviles.

La tendencia es, pues, dispositivos cada vez más potentes y a la vez, económicos, reduciendo al mismo tiempo su tamaño para que puedan formar parte de sistemas más complejos.

Cabe destacar que, precisamente, una de las tarjetas analizadas está diseñada como plataforma de

aprendizaje a muy bajo coste para países en vías de desarrollo. No es necesario invertir gran cantidad de dinero en una CPU, aún en investigación, cuando algunas de estas soluciones se pueden adquirir por debajo de 50€.

Algunas de las características más importantes de los sistemas empotrados son las siguientes:

- **Tamaño reducido**: las medidas de las soluciones del presente trabajo no superan a las medidas de una tarjeta de crédito.
- **Peso reducido**: Menos de 50 gramos
- **Requerimientos de energía reducidos**: son sistemas que se alimentan con 5V, con un consumo inferior a los 700 mA (menos de 4W) con lo que pueden alimentarse con baterías muy ligeras, con una autonomía considerable
- **Alta conectividad:** en muchos casos encontramos conectividad por USB, ethernet, Wifi, CAN, I2C, GPIO, etc

Por todo ello, son una clara ventaja en el desarrollo de la robótica móvil, ya sean robots terrestres como aéreos, donde los requisitos de tamaño y peso son todavía más ajustados.

Por tanto, en el presente trabajo, se pretende facilitar la construcción de un proyecto de robótica mediante tarjetas de electrónica embebida programables y herramientas software para dichas tarjetas.

Para ello, se han analizado 3 tarjetas hardware de tamaño similar apropiadas para dichos proyectos, debido a su precio, conexiones, tamaño, popularidad y compatibilidad con otro hardware usado en robótica (sensores, servomotores, etc)

Así pues, el desarrollo del presente proyecto queda justificado mediante uno de los objetivos principales, que es obtener una plataforma de desarrollo orientada a la robótica móvil de bajo coste, alta capacidad de cómputo y con la máxima compatibilidad y versatilidad para los distintos proyectos que se puedan plantear.

A partir de ello, se ha realizado una comparativa de 3 soluciones hardware, en las cuales se busca integrar el software Robotic Operative System (ROS) y Open RObot Control Software (OROCOS) y probar su rendimiento.

Finalmente, con una de las soluciones conseguidas, se pretenderá controlar un robot móvil usando el middleware Orocos y las interfaces de conexión que la propia tarjeta proporciona.

La elección de dicho software viene fijado por ser Software Libre, bien documentado, muy utilizado en investigación y concretamente en proyectos actuales de la UPV[3], siento otro de los objetivos de este TFM el proporcionar una base a partir de la cual trabajar en nuevos proyectos.

2. Desarrollo Teórico

2.1. Componentes Hardware estándar utilizados

El hardware utilizado, además de las tarjetas que se describen a continuación, es todo hardware muy sencillo de encontrar y que sigue los principios de los sistemas empotrados: bajo coste, medidas reducidas, bajo peso y, en su caso, alimentación reducida (si la precisa).

- **HUB USB (alimentado)**

Las tarjetas analizadas tienen muy pocos puertos de conexión USB (1 ó 2). Además, el consumo de las tarjetas es muy reducido, por lo que es complicado alimentar muchos dispositivos por USB, máxime cuando es posible que en nuestro proyecto utilicemos dispositivos que tengan un alto consumo, como pueden ser cámaras, motores, un Kinect, etc

Un HUB USB puede ser un buen complemento para tratar con distintos elementos USB, su coste no supera los 12 euros y, además, los hay de peso muy reducido.

Ilustración 1: Hub USB alimentado

- **Cables HDMI y USB**

Más que necesarios para los proyectos, otorgan comodidad. Las tres tarjetas analizadas tienen salida de vídeo por HDMI, por lo que resulta más cómodo conectarlas a un monitor que realizar todas las tareas desde otra máquina.

Uno de los objetivos es que toda tarea, incluyendo la compilación y programación, pueda realizarse en las tarjetas, sin necesidad de hardware adicional.

- **Adaptador inalámbrico TP-LINK WN725N.**

Se ha escogido este adaptador por su bajo precio (9 euros), bajo peso (10 gramos) y pequeño tamaño.

Es compatible con las tarjetas descritas, aunque cualquier adaptador inalámbrico compatible con Linux debería servir. El Chipset RALINK da buenos resultados de compatibilidad.

Ilustración 2: Adaptador Wifi TP-Link

- **Teclado USB**

Al igual que los cables HDMI y USB, no es necesario pero aporta comodidad a la hora de trabajar con las tarjetas analizadas.

- **Tarjeta de memoria microSD con adaptador SD**

Es necesaria para instalar el sistema operativo y sistema de archivos de las tarjetas. Su coste puede variar dependiendo de tamaño y categoría, aunque se puede encontrar por menos de 5 euros.

- **PC**

De nuevo, más que un elemento necesario, es un elemento que permite trabajar con mayor comodidad. Es preciso Linux (se ha utilizado Ubuntu 12.04) y Windows (cualquier versión) para realizar compilación cruzada, supervisión de datos, etc

2.2. Tarjetas embebidas utilizadas

A continuación se van a comentar cada una de las tres las tarjetas embebidas que han sido analizadas en el presente TFM.

2.2.1. Raspberry PI

La Raspberry PI es una placa desarrollada por la Fundación Raspberry. El objetivo de esta fundación, nacida en el Reino Unido, es crear una placa de muy bajo coste para distribuir en países del Tercer Mundo, fomentando así el estudio de la computación en dichos países y facilitando el acceso a Internet.

Para lograr un coste reducido, está basada en una CPU ARM y cuenta por defecto con un Sistema Operativo llamado Raspbian, que es una modificación de Debian realizado para la placa. Debido a su reducido tamaño y bajo coste, esta placa ha tenido una repercusión mediática muy importante, siendo la placa que cuenta con mayor soporte de la comunidad.

Existen dos tipos de modelos, y se diferencian, principalmente, en las conexiones de red y en la memoria RAM. Concretamente, se ha utilizado el modelo B. Sus características Hardware son las siguientes:

	Modelo A	Modelo B
Precio	25 $	35 $
SoC	BroadCom BCM2835	
CPU	ARM1175JZ (700Mhz, ARM11)	
GPU	BroadCom VideoCore IV, OpenGL ES 2.0 1080p	
Memoria SDRAM	256 MB (compartidos)	512 MB (compartidos)
Puertos USB	1	2
Entradas Vídeo	Conector MIPI. Permite instalar una cámara RF	
Salidas Audio	Conector Jack 3.5 mm y HDMI	
Salidas Video	Conector RCA y HDMI, interfaz DSI para LCD	
Almacenamiento	SD/MMC/SDIO	
Red	Ninguna	Ethernet 10/100 RJ-45
Periféricos	8 x GPIO , SPI, I2C, UART	
Reloj RT	Ninguno	
Consumo	500 mA, 2.5 W	700 mA, 3.5 W
Dimensiones	85.60mm × 53.98mm	
Alimentación	5 V	

(Información extraída de www.raspberrypi.org)

Ciertamente, destaca su capacidad gráfica aunque es una característica que en principio no aporta nada para proyectos de robótica y similares.

Lo que sí es muy importante es su conectividad, ya que por GPIO o I2C se pueden conectar infinidad de dispositivos. Por ejemplo, por I2C pueden conectarse toda la variedad de actuadores y sensores de Lego Robotics, así como drivers capaces de generar hasta 16 señales PWM independientes (y así controlar infinidad de motores).

En cuanto a Sistema Operativo, hay una variedad inmensa de opciones. Las más destacables son las siguientes:

- **Raspbian** (debian para raspberry): Sistema operativo desarrollado por la Fundación Raspberry, basado en Debian.
- **RiscOS**: Sistema operativo diseñado en Cambridge por la extinta empresa Acorn en el año 1987. Actualmente es propiedad de Castle Technology.
- **Archpi Linux**: Sistema operativo basado en Arch.

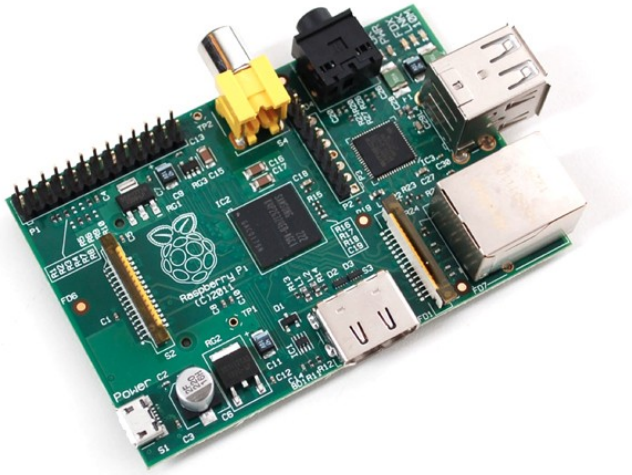

Ilustración 3: Raspberry PI B

2.2.2. BeagleBone

La Beaglebone ha sido desarrollada por la BeagleBoard Foundation, de nuevo una Fundación sin ánimo de lucro, esta vez de origen Estadounidense, y cuyo objetivo es promocionar el uso de software y hardware open-source para el desarrollo de sistemas empotrados.

Beagleboard nace como un foro de desarrolladores software y hardware, donde pueden intercambiar ideas, conocimiento y experiencia.

El germen de BeagleBoard podemos encontrarlo en la empresa Texas Instruments, patrocinadora de la fundación dado su interés de crear dispositivos empotrados de carácter abierto y con distintas posibilidades

Como podemos encontrar en BeagleBoard, el objetivo es crear "dispositivos de bajo coste, basados en procesadores de bajo consumo de Texas Instruments, utilizando núcleos ARM Cortex-A". Es decir, "hacer más con menos" [4]

	Modelo Principal	Modelo Black	Modelo xM
Precio	89 $	45$	149$
SoC	TI AM3358	TI AM3359	TI DM3730
CPU	720 MHz ARM Cortex-8	1 GHz ARM Cortex-8	1 GHz ARM Cortex-8
GPU	Power VR SGX530	Power VR SGX530	Unspecified PowerVR SGX
Memoria	256 MB DDR2	512 MB DDR3	512 MB DDR2
USB	1	1	2
Salida video	micro HDMI	micro HDMI	DVI, S-Video
Salida audio	micro HDMI	micro HDMI	3.5 mm jack
Periféricos	4x UART, 8x PWM, LCD, GPMC, MMC1, 2x SPI, 2x I2C, A/D Converter, 2xCAN Bus, 4 Timers, FTDI USB to Serial, JTAG via USB	4x UART, 8x PWM, LCD, GPMC, MMC1, 2x SPI, 2x I2C, A/D Converter, 2xCAN Bus, 4 Timers	McBSP, DSS,I2C, UART, LCD, McSPI, PWM, JTAG, Camera Interface

(Información extraída de beagleboard.org)

Es de nuevo destacable la conectividad de la tarjeta, ya que cuenta con PWM por hardare, I2C e incluso bus CAN.

En cuanto a Sistemas Operativos podemos encontrar una variedad todavía mayor que en la Raspberry Pi, ya que el ARM que montan estas tarjetas no está obsoleto por lo que cuenta con el soporte de multitud de distribuciones Linux y Android. Para la redacción del presente documento se ha utilizado Ubuntu 12.04.

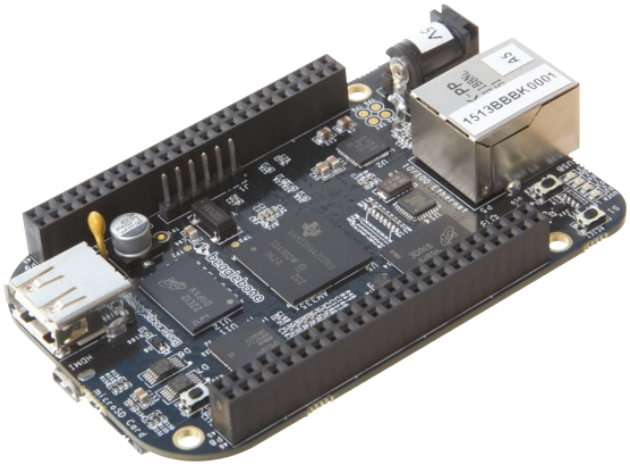

Ilustración 4: Beaglebone

2.2.3. Igep V2

La Igep V2 es una placa ARM de fabricación española, producida por la empresa patria ISEE. Isee es una empresa de ingeniería especializada en Sistemas Empotrados, fundada en 2006 por Agustí Fontquerni y Manel Caro, con el objetivo de ofrecer soluciones para la vida diaria y que puedan ayudar a aumentar la productividad y reducir costes.

La Igep V2 es una placa de carácter industrial**,** con el objetivo de "aportar la potencia de una máquina de escritorio pero de forma más económica, pequeña y con menor consumo".

Su temperatura de trabajo le permite trabajar en estos ámbitos industriales, ya que trabaja en el rango de [-40;85] ºC y con una vida útil muy larga.

Isee diseña y provee también de dispositivos industriales compatibles con Igep V2, como por ejemplo el Sensor Lambda, que puede medir la distancia, velocidad y propiedades físicas de distintos objetos, y cuya salida puede ser conectada a la Igep V2 para el tratamiento de la información.

La Igep V2 es también una placa de hardware abierto, publicada bajo licencia Creative Commons, siendo sus principales caracterísiticas las siguientes:

Precio	250$
CPU	DM3730 ARM 1GHz Cortex-A8 + Coprocesador Neon SIMD
GPU	PowerVR SGX530 200 MHz
Memoria	512 MB Ram + 512 MB Flash
USB	2
Salida Video	HDMI
Salida Audio	3.5 mm jack
Conexiones	Wifi 802.11b/g, bluetooth 2.0, ethernet 100Mb
Periféricos	RS-485, UART, McBSP, McSPI, I2C, GPIO

(Información extraída de isee.biz)

A pesar de ser una tarjeta de mayor coste que BeagleBone, vemos que no cuenta con bus CAN, aunque sí con otros tipos de conectividad. Sin embargo, esta tarjeta está concebida totalmente para ser utilizada en ambientes industriales, siendo mucho más robusta que sus homólogas.

Al contar con el mismo procesador que Beaglebone, podemos instalar los mismos sistemas operativos: Ubuntu, Arch, CentOS... También se ha utilizado Ubuntu 12.04

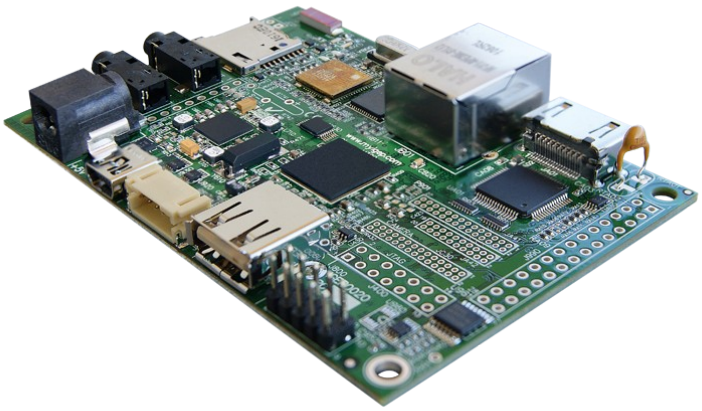

Ilustración 5: Igep V2

2.3. Software

En esta sección se comentará todo lo relacionado con el software que se ha empleado en este proyecto. En primer lugar, la elección del sistema operativo elegido, así como el middleware Orocos y el meta-sistema operativo ROS.

Como base software, se ha intentado integrar en las distintas tarjetas Orocos y Ros, debido a su extensa documentación, publicaciones relacionadas y ser Software Libre, lo que facilita su adaptación a cualquier proyecto, especialmente para investigación.

Además de ello, es necesario que las tarjetas cuenten con un sistema operativo determinado. Para facilitar su instalación y configuración, se ha añadido un apéndice para cada una de las tarjetas.

2.3.1. Sistema Operativo

En cuanto al sistema operativo a utilizar, se partía de dos posibles soluciones: Android y Linux. Evidentemente, la opción de instalar Windows en cualquiera de las tarjetas empotradas se descartó desde un principio, ya que no es un sistema operativo ni libre ni gratuito (además de no existir soporte de Windows para este tipo de tarjetas).

En cuanto a Android y Linux, bien es cierto que Android ha tomado bastante ventaja sobre Linux en el ámbito de los dispositivos móviles. Sin embargo, Linux sigue siendo la opción más popular para realizar el control mediante este tipo de tarjetas embebidas, debido a la versatilidad y a las posibilidades que ofrece el código abierto (gracias a su sólida comunidad de desarrollo).

Finalmente, otro de los motivos principales para instalar Linux es que tanto ROS como Orocos proporcionan el soporte necesario para esta plataforma.

Como se verá en las próximas secciones, se ha instalado Raspbian (para la RaspberryPI) y Ubuntu 12.04 para BeagleBone y Linaro para Igep V2.

2.3.2. Orocos

Orocos es un middleware (o framework) de Tiempo Real escrito en C++ bajo la idea inicial de Herman Bruyninckx, aunque el proyecto ha ido creciendo y adhiriéndose distintos grupos y programadores. [5]

La idea inicial para Orocos era la creación de un framework de Tiempo Real basado en Software Libre para control de robots, ya que hasta entonces se había intentado utilizar el software industrial de robots para investigación, con pésimos resultados.

Su primera versión, como tal, fue lanzada en 2002, aunque era bastante complicada de usar y se probó su uso con un brazo manipulador de 6 grados de libertad, controlando su posición y velocidad. Desde ese punto, su desarrollo se disparó de forma que en menos de un año se lanzaron 7 versiones diferentes.

Aunque como se ha comentado, su nacimiento surgió por los pésimos resultados que el software industrial tuvo en la investigación, se ha acabado instaurando en el control de la máquina, dejando atrás sus inicios en la investigación sobre la robótica.

Destacó desde su inicio por su versatilidad y portabilidad, gracias a su arquitectura basada en componentes. Destaca por su modularidad, lo que le otorga de versatibilidad para adaptarse a distintos proyectos, incluyendo paquetes complementarios tales como Filtros de Bayes, Librerías de control Dinámico y Cinemático o Visión.

Sus tres objetivos principales, son:

- **Basado en Componentes**. Se puede añadir funcionalidades de forma sencilla y sin recompilar todo el código. Es decir, permite reutilizar el código fuente, seguir el flujo de ejecución de forma sencilla, instanciar distintos componentes o incluso ejecutar componentes de forma distribuida con comunicación entre sí. [6]

- **Multi-vendor**: Los sistemas más complejos se construyen habitualmente escogiendo partes de distintas fuentes (empresas, proyectos open source, etc) por lo que Orocos tiene la arquitectura necesaria para adaptarse a cualquier componente adicional, ya sea hardware o software.

- **Enfocado a su objetivo**: Orocos quiere ser el mejor framework dentro de su nicho de aplicación, por lo que si alguna característica no se adapta a Orocos, se relega a proyectos colaborativos similares, como pueden ser Corba o Xenomai.

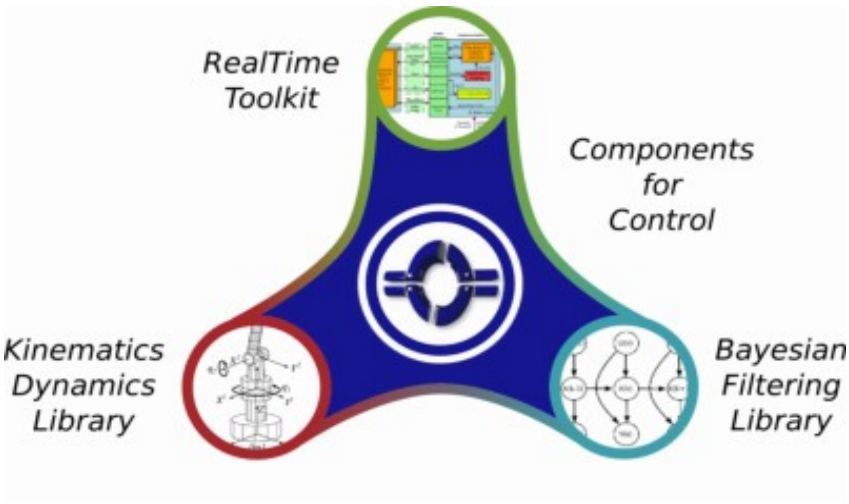

Ilustración 6: Esquema Orocos

Así pues, la arquitectura de Orocos es la siguiente:

- **Orocos Real Time Toolkit (RTT)**: No es una aplicación de Orocos, sino que provee la infraestructura y funcionalidades para la programación de aplicaciones C++. Como su nombre indica, pone enfasis en dotar de Tiempo Real (Hard Real Time) para las aplicaciones.

- **Orocos Components Library (OCL)** provee componentes de control preparados para usar. También es el encargado de dotar de acceso al Hardware.

- **Orocos Kinematics and Dynamics Library (KDL)**: es una librería C++ para calcular la Dinámica y Cinemática en Tiempo Real.

- **Orocos Bayesian Filtering Library (BFL)**: Provee al framework de los recursos necesarios para utilizar distintos filtros basados en Bayes, como Filtros de Partículas, Kalman, etc

-

2.3.3. ROS

ROS (Robotic Operating System) es un framework open source para el desarrollo de software de robots, proveyendo servicios de abstracción hardware, control de dispositivos a bajo nivel, mensajes entre procesos, control de paquetes e implementación de las librerías más utilizadas.[7]

Es por ello que se le conoce como "meta-sistema operativo" a pesar de que, realmente, no lo es. Desarrollado en 2007 originalmente por la Universidad de Stanford, desde 2008 se mantiene gracias

a la colaboración de más de 20 instituciones federadas bajo el instituto Willow Garage. Está licenciado bajo BSD, lo que permite plena libertad para uso comercial e investigador.

Ros está diseñado para facilitar la escritura de software robótico tanto para la industria como para la investigación, siendo sus principales objetivos los siguientes:

- Soportar la **programación de diferentes hosts** unidos en una topología punto a punto, lo que permite aprovechar los beneficios del diseño multi-proceso y multi-host. Gracias a ello, es posible diseñar un sistema de múltiples computadores conectados vía ethernet.

- **Multilenguaje**. Dado que hay programadores con distintas preferencias en cuanto al lenguaje de programación, soporta C++, Python, Octave, Lisp y otros.

- **Basado en herramienta**. Ros es muy complejo, así que se ha optado por una arquitectura de microkernel con muchas pequeñas herramientas en lugar de un diseño monolítico y grande. De esta forma, el código puede ser reaprovechado, es modular, fácilmente estructurable y acorde a la topología punto a punto.

- **Liviano**. Ros tiene una gran variedad de librerías que pueden utilizarse para un proyecto (o no) de forma que se cree un ejecutable bastante ligero.

- **Gratuito y Opensource**. Está distribuido bajo licencia BSD.

Estos objetivos podrían ser muy importantes para este TFM, ya que contamos con placas con recursos limitados, lo que hace especialmente importante que el software pueda estructurarse en distintos módulos ligeros que se comuniquen entre sí.

Es decir, permite dividir el software de un robot en distintos nodos, por lo que es especialmente interesante para crear distintos nodos controlados por cada una de las tarjetas en distintas partes del robot y comunicados entre sí por una red inalámbrica / ethernet. Por ejemplo, para un robot con distintas partes puede ser interesante que distintas placas se comuniquen entre sí para controlar el robot, así como distintos pequeños robots pueden ser controlados cada uno por una tarjeta, comunicándose entre ellos para la consecución de un objetivo.

Además de todo ello, ROS es rápido de instalar y poner a funcionar. Por ello, los desarrolladores de Orocos implementaron un stack (o paquete) de Ros, en el que estaba implementado Orocos y su integración. Es decir, han creado un paquete de Orocos que puede ser utilizado en Ros para

aprovechar tanto la simplicidad de Ros como sus herramientas (visualización, comunicación, etc) mientras que a la vez se utiliza Orocos para implementar el control.

Otros paquetes de Ros encontramos algunos de localización y mapeo simultáneo (Slam), percepción, etc y, concretamente, encontramos el stack de Orocos (Orocos toolchain) que se va a intentar integrar en el presente trabajo.

2.4. Algoritmo EDF

Uno de los objetivos del presente TFM es realizar una comparación objetiva entre las distintas tarjetas propuestas.

Para ello, podría utilizarse uno de los múltiples algoritmos de benchmark disponibles, pero se ha declinado esa opción por las siguientes razones:

- Muchos de estos algoritmos se basan en el tiempo de escritura a disco. En este caso, en el tiempo de escritura a SD, cosa muy variable dependiendo de la categoría y tamaño de la SD utilizada para el proyecto.

  Además, es una característica que, generalmente en proyectos de robótica, no se utiliza demasiado

- Debe integrarse con Orocos. Se propone Orocos como software de control para las tarjetas, por lo que el algoritmo de benchmark debe integrarse con Orocos.

  Generalmente no es complicado adaptar estos algoritmos

- Ya que el TFM se engloba dentro del Máster de Automática e Informática Industrial, sería interesante que el algoritmo de benchmarking esté relacionado con el resto del Máster.

Por ello, se ha preferido implementar un algoritmo de planificación de procesos, en este caso el algoritmo de planificación dinámico EDF [8], utilizado en Sistemas de Tiempo Real.

2.4.1. Descripción del algoritmo

El EDF coloca los distintos procesos en una cola de prioridades.En cada ciclo el algoritmo busca en la cola por el proceso con un menor tiempo de deadline y le asigna tiempo de procesador.

Es decir, las tareas cuyo plazo absoluto (tiempo límite) vence antes, son las más prioritarias, y dicha prioridad se asigna dinámicamente, en función el tiempo límite de cada tarea.

Una tarea es más prioritaria en cuanto más cercano está el instante en que finaliza su plazo de ejecución, o lo que es lo mismo, cuanto más cerca se encuentre de incumplir sus requisitos temporales.

Con esta política, una tarea se mantiene en ejecución hasta que se bloquee, finalice el trabajo correspondiente a la activación presente o sea expulsada por la activación de una nueva tarea con un plazo de finalización más cercano.

2.4.2. Factor de utilización

Los plazos están garantizados sí y sólo sí se cumple:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

Ilustración 7: Factor de utilización

donde: C_i es el tiempo de cómputo de la tarea en el peor caso y T_i es el periodo de la tarea.

El EDF es, por tanto, un algoritmo óptimo, ya que si una tarea es planificable, entonces EDF puede planificarla:

- U>1 ningún algoritmo puede planificar la tarea
- U<1 la tarea es planificable por EDF (y quizá por algún algoritmo más)

El algoritmo ha sido programado en C++, con modificaciones mínimas para cada una de las tarjetas (principalmente, relacionadas con el sistema de archivos para poder guardar los datos referentes a los tiempos de ejecución en ficheros de texto)

3. Desarrollo Práctico

3.1. Preparación de las Tarjetas

Como se ha comentado en apartados anteriores, para empezar a trabajar con estas tarjetas empotradas, el primer paso es la instalación del sistema operativo elegido en la tarjeta de memoria.

Una vez se tiene instalado el sistema operativo, lo único que hay que hacer es conectar los dispositivos de entrada/salida y comenzar a desarrollar.

En la figura 8, se pueden ver los periféricos que se han conectado para comenzar a desarrollar con la BeagleBone, aunque es similar en las 3 tarjetas. Notar que Beaglebone y Raspberry no tienen Wifi incorporado y que se aprovecha el HUB para alimentar a las tarjetas, eliminando un cable.

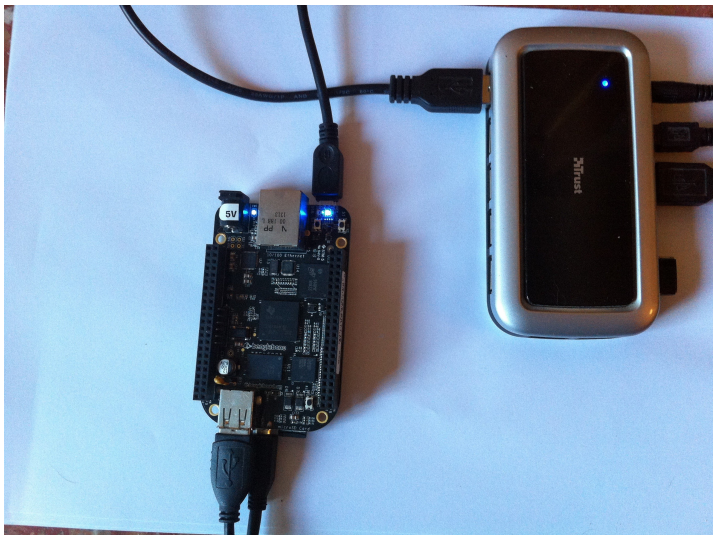

Ilustración 8: Conexión Beaglebone

Concretamente, para la Raspberry se ha utilizado una distribución "Raspbian" mientras que en Beaglebone/Igep una distribución "Ubuntu 12.04".

En ambos casos, se especifica el proceso a seguir para la preparación de las tarjetas de memoria, así como los distintos elementos utilizados. Al instalar el sistema operativo en cualquiera de estas tarjetas empotradas, se crean 2 particiones en la tarjeta: la partición boot que contiene los elementos necesarios para el arranque del sistema operativo y la partición rootfs, que contiene todo el sistema de archivos.

Destacar que la partición boot puede ser leída desde un sistema Windows (FAT32), mientras que la

partición rootfs (y también la boot) puede ser accedida desde Linux (ext4). Esto es algo bastante importante, ya que no se va a hacer uso, en ningún caso, de una interfaz gráfica de usuario, por lo que leer los archivos desde un sistema Linux en un PC otorga bastante comodidad a la hora de desarrollar software para cualquiera de las plataformas propuestas.

3.1.1. Instalación de Raspbian en la tarjeta Raspberry PI

Raspbian es un Sistema Operativo basado en Debian y optimizado para el hardware de Raspberry PI. Instalarlo no es demasiado complicado y puede hacerse, incluso, desde Windows.

Sólo hay que descargar Raspbian desde www.raspbian.org y cualquier programa para escribir una imagen de linux en una tarjeta SD o stick USB. En mi caso he utilizado Win32diskImager http://sourceforge.net/projects/win32diskimager/

Por supuesto, también es necesario una microSD con tamaño suficiente (aprox. 4GB) y puede ser necesario un lector de tarjetas para poder escribir en ella desde un PC.

Simplemente hay que seleccionar en Win32diskimager la tarjeta SD y la imagen a escribir y pulsar sobre Write, tal y como puede verse en la imagen siguiente:

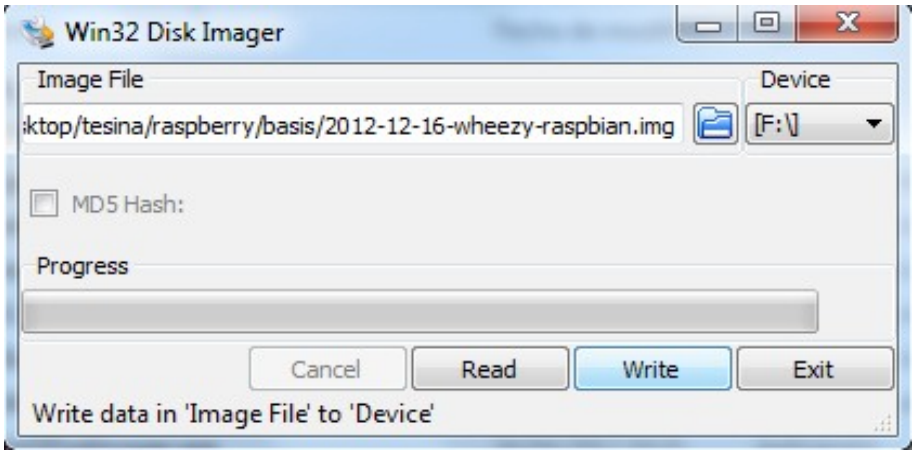

Ilustración 9: Win32diskImager

Win32diskimager ya se ocupa de particionar la tarjeta con las citadas boot y rootfs, así como instalar el sistema operativo y la estructura de ficheros del mismo.

3.1.2. Instalación de Ubuntu en la tarjeta BeagleBone

Para las distintas pruebas del presente trabajo he escogido la versión 12.04 de Ubuntu ya que, aunque no es la más nueva en el momento de escribir el documento, la versión 13.04 presenta problemas con algunas tarjetas y versiones de Orocos.

En primer lugar, hay que descargar la distribución escogida para instalar (en este caso, la Ubuntu 12,04). Concretamente, la comunidad de Ubuntu provee imágenes precompiladas para las tarjetas BeagleBone (con arquitectura "arm" y con soporte para operaciones en "hard float")

```
wget http://s3.armhf.com/debian/precise/bone/ubuntu-precise-12.04.2-armhf-3.8.13-bone20.img.xz
```

A continuación, haciendo uso de la cuenta de administrador se vuelca el archivo descargado a la tarjeta sd y se sincroniza.

```
sudo su
xz -cd ubuntu-precise-12.04.2-armhf-3.8.13-bone20.img.xz > /dev/sdb
partprobe /dev/sdb
sync
umount /dev/sdb
```

En este instante, la tarjeta SD ya cuenta con todo lo necesario para arrancar Ubuntu en la Beaglebone. Sin embargo, el espacio de la misma no estará completamente aprovechado, por lo que es recomendable extender la partición raíz.

En este momento, la SD cuenta con 2 particiones.

```
root@debian-armhf:/# fdisk /dev/sdb

Command (m for help): p



Device Boot      Start         End      Blocks   Id  System
/dev/mmcblk0p1   *        2048        4095        1024    1  FAT12
/dev/mmcblk0p2            4096     3751935     1873920   83  Linux
```

La partición número 2 puede ser eliminada para aprovechar el espacio. Pulsamos 'd' para eliminar una partición y seleccionamos la número 2

```
Command (m for help): d
Partition number (1-4): 2
```

Creamos una nueva partición pulsando 'n' para nueva partición, 'p' para partición primaria y creamos la partición 2. Dejamos los sectores inicial y final por defecto

```
Command (m for help): n
Partition type:
   p   primary (1 primary, 0 extended, 3 free)
   e   extended
Select (default p): p
Partition number (1-4, default 2): 2
First sector (4096-7710719, default 4096):
Using default value 4096
Last sector, +sectors or +size{K,M,G} (4096-7710719, default 7710719):
Using default value 7710719

Command (m for help): p

Disk /dev/mmcblk0: 3947 MB, 3947888640 bytes
4 heads, 16 sectors/track, 120480 cylinders, total 7710720 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x80000000

      Device Boot      Start         End      Blocks   Id  System
/dev/mmcblk0p1   *        2048        4095        1024    1  FAT12
/dev/mmcblk0p2            4096     7710719     3853312   83  Linux
```

Confirmamos cambios con 'w' y reiniciamos.

A continuación procedemos a expandir el espacio de la partición con resize2fs indicando la partición adecuada de nuestro sistema

```
root@debian-armhf:/# resize2fs /dev/sdb2
resize2fs 1.42.5 (29-Jul-2012)
Filesystem at /dev/mmcblk0p2 is mounted on /; on-line resizing required
old_desc_blocks = 1, new_desc_blocks = 1
```

Y podemos ver los cambios con df

```
root@debian-armhf:/# df
Filesystem     1K-blocks   Used Available Use% Mounted on
rootfs           3761680 741096   2851404  21% /
/dev/root        3761680 741096   2851404  21% /
devtmpfs          253920      0    253920   0% /dev
tmpfs              50816    216     50600   1% /run
tmpfs               5120      0      5120   0% /run/lock
tmpfs             101620      0    101620   0% /run/shm
/dev/mmcblk0p1      1004    474       530  48% /boot/uboot
```

Cabe destacar que el proceso para extender la partición del sistema de ficheros (y así tener más espacio de almacenamiento) es una etapa esencial. Además, en caso de tener un PC con Linux con interfaz gráfica, haciendo uso de herramientas gráficas como “Gparted” es posible llegar al mismo destino que con la terminal de una forma más intuitiva.

3.1.3. Conexión a la red en Linux

Una de las características principales de los sistemas operativos diseñados para tarjetas embebidas es que, por regla general, no existe una interfaz gráfica, por lo que el acceso a ellas se realiza mediante la terminal.

Es muy frecuente acceder a estas tarjetas a través de la red mediante SSH (Secure Shell), por lo que no es necesario que estén conectadas a un monitor. Es decir, con estar conectadas a la red, mediante otro equipo conectado a esa misma red, es posible trabajar sobre ellas de forma muy cómoda.

La forma más sencilla de conectar a la red una tarjeta embebida es mediante una conexión con un

cable de red RJ45. Generalmente, el fichero de configuración que maneja las interfaces de red (/etc/network/interfaces) está a un valor por defecto en el que por DHCP, el router de la red asigna una dirección IP válida a la tarjeta empotrada.

En caso de disponer un adaptador Wifi (explicado en apartado anteriores) para la tarjeta empotrada, o que la propia tarjeta posea la conexión Wifi (como la Igep, por ejemplo), es necesario modificar el fichero comentado anteriormente para indicar que se dispone de una interfaz inalámbrica, que se quiere conectar a una red concreta y con una contraseña concreta.

Para editar el fichero "interfaces", se puede realizar de la siguiente manera:

```
sudo nano /etc/network/interfaces
```

Y escribiendo en el mismo la configuración SSID

```
iface wlan0 inet dhcp
  wpa-ssid MISSID
  wpa-psk MISSIDPASSWORD
```

Finalmente, para que se conecte a la red es necesario reiniciar el adaptador. Puede hacerse reiniciando la tarjeta (sudo reboot) o mediante los comandos

```
sudo ifdown wlan0
sudo ifup wlan0
```

De esta forma se consigue conectar a la red, ya sea por conexión cableada o inalámbrica. Esto ofrece la posibilidad de instalar tarjetas en cualquier lugar con conexión a la red, poder acceder a ellas mediante SSH y trabajar sobre ellas cómodamente, así como crear una red de tarjetas que comuniquen entre sí.

3.2. Instalación

A continuación se describe el proceso necesario para la instalación de Orocos, Ros y Ros con Orocos integrado (donde ha sido posible), así como la creación de componentes en cada una de las tarjetas.

3.2.1. Raspberry PI

En primer lugar, se parte de una Raspberry PI, con Raspbian Wheezy instalado de la manera explicada anteriormente. Como se sabe, el mayor inconveniente de Raspberry Pi es su procesador. La arquitectura ARM11 de su procesador fue anunciada en el año 2002 y diversas comunidades, entre las que destaca la comunidad Ubuntu, han dejado de desarrollar código para este tipo de procesador. Por tanto, la comunidad de Ubuntu decidió no dar soporte a Raspberry [9]

3.2.1.1. Ros

La instalación de Ros en Raspberry no es demasiado complicada, puesto que el paquete de ROS está incluido en los repositorios de Debian. Por ello, sólo hay que añadir la dirección de los repositorios a la lista, actualizar apt-get e instalar. Las órdenes empleadas se pueden ver a continuación.

Añadir el repositorio y la key.

```
$ sudo sh -c 'echo "deb http://64.91.227.57/repos/rospbian wheezy main" >
/etc/apt/sources.list.d/rospbian.list'

$ wget http://64.91.227.57/repos/rospbian.key -O - | sudo apt-key add -
```

Actualizar los repositorios.

```
$ sudo apt-get update
```

Instalar Ros

```
$ sudo apt-get install ros-groovy-ros-comm
```

En este momento, sólo hay que teclear "roscore" en una terminal para lanzar el núcleo de ROS y poder comenzar a utilizar todas las herramientas que provee.

3.2.1.2. Orocos

Instalar Orocos en una Raspberry Pi es mucho más complejo que ROS, y esta puede realizarse mediante compilación cruzada (desde un PC con Linux, compilarlo para la arquitectura específica ARM) o nativa (desde la propia tarjeta empotrada con su propio compilador).

Por lo que se refiere a la compilación nativa, es más sencilla que la cruzada, pero debido a las limitadas características del hardware, es un proceso que puede durar varias horas.

La Raspberry tiene poca memoria RAM, a pesar de que para el presente trabajo se ha utilizado el modelo B de 512 MB RAM, pero compartido de todos modos con la GPU.

Por ello, es necesario crear una partición (o fichero) SWAP o de intercambio en la Raspberry que permitirá la instalación de la orocos_toolchain. El principal inconveniente de tener poca RAM es que si no se crea una partición (o fichero) de intercambio, se produce un desbordamiento de pila en el proceso de compilación, por lo que es imposible instalar Orocos.

Para habilitar un fichero de intercambio, hay que crear un fichero vacío en la tarjeta SD

```
sudo dd if=/dev/zero of=/home/swap bs=1024 count=524288;
```

Tras ello, hay que proceder a indicar al sistema operativo que se ha creado un fichero de intercambio y que se quiere utilizar.

```
sudo mkswap /home/swap;
sudo chmod go-rwx /home/swap;
sudo swapon /home/swap;
```

Se puede comprobar con el comando “free” que el fichero de intercambio a sido creado correctamente.

```
             total       used       free     shared    buffers     cached
Mem:        389112      77896     311216          0       5192      29600
-/+ buffers/cache:      43104     346008
Swap:       524288          0     524288
```

Una vez se ha creado este fichero de intercambio correctamente, es necesario instalar en la Raspberry las siguientes dependencias, las cuales se van a utilizar en todo el proceso de instalación de Orocos:

```
sudo apt-get install libreadline-dev omniorb omniidl omniorb-nameserver libomniorb4-1 libomniorb4-dev libomnithread3-dev libomnithread3c2 gccxml antlr libantlr-dev libxslt1-dev liblua5.1-0-dev ruby1.8-dev libruby1.8 rubygems1.8 libcppunit-dev
```

Para la instalación de Orocos, los propios desarrolladores proveen un script de instalación, el cual hay que descargarlo y ejecutarlo en la carpeta de instalación deseada

```
wget http://gitorious.org/orocos-toolchain/build/raw/toolchain-2.6:bootstrap.sh
```

Al ejecutarse el bootstrap, de forma automática se creará el sistema de archivos y carpetas necesario para la instalación de la orocos toolchain.

```
sh bootstrap.sh
```

La orocos_toolchain tiene un paquete llamado orogen que no es posible compilar para ARM. Por ello, hay que deshabilitar la compilación del componente.

Esto no supone ningún problema, ya que Orogen es una herramienta de la Orocos RTT que permite crear nuevos componentes. Simplemente, es un generador de código intermedio que permite generar automáticamente el código C++ de un componente a partir de las especificaciones que el programador introduce.

No poder compilar Orogen nos hace perder una herramienta útil, pero en absoluto ninguna funcionalidad del proyecto Orocos.

Para ello, dentro de las carpetas generadas por el bootstrap, hay que modificar el fichero autoproj/manifest y comentar la compilación de Orogen.

```
layout:
- typelib
- utilrb
- utilmm
- log4cpp
- rtt
- rtt_typelib
# - orogen
- ocl
```

Para compilar orocos, cargamos las variables de entorno en el bash mediante:

```
source env.sh
```

Finalmente, el último paso para compilar e instalar Orocos es el siguiente:

```
autoproj update
autoproj build
```

Como se ha comentado anteriormente, puesto que la compilación e instalación se ha hecho en la propia tarjeta, debido a su capacidad de cómputo, es un proceso que puede durar varias horas (alrededor de 8 horas de media).

Finalmente, si todo ha funcionado correctamente, se puede comprobar la instalación, por ejemplo, lanzando el deployer:

```
deployer-gnulinux
```

3.2.1.3. Instalación Ros+Orocos integrado

Desafortunadamente, no ha sido posible instalar Ros con Orocos integrado. Sin embargo, el proceso que se ha seguido es interesante y puede servir de base para investigaciones futuras, por lo que se adjunta en este trabajo.

Al igual que en el proceso anterior, es necesario instalar las siguientes dependencias y crear un swap mayor.

```
sudo apt-get install git-core
sudo apt-get install subversion autoconf automake python python-dev
sudo apt-get install rubygems1.8 libxslt1-dev;
sudo apt-get install ruby;
sudo apt-get install libxerces-c2-dev libxerces-c28 libxerces-c3.1;
sudo apt-get install omniorb;
sudo apt-get install libreadline-dev omniorb omniidl omniorb-nameserver libomniorb4-1 libomniorb4-dev
libomnithread3-dev libomnithread3c2 gccxml antlr libantlr-dev libxslt1-dev liblua5.1-0-dev ruby1.8-dev
libruby1.8 rubygems1.8 libcppunit-dev
```

El siguiente paso es descargar los paquetes para la instalación de orocos en la carpeta que se pretenda instalar.

```
git clone --recursive git://gitorious.org/orocos-toolchain/orocos_toolchain.git
git clone http://git.mech.kuleuven.be/robotics/rtt_ros_integration.git
git clone http://git.mech.kuleuven.be/robotics/rtt_ros_comm.git
git clone http://git.mech.kuleuven.be/robotics/rtt_common_msgs.git
git clone http://git.mech.kuleuven.be/robotics/rtt_geometry.git
```

Para mayor facilidad, se puede hacer uso de la herramienta distcc.

Distcc es una herramienta de compilación distribuida cuya función es poder utilizar varias máquinas en red para compilar un trabajo considerable. Pero también puede ser utilizada para compilar de forma cruzada un trabajo, lanzando todos los comandos desde otra máquina (como puede ser la RaspBerry) de forma transparente y muy sencilla.

Es necesario instalar distcc tanto en la raspberry como en el pc. En el PC es muy sencillo, ya que se instala con el siguiente comando.

```
sudo apt-get install distcc
```

En la Raspberry, es necesario descargalo, compilarlo e instalarlo

```
svn checkout http://distcc.googlecode.com/svn/trunk/ distcc-read-only
```

Y dentro de la carpeta creada distcc-read-only

```
./autogen.sh
./configure --with-gtk && make && sudo make install
```

Ahora podemos utilizar un PC para realizar la compilación. Pero antes, es necesario instalar en el PC la toolchain para la compilación cruzada

```
git clone https://github.com/raspberrypi/tools.git --depth=1
```

Esto descargará la toolchain oficial proporcionada por la Fundación Raspberry. Para que la compilación sea completamente transparente, podemos ejecutar los siguientes comandos que crearán los enlaces simbólicos necesarios para la compilación cruzada, ya que apuntarán a los compilados de ARM.

```
cd tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/
```

```
~/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin$ ln -s arm-linux-gnueabihf-gcc gcc
~/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin$ ln -s arm-linux-gnueabihf-gcc cc
~/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin$ ln -s arm-linux-gnueabihf-c++ c++
~/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin$ ln -s arm-linux-gnueabihf-cpp cpp
export PATH=$HOME/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin:$PATH
```

Si se ejecuta el comando

```
which gcc
```

devuelve el compilador que se utilizará al ejecutar gcc y que será el siguiente

```
arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/gcc
```

Es importante esto ya que desde la Rasbperry vamos a utilizar el rosmake. Así que para no tener que editar ningún fichero de compilación, los enlaces simbólicos utilizarán el compilador apropiado.

En la raspberry, se va a usar un proceso similar que nos permita utilizar el distcc en lugar del compilador por defecto:

Averiguamos primero dónde está instalado gcc y dónde está distcc

```
pi@raspberrypi ~ $ which gcc
/usr/bin/gcc
pi@raspberrypi ~ $ which distcc
/usr/local/bin/distcc
```

Y ahora creamos los enlaces simbólicos apropiados para que se ejecute distcc en lugar de los compiladores por defecto:

```
pi@raspberrypi ~ $ sudo ln -s /usr/local/bin/distcc /usr/local/bin/gcc
pi@raspberrypi ~ $ sudo ln -s /usr/local/bin/distcc /usr/local/bin/cc
pi@raspberrypi ~ $ sudo ln -s /usr/local/bin/distcc /usr/local/bin/g++
pi@raspberrypi ~ $ sudo ln -s /usr/local/bin/distcc /usr/local/bin/c++
pi@raspberrypi ~ $ sudo ln -s /usr/local/bin/distcc /usr/local/bin/cpp
```

Y añadimos /usr/local/bin al path.

```
export PATH=/usr/local/bin:$PATH
```

Así, y desde este momento, cuando se llame a gcc, g++, etc desde la Raspberry, utilizaremos distcc en su lugar.

Una vez listo, necesitamos una serie de variables de entorno para saber qué nodo va a compilar cuando la Rasbperry lo necesite.

Es decir, hay que indicar la IP del PC donde se ha instalado distcc. También es necesario configurar el timeout y otros valores, se recomienda dejar los siguientes por defecto.

```
export DISTCC_HOSTS="192.168.1.20/16" #ip del pc
export DISTCC_BACKOFF_PERIOD=0
export DISTCC_IO_TIMEOUT=3000
export DISTCC_SKIP_LOCAL_RETRY=1
```

Para finalizar, se ejecuta en el PC distcc, permitiendo la IP de la raspberry de la siguiente forma.

```
distccd --daemon --jobs 16 --allow 192.168.1.42 --verbose --log-stderr --no-detach
```

Finalmente pasamos al proceso de instalación de ros+orocos. Para la instalación de orocos hay que añadir al PATH de paquetes de ROS los paquetes de la orocos toolchain.

```
export ROS_PACKAGE_PATH=/path/to/orocos_toolchain:${ROS_PACKAGE_PATH}
export ROS_PACKAGE_PATH=/path/to/rtt_ros_integration:${ROS_PACKAGE_PATH}
export ROS_PACKAGE_PATH=/path/to/rtt_ros_comm::${ROS_PACKAGE_PATH}
export ROS_PACKAGE_PATH=/path/to/rtt_ros_common_msgs::${ROS_PACKAGE_PATH}
export ROS_PACKAGE_PATH=/path/to/rtt_geometry::${ROS_PACKAGE_PATH}
```

No es posible compilar el paquete orogen de la orocos toolchain. En principio, este paquete sólo sirve para generar código intermedio para la ceación de módulos de Orocos y no impide el correcto funcionamiento de orocos.

Por ello, puede suprimirse su compilación añadiéndolo a la lista negra del compilador colocando un fichero con el nombre ROS_BUILD_BLACKLIST dentro de la carpeta orocos_toolchain/orogen.

Y una vez hecho esto, pasamos a compilar con rosmake de la siguiente forma.

```
rosmake orocos_toolchain rtt_ros_integration rtt_ros_comm rtt_ros_common_msgs rtt_geometry --skip-blacklist
```

Desgraciadamente, aunque el proceso finaliza con ningún error ni advertencia, la instalación no es funcional.

Al intentar arrancar el deployer de Orocos, la Raspberry se queda congelada y hay que reiniciarla.

Es posible que el procesador ARM11 de Raspberry no sea compatible con la instalación de Orocos+Ros, ya que esta se ha realizado varias veces sin llegar nunca a poder arrancar.

3.2.2. Beaglebone/Igep

A continuación se describe el proceso para instalar el software en las tarjetas Beaglebone e Igep. Se ha agrupado en un único apartado ya que el procedimiento a seguir en las 2 tarjetas es muy similar.

3.2.2.1. Orocos

El proceso para instalar Orocos toolchain en Beaglebone/Igep es exactamente igual al seguido en Raspberry y descrito en el apartado anterior. Esto es debido a que, puesto que Ubuntu está basado en Debian, , por lo que no es necesario volver a describirlo aquí.

3.2.2.2. Ros

La instalación de Ros en Beaglebone/Igep es similar al seguido en Raspberry, sólo cambian los repositorios.

Así pues, primeramente es necesario añadir el repositorio y la key.

```
sudo sh -c 'echo "deb http://packages.ros.org/ahendrix-mirror/ubuntu precise main"
>/etc/apt/sources.list.d/ros-latest.list'
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

Actualizar los repositorios.

```
$ sudo apt-get update
```

Instalar Ros

```
$ sudo apt-get install ros-groovy-ros
```

En este momento, sólo hay que teclear "roscore" en una terminal para lanzar el núcleo de ROS y poder comenzar a utilizar todas las herramientas que provee.

3.2.2.3. Orocos+Ros

Al contrario que con la Raspberry, el ARM de la beaglebone y de Igep sí que está soportado por la comunidad por lo que encontramos binarios listos para la instalación tanto de Ros como de Orocos, de una forma muy sencilla.

Primero, es preciso añadir a la lista de repositorios el repositorio donde podemos encontrar Ros groovy y validarlo

```
sudo sh -c 'echo "deb http://packages.ros.org/ahendrix-mirror/ubuntu precise main" >
/etc/apt/sources.list.d/ros-latest.list'
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

A continuación, actualizamos apt-get

```
sudo apt-get update
```

E instalamos Ros

```
sudo apt-get install ros-groovy-ros
```

Una vez instalado Ros, instalamos la orocos toolchain

```
sudo apt-get install ros-groovy-orocos-toolchain ros-groovy-rtt-ros-integration ros-groovy-rtt-geometry ros-
groovy-rtt-ros-comm ros-groovy-rtt-common-msgs
```

3.2.3. Creación de un componente en la Raspberry

Una vez instalado Orocos en la Raspberry, se procede a crear un componente. Primeramente, es necesario añadir las variables de entorno y path correspondientes mediante.

```
orocos_toolchain/source env.sh
```

Si no está instalado cmake, instalarlo mediante

```
apt-get install cmake-curses-gui libxml-xpath-perl
```

Y creamos la estructura del componente con el nombre deseado, en mi caso, "prueba".

```
orocreate-pkg prueba
```

Ahora se ha creado una estructura de componente Orocos que podemos manipular, antes de nada, crear una carpeta build

```
mkdir build
cd build
```

Y crear el componente mediante

```
ccmake ..
```

Aparece un menú de configuración del make, hay que pulsar las siguientes opciones:

```
--> letra 'c' para configurar
--> letra 'e' para salir
```

No debe salir ningún error, en la primera línea CMAKE_BUILD_TYPE hay que editar las opciones y escribir.

```
RelWithDebInfo
```

Y de nuevo pulsamos la secuencia.

```
--> letra 'c' para configurar
--> letra 'e' para salir
--> letra 'g' para generar el componente
```

```
cmake..
cd ..
```

Y en la carpeta raíz del componente, editamos el fichero CMakeLists.txt eliminando las cabeceras siguientes.

```
# orocos_typegen_headers(src/prueba-types.hpp)
```

En este momento ya podemos hacer 'make' para compilar el componente generando el ejecutable.

Sin embargo, este componente aún no haría nada, ya que se trata de los fuentes creados por defecto.

Dentro de la carpeta prueba/src encontramos los fuentes. Podemos sustituir los ficheros por los adjuntos para crear el componente de Benchmarking o cualquier otro componente deseado.

Una vez modificado el código, ejecutar 'make' dentro de la carpeta build para generar el ejecutable, que se llamará (en este caso) libprueba-gnulinux.so y estará en la carpeta prueba/build/src

Hay que copiar el componente a la carpeta de la orocos_toolchain, concretamente a la subcarpeta ocl, ya que esta carpeta es la carpeta donde, por defecto, el deployer trata de encontrar los componentes implementados para cargarlos.

En caso de no copiar un componente a dicha carpeta, el deployer será incapaz de encontrarlo y, por lo tanto, al intentar ejecutarlo obtendremos un error de componente inexistente.

```
cp libprueba-gnulinux.so /home/pi/orocos-toolchain/install/lib/orocos/gnulinux/ocl/
```

Ejecutamos orocos mediante.

```
deployer-gnulinux
```

Y configuramos y cargamos el componente mediante.

```
->loadComponent("a","Prueba")  #para cargar el componente
->a.setPeriod(0.1)              #para fijar el periodo deseado para la ejecución del componente
->a.configure                   #para guardar la configuración deseada
->a.start                       #para iniciar
```

3.2.4. Creación de un componente en la BeagleBone/Igep

En beaglebone/igep se ha instalado ROS+Orocos integrado, con lo que para la creación de un componente, el camino a seguir es algo distinto, puesto que se va a usar el comando "rosmake", particular de ROS.

Se describe a continuación el proceso completo

Primeramente, es necesario usar un ejecutable de ros para crear un espacio de trabajo, pero no está incluido en la instalación por defecto de ros, así que se instala mediante

```
sudo apt-get install python-rosinstall
```

Y ahora cargamos el entorno de Ros y de Orocos mediante

```
source /opt/ros/groovy/setup.sh
source /opt/ros/groovy/stacks/orocos_toolchain/env.sh
```

Es recomendable cargar el entorno de Ros

```
sudo rosdep init
sudo rosdep update
```

A continuación, se procede a crear un directorio de trabajo (workspace). En mi caso, lo he creado dentro del entorno de usuario por defecto

```
rosws init ~/groovy_workspace /opt/ros/groovy
source /home/ubuntu/groovy_workspace/setup.bash
```

Y una carpeta sandbox donde se crean los componentes

```
mkdir ~/groovy_workspace/sandbox
```

Se crea ahora la estructura del componente dentro del sandbox. De nuevo, en mi caso se llamará prueba

```
rosrun ocl orocreate-pkg prueba
```

Y dentro de ~/groovy_workspace/sandbox/prueba/src encontramos los fuentes del componente que podemos modificar para que realice la tarea deseada. Una vez creada la funcionalidad del componente, se compila mediante

```
rosmake -s prueba
```

Lo cual crea el ejecutable que se deberá importar desde el deployer de Orocos.

En este momento, se puede ejecutar orocos

```
rosrun ocl deployer-gnulinux
```

Y a continuación el componente

```
->import("Prueba")                 #para importar el componente
->loadComponent("a","Prueba")  #para cargar el componente
->a.setPeriod(0.1)                  #para fijar el periodo deseado para la ejecución del componente
->a.configure                       #para guardar la configuración deseada
->a.start                           #para iniciar
```

4. Benchmarking

Para comparar las 3 placas entre sí se ha desarrollado un pequeño algoritmo como algoritmo de benchmarking.

Para continuar con la temática del máster, se ha escogido un algoritmo de planificación de procesos.

Normalmente, para realizar tareas de benchmarking hay algoritmos específicos en los que se realizan multitud de operaciones de cálculo (inversión de matrices, operaciones logarítmicas, potencias...) pero he preferido desarrollar un algoritmo más afín a la temática del Máster.

Por ello, he realizado un planificador de tareas EDF, ya que hay comparaciones y distintas operaciones matemáticas.

Se ha optado por desarrollar un algoritmo ya que el objetivo es testar estas tarjetas funcionando con Orocos y realizando cálculos relacionados con el control, dejando de lado otros aspectos que programas de benchmark comerciales tienen en cuenta, como por ejemplo tiempos de escritura en disco.

4.1. Benchmarking con 1 único proceso

Para realizar la comparativa, se ha utilizado el algoritmo anterior programado en C++ e implementado como un único componente Orocos y una batería de datos con la que poder trabajar.

Concretamente, se han introducido 100 tareas con una utilización total de 0.97 (es decir, planificable).

De esta forma, y a pesar de que el algoritmo tiene un coste $O(n^2)$, cuenta con bastantes datos de forma que el tiempo que las tarjetas tardan en resolverlo es significativo.

En Orocos, un componente puede configurarse para reaccionar ante eventos o para que se ejecute en un periodo determinado, fijado de forma estricta por el sistema operativo.

Para el benchmarking se ha preferido trabajar para que el componente se ejecute con un periodo determinado, ya que si dependiese de un determinado evento, sería más complicado analizar la información recibida y obtener conclusiones.

Para tener distintas medidas, se ha procedido a simular la planificación del procesador en 1 ciclos hasta 500, obteniendo para cada tarjeta 500 mediciones.

Los resultados han sido los siguientes:

4.1.1. Raspberry

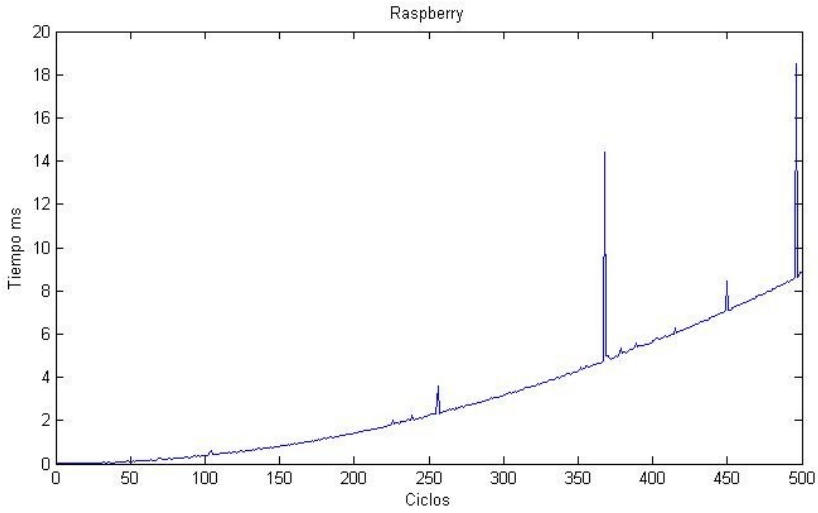

Ilustración 10: Tiempos Raspberry 1 componente

Dado que se han realizado distintas simulaciones con un crecimiento del procesamiento de forma cuadrático, es lógico que el tiempo de procesamiento también crezca.

En la Raspberry podemos ver ese crecimiento del tiempo de cómputo, con algunos picos. El tiempo máximo se sitúa en los 18.2 ms y el tiempo medio en 2,99 ms

También se ha realizado un overclock en la Raspberry como se ha descrito en el Anexo.

Este overclock logra que el procesador trabaje a 1GHz (una velocidad 42.8% superior a la normal) así como una frecuencia ligeramente superior de acceso a RAM (450MHz en lugar de 400MHz).

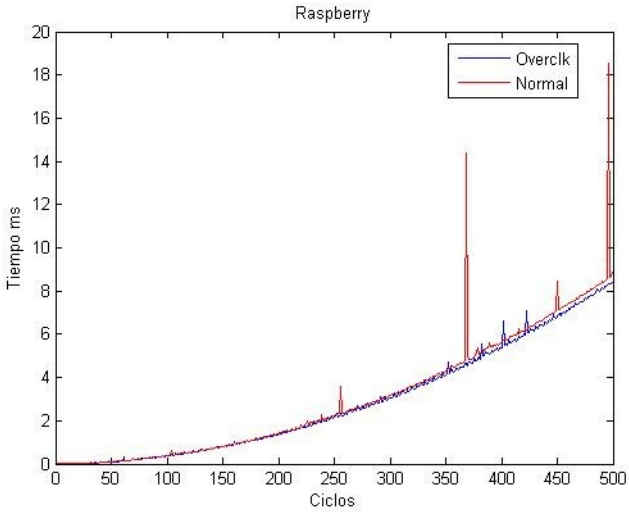

Ilustración 11: Raspberry vs Overclock

Este trabajo extra (que supone una sobrealimentación de 6 voltios) debería significar una mejora del tiempo significativa, sin embargo se observa en la gráfica comparativa que, excepto en algunos picos, el tiempo de cómputo no mejora apenas, ya que se sitúa en los 2.86 ms, lo que supone una mejora de un 4.4%

Por ello, podemos intuir que no sólo estamos trabajando con la CPU.

4.1.2. BeagleBone

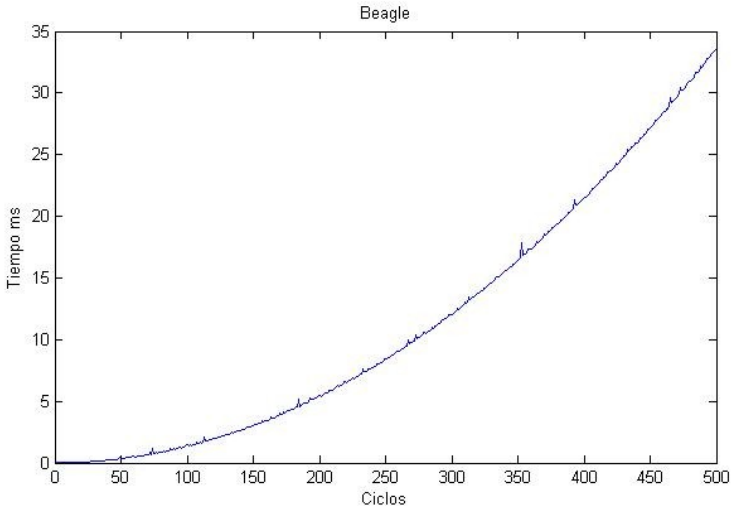

Ilustración 12: Tiempos Beaglebone 1 componente

En la Beaglebone podemos observar menos picos que en Raspberry, con un tiempo máximo de 33 ms y un tiempo medio de 11.25 ms, significativamente peor.

Raspberry es la única de las 3 tarjetas que cuenta con un sistema operativo hecho a medida, por lo que probablemente estamos sacando más partido al hardware que en las otras tarjetas.

4.1.3. Igep

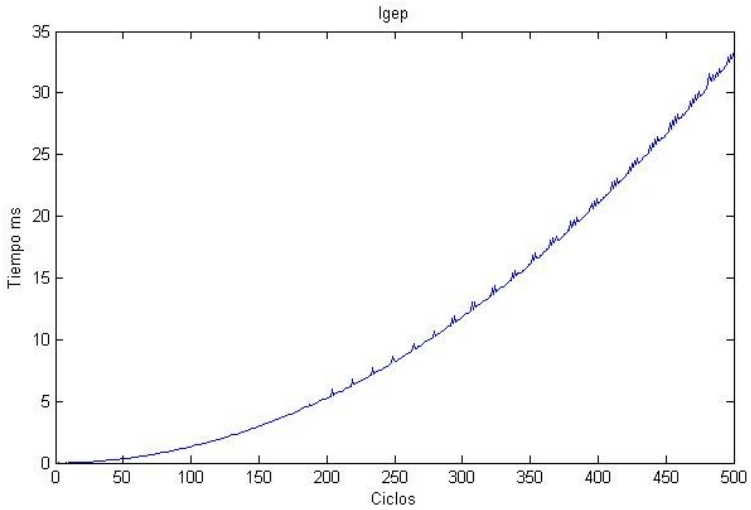

Ilustración 13: Tiempos Igep 1 componente

En la Igep, a primera vista, observamos una gráfica muy similar a la Beaglebone, de hecho obtenemos un tiempo medio de 11.14 ms. Como ya se suponía por la documentación, Igep es un clon de Beaglebone.

4.2. Benchmarking procesamiento distribuido

En la descripción de Orocos se ha descrito su potencial para computación distribuida, ya que es muy sencillo crear una serie de nodos, comunicarlos entre sí y lanzarlos para ejecutar una aplicación.

Por ello, se procede a realizar un Benchmarking similar al anterior, pero con distintos nodos trabajando en paralelo.

Concretamente, hay 4 nodos de ejecución del algoritmo EDF, realizando cada uno de ellos una planificación independiente y un nodo supervisor encargado de tomar datos y almacenarlos en un fichero.

Concretamente, en cada ciclo lanza los 4 nodos EDF, toma el tiempo de inicio más temprano y espera a que finalicen los 4 nodos, tomando el tiempo de finalización más tardío.

Con estos 2 tiempos, se puede analizar cuánto tiempo ha tomado Orocos para ejecutar los nodos.

4.2.1. Raspberry

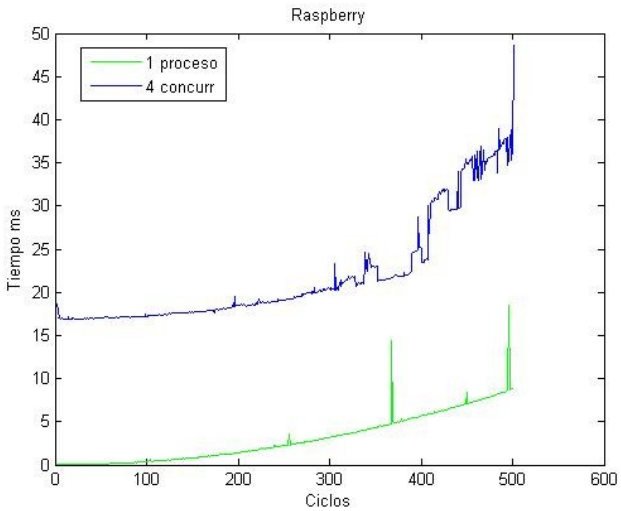

Ilustración 14: Tiempos Raspberry 4 componentes

Al ejecutar 4 procesos concurrentes, el tiempo obviamente empeora de forma significativa.

De hecho, el tiempo peor se sitúa en 48.5 ms y el tiempo medio en 21.9 ms, bastante más alto que los tiempos de 18.2 ms (máximo) y 2.99 ms (tiempo medio) para un único proceso.

Es lógico ya que contamos con un único núcleo, por lo que al realizar las 4 tareas concurrentes, estas se ejecutan en un único procesador.

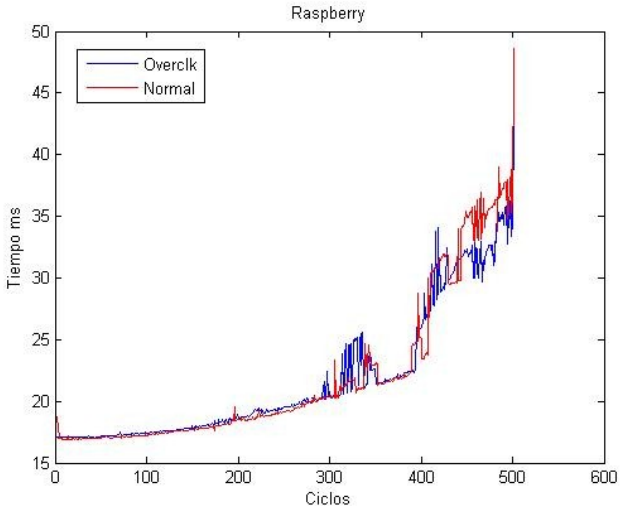

Ilustración 15: Tiempos Raspberry Oclk 4 componentes

Al realizar un overclock a la Raspberry, de forma que de nuevo el procesador trabaje a 1GHz (un 42.8% superior) vemos que el tiempo que se toma es muy similar en promedio al caso anterior

(21.7 ms en overclk contra 21.9 ms sin overclk) habiendo una mejora algo más significativa en los picos, ya que en este caso tenemos un tiempo peor de 43.57 ms.

En todo caso, y al igual que en 1 único proceso, la mejora no es significativa.

4.2.2. Beaglebone

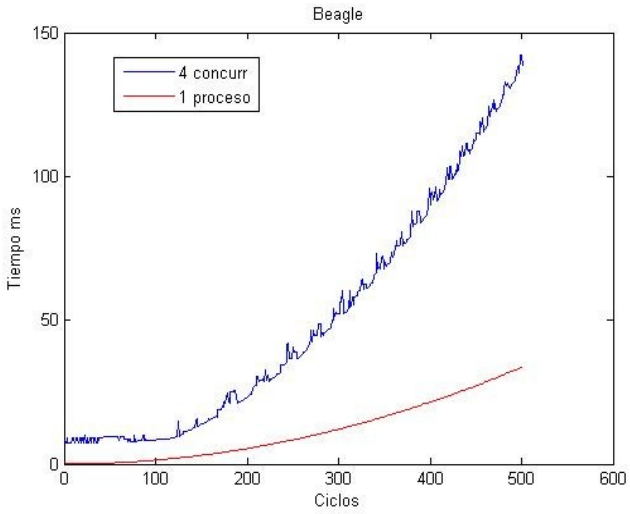

Ilustración 16: Tiempos Beaglebone 4 componentes

En Beaglebone tenemos un tiempo medio de ejecución de 49.7 ms , mientras que el tiempo máximo es de 142.2 ms, contra los tiempos medio de 11.25 ms y máximo de 33 ms obtenidos para 1 único proceso.

4.2.3. Igep

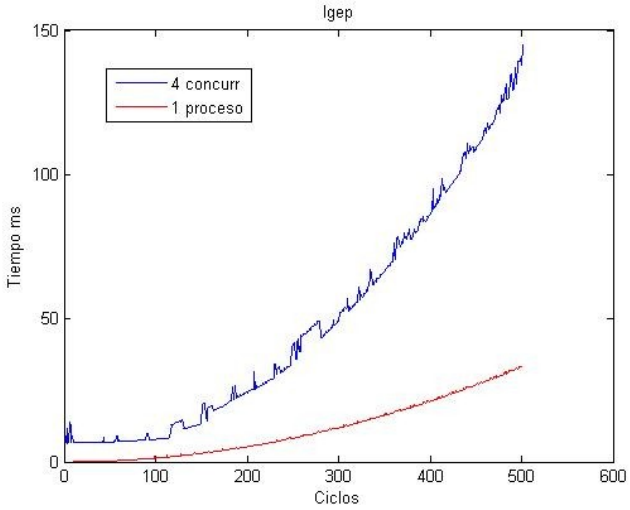

Ilustración 17: Tiempos Igep 4 componentes

Igep, de nuevo también en concurrente, tiempos muy similares a BeagleBone. Tiempo medio de 48.62 ms y tiempo máximo de 139.89 ms.

4.3. Análisis de resultados

Los resultados en Igep y BeagleBone son muy similares, si se grafican 2 a 2 se obtiene lo siguiente:

Procesando un único componente de Orocos:

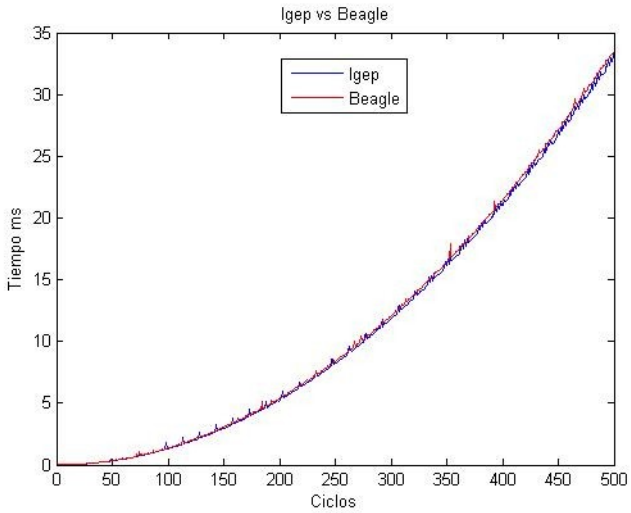

Ilustración 18: Igep vs Beaglebone 1 componente

Realizando el procesamiento de 4 componentes concurrentes:

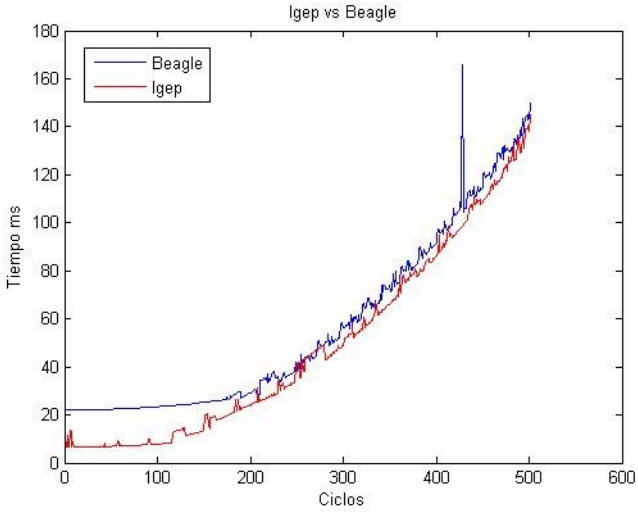

Ilustración 19: Igep vs BeagleBone 4 componentes

Se puede observar que, al hablar de Igep y Beaglebone, estamos hablando de tarjetas prácticamente clónicas, algo que ya intuíamos y además aunque el precio de la beaglebone es 4 veces inferior a la de Igep. Eso sí, debemos recordar que la Igep está adaptada para entorno industrial, con lo cual hay

que valorar qué tarjeta conviene dependiendo del entorno en el que se vaya a desarrollar la actividad.

Su rendimiento es muy similar, aún contando con precios muy diferentes.

Sin embargo, el resultado que realmente ha llamado la atención es que la Raspberry, en todas las pruebas, ha tenido unos resultados mejores que el resto de tarjetas que, aunque ahora sabemos que son prácticamente clónicas, eran, sobre el papel, bastante superiores.

Si se analizan los resultados en gráficas comparativas se obtiene lo siguiente:

Para un único proceso:

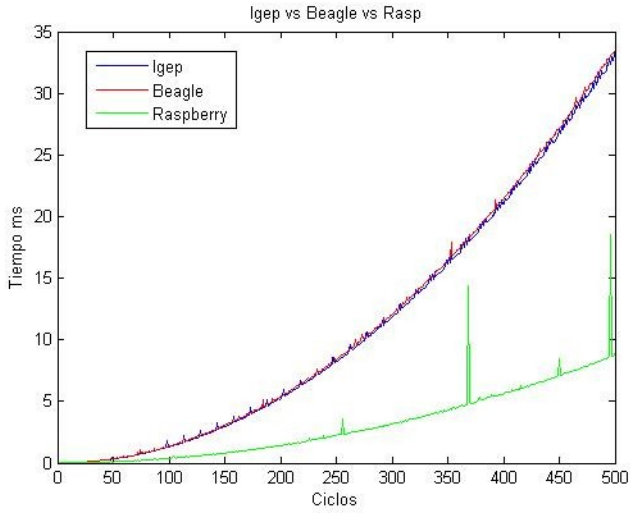

Ilustración 20: Igep vs Beaglebone vs Raspberry 1 componente

Para 4 procesos concurrentes:

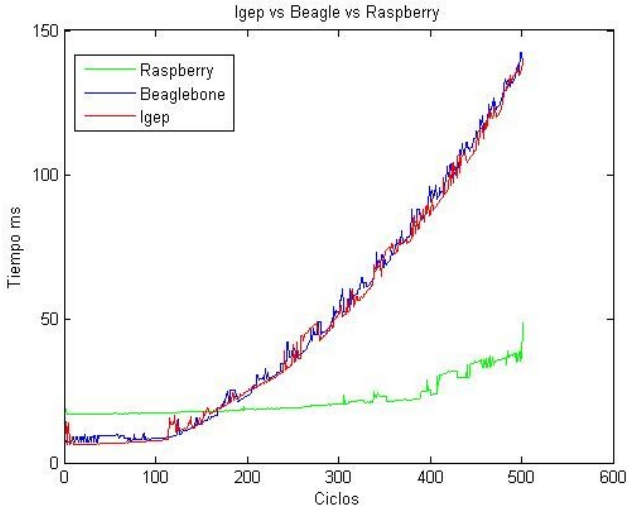

Ilustración 21: Igep vs Beagle vs Raspberry 4 componentes

Si se muestran los datos en forma de tabla:

	1 único proceso		4 procesos concurrentes	
	Tiempo medio	Tiempo máximo	Tiempo medio	Tiempo máximo
Raspberry	2,99	18,2	21,9	48,5
Beaglebone	11,25	33	49,7	142,2
Igep	11,14	32,8	48,62	139,89

Se observa que, tanto para un único proceso como para 4 procesos ejecutándose de forma concurrente, los tiempos medios y máximos son mucho mejores en Raspberry que en Beagle o Igep.

La única diferencia en la instalación de Orocos ha sido que en Igep y Beaglebone se ha instalado junto a Ros, debido a las facilidades que Ros aporta para el desarrollo de software para robots.

De hecho, en los procesos de creación de componentes de Orocos, ya se ha visto que crear un componente en Ros+Orocos es mucho más sencillo y automatizado que en Orocos.

Podría darse el caso de que la instalación de Ros afecte de forma negativa al rendimiento de la tarjeta, y que por ello los resultados de Beagle-Igep sean mucho peores que en Raspberry donde no se ha instalado Ros.

Así pues, se ha procedido a instalar Orocos en la tarjeta Igep, prescindiendo de Ros.

Los resultados han sido los siguientes:

Para un único proceso:

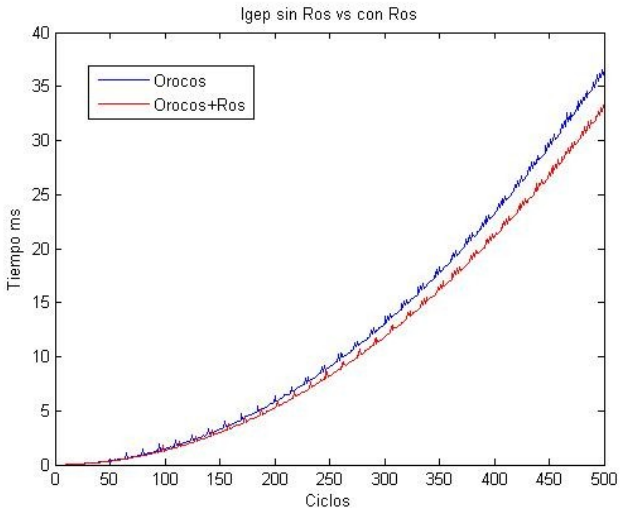

Ilustración 22: Igep Orocos vs Orocos+Ros 1 componente

Para 4 procesos concurrentes:

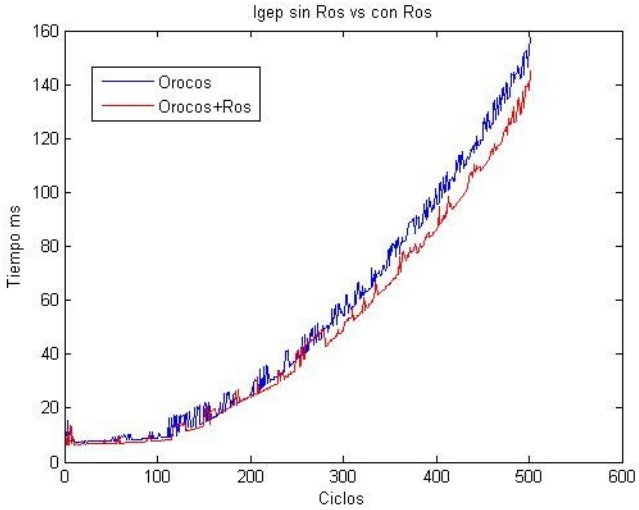

Ilustración 23: Igep Orocos vs Orocos+Ros 4 componentes

Apenas hay diferencia entre ejecutar el benchmark con Ros y sin Ros, siendo incluso mejores tiempos para Orocos+Ros.

El único elemento donde Raspberry es claramente mejor que las otras 2 tarjetas es en su GPU, además de que mientras que en Beagle e Igep se ha instalado un sistema operativo estándar pero compilado específicamente para ellas, en Raspberry se ha instalado un sistema operativo diseñado específicamente para Raspberry.

Cabe la posibilidad de que Raspberry ejecute parte de los cálculos con la GPU, obteniendo por ello resultados mejores que Beagle e Igep.

5. Aplicaciones del proyecto a un robot móvil

Tras haber realizado el estudio de las tres tarjetas empotradas, la instalación de Orocos y ROS, y el benchmarking usando un middleware basado en componentes, se ha realizado una actividad extra consistente en aplicar todo lo aprendido para tratar de mover un robot móvil.

Concretamente, se propuso controlar un robot realizado con piezas de Lego, usando los motores del mismo, con la particularidad de que en vez de usar el "Brick" de Lego, se usaría una de estas tarjetas empotradas para realizar el control y enviar la acción de control.

Para ello, tras revisar información sobre los motores de los Lego MindStorm, estos motores necesitan 2 señales PWM para funcionar. Una de esas señales PWM está relacionada con la velocidad del motor y la otra con el sentido de giro del mismo.

Por ello, se ha usado la tarjeta empotrada IgepV2, en la cual está instalada un Ubuntu con Ros+Orocos. Otro de los motivos para la utilización de esta tarjeta es que ya se había conseguido resolver la generación de señales Pwm a través de la tarjeta, pudiendo generar estas señales a la frecuencia y porcentaje de ciclo deseado.

Uno de los inconvenientes ha sido el voltaje pico a pico que se obtiene de la tarjeta (1.8 V), siendo insuficiente para mover los motores. Por ese motivo, se ha tenido que diseñar un controlador, con el cual esa señal PWM obtenida de la Igep fuera amplificada hasta, como mínimo, 4.6 V, siendo la tensión mínima para poder mover el robot. También se ha tenido que diseñar un regulador de tensión para adaptar la tensión de una batería LiPo (7,4V – 2A) a los requerimientos eléctricos de la Igep (5V – 1A).

En la siguiente figura se pueden observar los componentes utilizados:

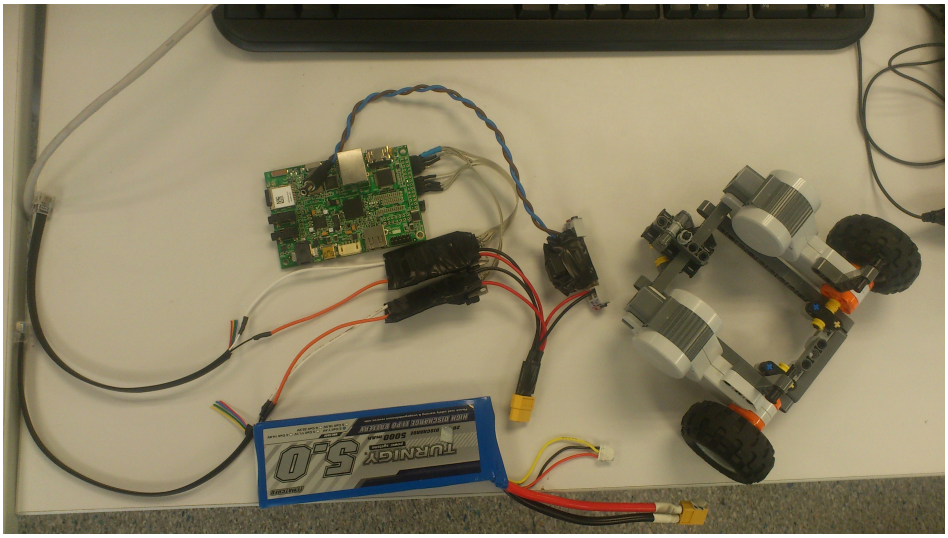

Ilustración 24: Componentes del robot

Se aprecia en la Figura 24 la tarjeta Igep, batería, configuración diferencial del robot con piezas de Lego y los 2 drivers necesarios realizados por el técnico de laboratorio.

A continuación, se puede observar el robot construido:

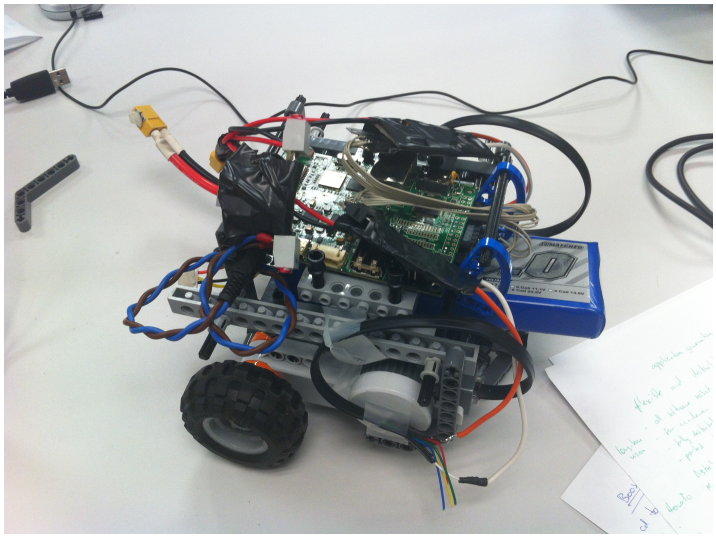

Ilustración 25: Robot diferencial

Para el control se implementó un módulo de Orocos muy simple, el cual, en función de la lectura por teclado que se realizaba, modificaba cada una de las PWM conectadas a los motores. Finalmente, para el acceso al robot de manera remota, desde la propia Igep se ha creado una red ad-

hoc, desde la cual, con cualquier dispositivo (tablet, teléfono o PC) con una tarjeta Wifi se puede acceder mediante SSH y realizar el control de forma remota de una manera eficiente.

6. Conclusiones

Una de las conclusiones que se ha extraído es que es imposible decidirse por una de las tarjetas propuestas para todas las aplicaciones posibles.

Raspberry ha destacado por su bajo precio y alta potencia, aunque no ha sido posible integrar Orocos+Ros en ella, por lo que no es posible aprovechar las facilidades y utilidades de Ros en proyectos realizados con Orocos.

Se propone para el futuro la integración de Orocos+Ros, a partir del proceso descrito anteriormente.

Por otra parte, si bien es cierto que BeagleBone e Igep son tarjetas clónicas, Igep está preparada para trabajar en situaciones más extremas, con certificación industrial, aunque su precio es muy superior al de BeagleBone (y Raspberry, por tanto).

Por tanto, en proyectos donde no se vaya a hacer uso de Ros, Raspberry se desmarca como la mejor opción, por la alta respuesta/bajo precio, aunque dependiendo de si el entorno de utilización es industrial, es probable que Igep V2 deba ser tenida en cuenta.

En proyectos donde se vaya a hacer uso de Ros, se deberá escoger entre Beaglebone o Igep, dependiendo de nuevo de las características del entorno, y teniendo en cuenta que el coste de Beaglebone es 4 veces inferior, con la misma capacidad de procesamiento que Igep.

7. Trabajos futuros

Respecto a trabajos futuros, este ha sido un paso importante, puesto que se ha conseguido integrar un middleware de control con tarjetas empotradas de bajo coste, consumo y tamaño. Además, debido a la potencia de estas tarjetas, los algoritmos de control implementados pueden ser mucho más eficientes que los implementados, por ejemplo, con el Brick de Lego MindStorm. También, gracias a la integración Orocos-Ros, la comunicación entre varios robots así como, por ejemplo, la visualización de los mismos es una tarea completamente resuelta por la comunidad. De esta forma se abre un nuevo abanico de posibilidades para continuar desarrollando código para estas arquitecturas y realizar tareas cada vez más complejas.

8. Anexo: Implementación del algoritmo EDF

8.1. Implementación

Se han implementado las siguientes clases:

- Tarea: cada una de las tareas a ejecutar, y que se caracterizan por los siguientes atributos:
  - computo: Es el tiempo de CPU necesario para completar la ejecución de la tarea en cada activación
  - computo_pendiente: cómputo pendiente de ejecutar en la presente activación de la tarea
  - periodo: Intervalo de tiempo de activación de la tarea
  - deadline: Plazo límite en la que la tarea debe haberse ejecutado (es decir, su computo_pendiente debe ser 0 antes del deadline)
  - ultimo_deadline: Deadline que la tarea tuvo en la activación inmediatamente anterior.

  Los métodos implementados para la clase tarea son los métodos de manipulación de atributos (get_computo(), set_computo(int)...) además de los siguientes:
  - void ejecuta_ciclo(): Ejecuta la tarea seleccionada un ciclo. Si la tarea finaliza (cómputo pendiente es 0) se reinician sus atributos computo_pendiente y ultimo_deadline

- Planificador: Simplemente consta de:
  - tareas: Cola de tareas esperando ejecución
  - total_tareas: Número de tareas en cola
  - ciclo: Ciclo que se está ejecutando
  - total_ciclos: Número de ciclos que se ejecutarán en total, y que puede definir el usuario

  Los métodos más importantes implementados para la clase planificador son:
  - void inserta_tarea(int c, int t): inserta en la cola una tarea de cómputo C y periodo T
  - bool es_planificable(): Devuelve true en caso de que las tareas en cola sean planificables. False en otro caso.

- Planificador_edf: Hereda de planificador, y sus métodos más importantes son:
  - int mas_prioritaria(): Devuelve la posición de la tarea más prioritaria según el algoritmo EDF dentro de la cola de tareas o -1 en caso de idle.
  - void ejecucion(): ejecuta en cada ciclo la tarea más prioritaria

8.2. Pseudocódigo de los métodos más importantes

```
bool es_planificable(){
        Si Σ(Ci / Ti) <=1 entonces
                devuelve true
        si no
                devuelve false
}
```

```
int mas_prioritariaedf(){
        posicion_prioritaria= -1
        menor_deadline= +∞

        recorre cola
                Si deadline (posicion) < menor_deadline entonces
                        posicion_prioritaria= posicion
                        menor_deadline=deadline (posicion)
        fin recorre

        devuelve posicion
}
```

```
void ejecucion(){
        para cada ciclo
                ejecuta_tarea_mas_prioritaria
        fin para
}
```

8.3. Código fuente

---8.3.1. Fichero clases.hpp---

```
class tarea{
private:
        int computo; //computo de la tarea
        int computo_pendiente;//computo pendiente en la ejecucion presente
        int ultimo_deadline; //deadline anterior de la tarea
        int deadline;//deadline tarea
        int periodo; //periodo de la tarea



        public:
        tarea(int c, int t){//constructor
                computo=c;
                computo_pendiente=c; //al inicializar el computo pendiente es todo
                periodo=t;
                ultimo_deadline=0; //al inicializar, la tarea no ha tenido ultimo deadline
                deadline=t; //al inicializar, el deadline de la tarea coincide con su ciclo
        }
        ~tarea(){};


        [métodos get y set]


        void ejecutat_ciclo(){
```

```
                /*TO DO
                cuerpo de la tarea
                */
                computo_pendiente--;                                    //se ejecuta un
ciclo
                if(computo_pendiente==0){                               //la tarea ha
finalizado su ejecucion este ciclo
                        set_ul_deadlinet(get_deadlinet());              //ultimo deadline de la tarea
                        set_deadlinet(get_deadlinet()+get_periodot());  //aumenta su deadline al siguiente ciclo
                        computo_pendiente=get_computot();               //vuelve a tener el
computo pendiente a ejecutar
                }
        }



        void finaliza(){
                this->set_deadlinet(this->deadline+this->periodo);
        }
};

class planificador

{
public:
        int total_tareas;
        tarea *tareas[100];
        int ciclos_totales;
        int ciclo;
public:
        planificador(){
                this->total_tareas=0;
                ciclos_totales=200;
                ciclo=0;

        }
        ~planificador(){}

        void set_ciclos(int c){
                this->ciclos_totales=c;
        }

        void inserta_tarea(int c,int t){
                tareas[this->total_tareas]=new tarea(c,t);
                total_tareas=total_tareas+1;
        }

        void reset(){
                total_tareas=0;
                ciclo=0;
        }

        [métodos get y set]

        bool es_planificable(){
                float total=0;
                int i=0;
                while(i<total_tareas){
                        total=total+((float)get_computo(i)/(float)get_periodo(i));
                        i++;
```

```
                }
////              printf("Factor utilizacion= %f",total);
                if(total<=1) return true;
                return false;
        }

};
class planificador_edf:public planificador{

public:
        //edf(){}
        //~edf(){}
        int mas_prioritaria(){ //devuelve la tarea mas prioritaria (deadline menor) o -1 en caso de idle
                int i=0;
                int prioritaria=-1;
                int menordeadline=ciclos_totales+1;

                while(i<total_tareas){ //para todas las tareas
                        if((get_ul_deadline(i)<=ciclo)&&(get_deadline(i)<menordeadline)){ //comprobamos que no
se haya ejecutado este ciclo y que sea la de menor deadline
                                menordeadline=get_deadline(i);
                                prioritaria=i; //en ese caso, es la tarea mas prioritaria
                        }
                        //en otro caso, no hay tarea a ejecutar (idle)
                        i++;
                }
                return prioritaria;
        }

        void ejecucion(){
                int tarea_prioritaria;

                /*codigo para la generacion del fichero html de visualizacion*/
//              FILE *fd;

//              fd=fopen("home/pi/Desktop/ficheroedf.txt","w");

                while(ciclo<ciclos_totales){
                        tarea_prioritaria=this->mas_prioritaria(); //obtenemos la tarea de mas prioridad

                        if(tarea_prioritaria==-1){
//                              printf("N ");
//                              fprintf(fd,"0\n");
                        }
                        else{
                                tareas[tarea_prioritaria]->ejecutat_ciclo();
//                              fprintf(fd,"%d\n",tarea_prioritaria+1);
//                              printf("P%d ",tarea_prioritaria+1);
                        }
                        ciclo++;
                }
//      fclose(fd);
        }
};
planificador_edf *t=new planificador_edf();
```

---8.3.2. Fichero prueba-component.cpp---

```
#include "prueba-component.hpp"

#include <rtt/Component.hpp>
#include <rtt/TaskContext.hpp>
#include <rtt/Port.hpp>
#include <rtt/Logger.hpp>
#include <iostream>
#include "clases.hpp"

using namespace std;
using namespace RTT;
using namespace RTT::detail;


Prueba::Prueba(std::string const& name) : TaskContext(name){
//InputPort<std::vector<double> > vectorPosicion; //configurar puerto entrada
//RTT::OutputPort<double> out_port; //configurar puerto salida
  std::cout << "Prueba constructed !" <<std::endl;
}

bool Prueba::configureHook(){
  std::cout << "Prueba configured !" <<std::endl;
  f=fopen("/home/pi/fichero.txt","w+");

  iter=1;


for(int i=0;i<10;i++){
  t->inserta_tarea(1,100);
  t->inserta_tarea(2,200);
  t->inserta_tarea(3,300);
  t->inserta_tarea(4,400);
  t->inserta_tarea(5,500);
  t->inserta_tarea(6,621);
  t->inserta_tarea(7,788);
  t->inserta_tarea(8,811);
  t->inserta_tarea(9,981);
  t->inserta_tarea(9,950);
}




  return true;
}

bool Prueba::startHook(){
  std::cout << "Prueba started !" <<std::endl;
  return true;
}

void Prueba::updateHook(){
//FILE *f=fopen("/home/pi/fichero.txt","w+");


if(iter>500)
  this->stopHook();
```

```
  t->set_ciclos(iter);
  time_begin=RTT::os::TimeService::Instance()->getTicks();
   t->ejecucion();
   time_end=RTT::os::TimeService::Instance()->getTicks();

   nano_begin=RTT::os::TimeService::ticks2nsecs(time_begin);
   nano_end=RTT::os::TimeService::ticks2nsecs(time_end);


    fprintf(f,"tiempo con %d ciclos : %lld\n",i,nano_end-nano_begin);
    iter++;

}
}
```

```
void Prueba::stopHook() {
  this->setPeriod(0);
  std::cout << "Prueba executes stopping !" <<std::endl;
  fclose(f);
}

void Prueba::cleanupHook() {
  std::cout << "Prueba cleaning up !" <<std::endl;
}


ORO_CREATE_COMPONENT(Prueba)
```

9. Anexo: Overclocking

El overclocking consiste en alcanzar una mayor velocidad de reloj (es decir, una mayor frecuencia de trabajo) para un componente electrónico, por encima de las especificaciones de fábrica. Con ello se pretende conseguir un rendimiento más alto, lo cual normalmente se aplica a gráficas GPU o a la CPU.

Este funcionamiento por encima de especificaciones puede suponer una pérdida de estabilidad, acortando la vida del producto.

El principal efecto de este sobretrabajo suele ser un mayor consumo acompañado de una temperatura más alta, lo cual es especialmente grave en sistemas empotrados como los que se está tratando. Es muy recomendable refrigerar de forma apropiada la tarjeta (mediante disipador + ventilador).

9.1. Raspberry PI

Raspberry PI es la única de las tarjetas probadas cuyo overclock (hasta 1 GHz) está cubierto por garantía de fábrica. Es decir, es la única tarjeta cuya refrigeración, consumo, etc está preparado para soportar overclock.

Para realizarlo, hay que seguir uno de los 2 métodos siguientes:

9.1.1. Opción 1

```
sudo raspi-config #lanzamos el menú de configuración
#seleccionamos la opción de overclock
```

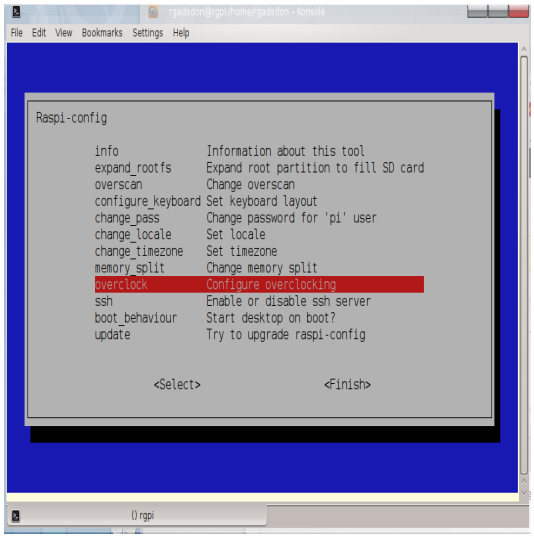

Ilustración 26: Overclock

Seleccionamos el overclock deseado (en el presente documento, máximo 1GHz) y reiniciamos la tarjeta.

9.1.2. Opción 2

Editamos el fichero config.txt de la SD en un PC con Windows. En la opción arm_freq introducir la velocidad deseada (por defecto, 800 MHz; en el presente trabajo, 1000 MHz)

10. Anexo: Ejecución de procesos concurrentes en Orocos

Para la prueba de ejecución de procesos concurrentes se ha programado un componente supervisor cuya misión es recibir el tiempo inicial y final de cada uno de los procesos en ejecución y analizarlos para ver cuánto tiempo han tardado los 4 componentes en realizar una iteración.

Se expone a continuación el código ya que anteriormente no se han realizado conexiones entre componentes:

---10.1. Fichero supervisor-component.hpp---

```
#ifndef OROCOS_SUPERVISOR_COMPONENT_HPP
#define OROCOS_SUPERVISOR_COMPONENT_HPP

#include <rtt/RTT.hpp>
#include <stdio.h>
using namespace std;
using namespace RTT;

class Supervisor : public RTT::TaskContext{
  public:
    Supervisor(std::string const& name);
    bool configureHook();
    bool startHook();
    void updateHook();
    void stopHook();
    void cleanupHook();


            InputPort < long long > entrada_ini1;
            InputPort < long long > entrada_ini2;
            InputPort < long long > entrada_ini3;
            InputPort < long long > entrada_ini4;

            InputPort < long long > entrada_fin1;
            InputPort < long long > entrada_fin2;
            InputPort < long long > entrada_fin3;
            InputPort < long long > entrada_fin4;

 private:
            long long nanosini1,nanosini2,nanosini3,nanosini4,nanosfin1,nanosfin2,nanosfin3,nanosfin4, nanosinitot,
nanosfintot;
            FILE * tiempos;
            bool leeini1,leeini2,leeini3,leeini4,leefin1,leefin2,leefin3,leefin4;


};
#endif
```

---10.2. Fichero supervisor-component.cpp---

```
#include "supervisor-component.hpp"

#include <rtt/Component.hpp>
#include <iostream>

Supervisor::Supervisor(std::string const& name) : TaskContext(name){

nanosini1=0,nanosini2=0,nanosini3=0,nanosini4=0,nanosfin1=0,nanosfin2=0,nanosfin3=0,nanosfin4=0,nanosinitot=0,n
anosfintot=0;
leeini1=false,leeini2=false,leeini3=false,leeini4=false,leefin1=false,leefin2=false,leefin3=false,leefin4=false;
  std::cout << "Supervisor constructed !" <<std::endl;

this->ports()->addEventPort("entrada_ini1",entrada_ini1);
this->ports()->addEventPort("entrada_ini2",entrada_ini2);
this->ports()->addEventPort("entrada_ini3",entrada_ini3);
this->ports()->addEventPort("entrada_ini4",entrada_ini4);
this->ports()->addEventPort("entrada_fin1",entrada_fin1);
this->ports()->addEventPort("entrada_fin2",entrada_fin2);
this->ports()->addEventPort("entrada_fin3",entrada_fin3);
this->ports()->addEventPort("entrada_fin4",entrada_fin4);

}

bool Supervisor::configureHook(){

  tiempos=fopen("/root/capturas/Tiempos.txt","w");
  std::cout << "Supervisor configured !" <<std::endl;
  return true;
}

bool Supervisor::startHook(){
  std::cout << "Supervisor started !" <<std::endl;
  return true;
}

void Supervisor::updateHook(){

if ( entrada_ini1.read(nanosini1) == RTT::NewData ) {
                        leeini1 = true;
                        }
if ( entrada_ini2.read(nanosini2) == RTT::NewData ) {
                        leeini2 = true;
                        }
if ( entrada_ini3.read(nanosini3) == RTT::NewData ) {
                        leeini3 = true;
                        }
if ( entrada_ini4.read(nanosini4) == RTT::NewData ) {
                        leeini4 = true;
                        }
if ( entrada_fin1.read(nanosfin1) == RTT::NewData ) {
                        leefin1 = true;
                        }
if ( entrada_fin2.read(nanosfin2) == RTT::NewData ) {
                        leefin2 = true;
                        }
if ( entrada_fin3.read(nanosfin3) == RTT::NewData ) {
                        leefin3 = true;
                        }
if ( entrada_fin4.read(nanosfin4) == RTT::NewData ) {
```

```
                leefin4 = true;
                }

if(leeini1 && leeini2 && leeini3 && leeini4 && leefin1 && leefin2 && leefin3 && leefin4)
        {
                leeini1=false;leeini2=false;leeini3=false;leeini4=false;
                leefin1=false;leefin2=false;leefin3=false;leefin4=false;

                nanosinitot=nanosini1;
                if (nanosinitot>nanosini2) nanosinitot=nanosini2;
                if (nanosinitot>nanosini3) nanosinitot=nanosini3;
                if (nanosinitot>nanosini4) nanosinitot=nanosini4;

                nanosfintot=nanosfin1;
                if (nanosfintot<nanosfin2) nanosfintot=nanosfin2;
                if (nanosfintot<nanosfin3) nanosfintot=nanosfin3;
                if (nanosfintot<nanosfin4) nanosfintot=nanosfin4;

                //Tenemos en nanosinitot el tiempo de iniciación de un componente (el instante más bajo)
                //Tenemos en nanosfintot el tiempo de finalización de un componente (el instante más alto)

                fprintf(tiempos,"%lld\n",(nanosfintot-nanosinitot));
                std::cout << "Supervisor: "<< nanosfintot-nanosinitot <<std::endl;

        }



}

void Supervisor::stopHook() {
fclose(tiempos);
  std::cout << "Supervisor executes stopping !" <<std::endl;
}

void Supervisor::cleanupHook() {
  std::cout << "Supervisor cleaning up !" <<std::endl;
}

/*
 * Using this macro, only one component may live
 * in one library *and* you may *not* link this library
 * with another component library. Use
 * ORO_CREATE_COMPONENT_TYPE()
 * ORO_LIST_COMPONENT_TYPE(Supervisor)
 * In case you want to link with another library that
 * already contains components.
 *
 * If you have put your component class
 * in a namespace, don't forget to add it here too:
 */
ORO_CREATE_COMPONENT(Supervisor)
```

---10.3. Fichero pruebaDistrib-component.hpp---

```
#ifndef OROCOS_PRUEBADISTRIB_COMPONENT_HPP
#define OROCOS_PRUEBADISTRIB_COMPONENT_HPP

#include <rtt/RTT.hpp>
#include <stdio.h>
using namespace std;
using namespace RTT;

class PruebaDistrib : public RTT::TaskContext{
  public:
    PruebaDistrib(std::string const& name);
    bool configureHook();
    bool startHook();
    void updateHook();
    void stopHook();
    void cleanupHook();


OutputPort < long long > salida_ticksini;
OutputPort < long long > salida_ticksfin;

private:
    RTT::os::TimeService::ticks time_begin, time_end;
    long long nano_begin, nano_end;
    int iter;

};

#endif
```

---10.4. Fichero pruebaDistrib-component.cpp---

```
#include "pruebaDistrib-component.hpp"

#include <rtt/Component.hpp>
#include <iostream>

PruebaDistrib::PruebaDistrib(std::string const& name) : TaskContext(name){

  this->ports()->addPort("salida_ticksini",salida_ticksini);
  this->ports()->addPort("salida_ticksfin",salida_ticksfin);

  std::cout << "PruebaDistrib constructed !" <<std::endl;
}

bool PruebaDistrib::configureHook(){

  std::cout << "Prueba configured !" <<std::endl;
iter=1;
for(int i=0;i<10;i++){
  t->inserta_tarea(1,100);
  t->inserta_tarea(2,200);
  t->inserta_tarea(3,300);
  t->inserta_tarea(4,400);
  t->inserta_tarea(5,500);
  t->inserta_tarea(6,621);
  t->inserta_tarea(7,788);
  t->inserta_tarea(8,811);
  t->inserta_tarea(9,981);
```

```
  t->inserta_tarea(9,950);
}

salida_ticksini.setDataSample(nano_begin);
salida_ticksfin.setDataSample(nano_end);

  std::cout << "PruebaDistrib configured !" <<std::endl;
  return true;
}

void PruebaDistrib::updateHook(){

 if(iter>500) this->stop();

  t->set_ciclos(iter);
  time_begin=RTT::os::TimeService::Instance()->getTicks();
  nano_begin=RTT::os::TimeService::ticks2nsecs(time_begin);
  salida_ticksini.write(nano_begin);


            t->ejecucion();



   time_end=RTT::os::TimeService::Instance()->getTicks();
   nano_end=RTT::os::TimeService::ticks2nsecs(time_end);
   salida_ticksfin.write(nano_end);

   iter++;

  std::cout << "Iter: "<<iter <<std::endl;

}

bool PruebaDistrib::startHook(){
  std::cout << "PruebaDistrib executes startHook !" <<std::endl;
return true;
}

void PruebaDistrib::stopHook() {
this->setPeriod(0);
  std::cout << "PruebaDistrib executes stopping !" <<std::endl;
}

void PruebaDistrib::cleanupHook() {
  std::cout << "PruebaDistrib cleaning up !" <<std::endl;
}

/*
 * Using this macro, only one component may live
 * in one library *and* you may *not* link this library
 * with another component library. Use
 * ORO_CREATE_COMPONENT_TYPE()
 * ORO_LIST_COMPONENT_TYPE(PruebaDistrib)
 * In case you want to link with another library that
 * already contains components.
 *
 * If you have put your component class
 * in a namespace, don't forget to add it here too:
 */
```

ORO_CREATE_COMPONENT(PruebaDistrib)

---10.5. Fichero Distrib.ops---

Para lanzar los distintos procesos en orden adecuado, es necesario crear un fichero de texto plano con la extensión ops y lanzarlo con orocos mediante:

```
deployer-gnulinux -s Distrib.ops
```

```
import("pruebaDistrib")
import("supervisor")

loadComponent("p1","PruebaDistrib")
loadComponent("p2","PruebaDistrib")
loadComponent("p3","PruebaDistrib")
loadComponent("p4","PruebaDistrib")
loadComponent("s1","Supervisor")

p1.configure
p2.configure
p3.configure
p4.configure
s1.configure

p1.setPeriod(0.5)
p2.setPeriod(0.5)
p3.setPeriod(0.5)
p4.setPeriod(0.5)

connect("p1.salida_ticksini","s1.entrada_ini1", ConnPolicy)
connect("p2.salida_ticksini","s1.entrada_ini2", ConnPolicy)
connect("p3.salida_ticksini","s1.entrada_ini3", ConnPolicy)
connect("p4.salida_ticksini","s1.entrada_ini4", ConnPolicy)

connect("p1.salida_ticksfin","s1.entrada_fin1", ConnPolicy)
connect("p2.salida_ticksfin","s1.entrada_fin2", ConnPolicy)
connect("p3.salida_ticksfin","s1.entrada_fin3", ConnPolicy)
connect("p4.salida_ticksfin","s1.entrada_fin4", ConnPolicy)

s1.start
p1.start
p2.start
p3.start
p4.start
```

11. Anexo: Implementación del PWM para el robot

```
bool RobotPwm::configureHook(){

        fd_pwm0 = pwm_init(0);
        fd_pwm1 = pwm_init(1);
        res = pwm_set_frequency(0, 1000);
        res = pwm_set_frequency(1, 2000);
        cont = 0;
        ch=0;
        std::cout << "RobotPwm configured !" <<std::endl;
        return true;
}

bool RobotPwm::startHook(){

        std::cout << "RobotPwm started !" <<std::endl;
init_keyboard();

        return true;
}

void RobotPwm::updateHook(){

if(kbhit()) { //Si te pulsa alguna techa
switch (ch=getchar())
        {
                case 'w'://Adelante
                case 'W':
                        cont=90;
                        cont2=90;
                        break;
                case 's'://Parado
                case 'S':
                        cont=0;
                        cont2=0;
                        break;
                case 'a'://Giro derecha
                case 'A':
                        cont=0;
                        cont2=90;
                        break;
                case 'd'://Giro izquierda
                case 'D':
                        cont=90;
                        cont2=0;
                        break;
        }
        res = pwm_set_value(fd_pwm0, cont); //La Pwm0 saca un porcentaje "cont"
        res = pwm_set_value(fd_pwm1, cont2);//La Pwm1 saca un porcentaje "cont2"
        printf("You pressed '%c'!\n", ch);

}
```

12. Referencias

[1.] Alfons Crespo, Alejandro Alonso. Informática Industrial / Sistemas de Tiempo Real. *Revista Iberoamericana de Automática e Informática Industrial. Vol 3. Núm 2. pp 4-6. Abril 2006*
[2.] Jonathan W Valvano. *Introducción a los Sistemas de Microcontroladores Empotrados. Cengage Learning Editores. 2003*
[3.] Marina Vallés, J. Ignacio Cazalilla, Ángel Valera y otros. Implementación basada en el middleware Orocos de controladores dinámicos pasivos para un robot paralelo. *Revista Iberoamericana de Automática e Informática Industrial. 00 (2012) 1-8*
[4.] BeagleBoard Foundation. *BeagleBoard Brief.*
[5.] Bruyninckx, H. (2001). Open robot control software: the OROCOS project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on* (Vol. 3, pp. 2523-2528). IEEE.
[6.] Herman Bruyninckx. Orocos: design and implementation of a robot control software framework. *April 2002*
[7.] Morgan Quigley, Andrew Ng, Brian Gerkey y otros. Ros: an open-source Robot Operating System.
[8.] Liu, C.L y Layland J.W. Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment. *Journal of the Association for Computing Machinery. vol 20-1, pp 46-61. January 1973*
[9.] Eben Upton. Interview. Linux Co UK. February 2013