Un Manejador de Comunicaciones Minimal para Programas Síncronos sobre Redes de Transputers

Carlos Figueira

Departamento de Computación y
Tecnología de la Información
Universidad Simón Bolívar
Apdo. 89000 - Caracas 1080-A
Venezuela

Resumen

Este artículo presenta el diseño de un Manejador de Comunicaciones Minimal para aplicaciones, programadas en SIGNAL e implementadas en Redes de Transputers. SIGNAL es un lenguaje síncrono tipo flujo de datos para la especificación y programación de sistemas tiempo real. Luego de una etapa de validación, el programa SIGNAL es distribuido en la red de Transputers. El Manejador de Comunicaciones debe permitir las comunicaciones resultantes de dicha distribución.

Abstract

This article introduces the design of a minimal routing kernel, for applications, programmed in SIGNAL and implemented in a Transputer Network. SIGNAL is a synchronous data-flow language for real-time systems specifications and programming. After a validation stage, the SIGNAL program is distributed on the Transputer network. The routing kernel should permit the communications induced by the distribution.

1 Introducción

En los últimos años, hemos asistido a la aparición de un nuevo tipo de lenguajes, los lenguajes *síncronos* (ESTEREL[2], STATECHARTS[6], LUSTRE[3] y SIGNAL[12, 4]), diseñados para la especificación y programación de sistemas complejos de tiempo real, tales como el procesamiento de señales en aeronáutica, robótica, sistemas reactivos y reconocimiento de voz. La particularidad esencial de estos lenguajes es el manejo del tiempo según el principio que las acciones tienen una duración de tiempo lógico nula, liberando así al programador de los problemas

relacionados con la implementación, durante la etapa de diseño de los algoritmos. Por otro lado, este tipo de aplicaciones exige tiempos de respuesta muy rápidos, lo cual hace necesario el uso de multiprocesadores.

SIGNAL es un lenguaje tipo flujo de datos, desarrollado en el IRISA, Rennes, y forma parte de un proyecto de sistema de Diseño Asistido por Computador que engloba desde el desarrollo de los algoritmos de tiempo real hasta la implementación de estos en una arquitectura de multiprocesadores. El elemento básico manejado por el lenguaje se denomina *señal*: una señal puede ser considerada como una variable, de tipo definido, cuyo contenido es válido (y disponible) sólo en ciertos instantes lógicos, determinados por su *reloj*.

El compilador de SIGNAL verifica que el programa es correcto; en particular, que no contenga ciclos de dependencia, lo cual garantiza la ausencia de bloqueo.

En el contexto actual, la arquitectura prototipo consiste en una red de Transputers[7]. El Transputer es un microprocesador de 32 bits diseñado para ser utilizado como elemento de base de arquitecturas de multiprocesadores: es fácilmente interconectable gracias a sus 4 conectores seriales, posee una buena capacidad de cómputo entera y flotante (T800), y es programable en occam[9], que es un lenguaje de alto nivel.

Una vez seleccionada la topología de la arquitectura, el programa original, compilado, es particionado y cada partición es asignada a uno de los procesadores de la arquitectura. Es necesario entonces agregar el código de un *Manejador de Comunicaciones* en cada procesador, que se encargue de compartir los conectores seriales de los Transputers entre las diferentes señales que pasan por dichos conectores, y las señales que comunican al código de ese procesador con otros procesadores.

El código asignado a un procesador está compuesto por dos procesos:

- Un proceso *módulo*, que corresponde a aquella parte de la aplicación asignada al procesador.
- un proceso *sistema*, el cuál se encarga del ruteo, multiplexado, comunicación con el exterior y control del principio y fin de los instantes lógicos.

Estos dos procesos se implementan en *paralelo* en occam de la manera siguiente[1]:

```
...
PRI PAR
  sistema
  módulo
```

La ejecución de programas SIGNAL en la red de Transputers se rige por el principio de *sincronía fuerte*:

[1] Las ventajas de colocar en alta prioridad las tareas sistema ha sido reportada en [1, 13].

En cada instante lógico, cada procesador ejecuta las acciones que le corresponden a dicho instante, y luego espera a que todos los procesadores hayan terminado sus respectivas acciones para ese instante.

Esta sincronización entre los distintos procesadores se efectúa por intercambio de mensajes. Un árbol cobertor se define sobre la arquitectura. Así, cada nodo recibe de su padre la señal de comienzo de instante, la retransmite a sus hijos, y luego envía su padre la señal de fin una vez que reciba de sus hijos la señal de fin de instante, y que él mismo haya terminado. El procesador en la raíz del árbol concentra la decisión de comienzo y fin de instante.

El Manejador de Comunicaciones (llamado también MC en lo sucesivo) debe ser diseñado de manera tal que conserve las propiedades de ausencia de bloqueo del programa, ya verificadas por el compilador, pero minimizando la carga que él induce sobre el procesador. El artículo presenta cómo se alcanzan estos objetivos.

2 El Manejador de Comunicaciones

Los sistemas de enrutamiento utilizados en redes de computadores se clasifican en dos tipos, según el procesamiento que se aplica a los mensajes en los nodos intermedios:

- *store and forward*: los mensajes son recibidos, guardados en memoria y luego retransmitidos hacia su destino.
- *virtual cut-through*: los mensajes comienzan a ser retransmitidos antes de haber sido recibidos por entero.

Este último requiere de circuitería especializada. Una variante de éste método, denominada enrutamiento *wormhole*, ha sido implementada en VLSI para los hipercubos iPSC, de Intel, y Cosmic Cube, de Caltech. Las ventajas de disponer de circuitería especializada para el enrutamiento son evidentes; sin embargo, ésto sólo es posible para topologías regulares, tales como el hipercubo.

La interfaz de los conectores seriales del Transputer, la cuál funciona de manera concurrente a la unidad central del procesador, se presta bien para implementar un sistema de enrutamiento del primer tipo (store and forward), ya que esta interfaz funciona en modo DMA, es decir, en acceso directo a la memoria. La próxima generación de Transputers, el H1, incorporará circuitería para enrutamiento wormhole, por lo cual el análisis presentado aquí no se le aplica.

2.1 Principios del Manejador de Comunicaciones

El MC está constituido por un conjunto de procesos occam generados para cada procesador. El objetivo consiste en proveer los servicios de enrutamiento

sin introducir bloqueos, utilizando el mínimo de recursos. Los recursos en este caso son el espacio de memoria y tiempo del CPU.

Un bloqueo en una red de interconexión ocurre cuando ningún mensaje puede avanzar hacia su destino debido a que las colas de mensajes están llenas [10]. Diversos métodos para evitar bloqueos han sido propuestos en la literatura [5].

En nuestro contexto, disponemos de toda la información acerca de las comunicaciones en la red, de manera estática, es decir, durante la compilación. Esta consiste en:

- El tipo de los datos que serán transmitidos, ya que las señales comunicadas entre procesadores han sido declaradas en el programa;
- La ruta que siguen los datos.

Según Günther [5], para prevenir la aparición de bloqueos en sistemas en los cuales las comunicaciones son conocidas estáticamente, debe establecerse un *canal virtual* para cada comunicación.

Un canal virtual es un canal lógico que comparte canales físicos de la red, pero que tiene asignada una cola propia en cada nodo intermedio.

En ese contexto, las únicas causas posibles de bloqueo se reducen a aquellas inherentes a la aplicación[2]: si las colas no son vaciadas oportunamente, su saturación provocará el bloqueo de otras señales.

Para ver que esta situación no puede aparecer en nuestro caso, consideremos una cola de dimensión 1, situada en un procesador perteneciente a la ruta de una señal a. Puesto que las colas son únicas para cada señal, ningún otro mensaje puede llegar a esa cola; es decir, **no se introducen nuevas dependencias entre señales**.

Supongamos ahora que la cola está llena con un valor a_t de la señal "a", correspondiente al instante lógico t. Para que la cola se sature, un nuevo valor a_{t+1} de "a" debería ser producido por el procesador fuente. Pero, si a_t no ha podido alcanzar su procesador de destino, esto implica que el instante lógico t no ha terminado aún, puesto que el procesador de destino requiere ese valor en sus acciones del instante t. Esto contradice el principio de sincronía fuerte.

Dos consecuencias importantes para el enrutamiento derivadas del principio de sincronía fuerte son:

- Colas de capacidad para un sólo valor son suficientes para cada señal. Esto minimiza la cantidad de memoria utilizada por el manejador.
- No hay necesidad de implementar un control de saturación de las colas, lo cual reduce el uso del CPU por parte del MC.

[2]En la notación de Günther, éstas son denominadas *Propagación de Bloqueos Causados por el Usuario en la Red.*

El sistema de enrutamiento que implementaremos difiere ligeramente del presentado anteriormente, en que las colas de los nodos intermedios son desplazadas hacia los procesadores fuente y destino. Es obvio que las propiedades de ausencia de bloqueo mostradas aquí son conservadas por esta transformación.

2.2 Descripción del Manejador de Comunicaciones

El MC asociado a cada procesador está compuesto por varios procesos occam. La separación en varios procesos se debe a:

- garantizar la unicidad de las colas. Esto se logra al asignar un buffer de entrada y un buffer de salida para cada señal.
- explotar al máximo la concurrencia entre el CPU y las interfaces de los conectores.
- explotar la concurrencia entre los diferentes conectores de un Transputer.

Por cada conector serial de entrada:

- un proceso llamado *receptor*, el cual recibe los mensajes y los pasa al enrutador de ese conector (ver más adelante).
- un *enrutador de conector*. Este interpreta la información de enrutamiento del mensaje y decide a cual *buffer* debe enviarlo.

Por cada conector serial de salida: un proceso *multiplex*, el cuál recibe los mensajes de salida de los buffers interconectores y de los buffers de las señales de salida del procesador, y los envía por el conector serial.

Buffers inter-conectores, situados entre cada enrutador de conector de entrada y cada multiplex de salida, en los casos en que estos dos conectores pertenezcan a alguna de las rutas definidas en la aplicación.

Por cada señal de entrada del módulo de un procesador, un *buffer de entrada* que se encarga de recibir el mensaje del enrutador, extraer el valor que éste contiene y pasar dicho valor al módulo.

Por cada señal de salida del módulo, un *buffer de salida*, el cuál recibe el valor de la señal del módulo, le agrega la información de enrutamiento y envía el mensaje así compuesto al multiplex del conector de salida que corresponde a la ruta de dicha señal.

Note que todos los procesos, excepto los buffers de entrada y salida, pueden ser compartidos por otras señales. Estos buffers desempeñan el papel de las colas de los canales virtuales, con la diferencia que aquí han sido desplazadas hacia los procesadores fuente y destino.

La implementación de estos procesos es presentada en el anexo.

2.3 Medidas preliminares

Algunas medidas preliminares han sido tomadas sobre una plataforma inmos B003, con T414 corriendo a 15 MHz y conectores a 20 Mbits por segundo, con el objeto de obtener un orden de magnitud de la carga sobre el CPU ocasionada por el MC.

Las pruebas consistieron en el envío de un mensaje desde un nodo a sí mismo vía un nodo intermediario. Fueron hechas a la vez estimaciones usando el manual del Transputer [8], y medidas con el *timer* del Transputer.

El mensaje de prueba es un número de punto flotante de 64 bits, a los cuales se agrega el encabezado del mensaje (2 octetos) más un octeto que indica el largo del mensaje, lo cual dá un largo total de 11 octetos para el mensaje.

El tiempo tomado por ambas transmisiones, del nodo fuente al nodo intermediario, luego de vuelta al nodo fuente, sobre el conector fué de

$$28.8\mu s$$

El tiempo total en ambos Transputers fué de

$$753.5 ciclos(20MHz) \approx 37.67\mu s$$

Por último, el tiempo CPU sobre el nodo intermediario fué de:

$$366 cycles \approx 18\mu s$$

Las pruebas fueron hechas sin ninguna otra carga en los Transputers, y con sólo las transmisiones especificadas arriba. Otras medidas deben ser realizadas en situaciones realistas para poder evaluar de manera precisa el MC.

3 Conclusiones

Un Manejador de Comunicaciones adaptado a la ejecución de programas SIGNAL sobre una arquitectura a base de Transputers ha sido presentada. Su interés principal consiste en la metodología seguida para aprovechar las características de las aplicaciones para obtener un manejador de comunicaciones eficaz y seguro.

La eficiencia exigida para cumplir con las especificaciones temporales en las aplicaciones de tiempo real, justifica el empleo de diseños específicos en lugar de utilizar sistemas de enrutamiento generalizados, como es el caso de SYNDEX [11].

Por otro lado, el análisis de las posibles causas de bloqueo, facilitado por las propiedades estáticas de los programas SIGNAL y la verificación realizada por el compilador, nos guiaron hacia un diseño que logra evitarlos por completo.

Referencias

[1] P. Atkin. Performance maximisation. *inmos Technical Note 17*, 1987.

[2] G. Berry, P. Couronné, and G. Gonthier. Synchronous programming of reactive systems: an introduction to ESTEREL In K. Fuchi and M. Nivat, editors, *Programming of future generation computers*, pages 35–56. ICOT INRIA, North-Holland, 1988.

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, 1987.

[4] T. Gautier, P. Le Guernic, and Loïc Besnard. SIGNAL: a declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *Functional programming languages and computer architecture*, pages 257–277. Lecture Notes in Computer Science, 274, Springer-Verlag, 1987.

[5] K. Günther. Prevention of deadlocks in packet-switched data transport systems. *IEEE Transactions on Communications*, Com-29(4):512–524, 1981.

[6] C. Huizing, R. Gerth, and W. P. de Roever. Modelling STATECHARTS behaviour in a fully abstract way. In M. Dauchet and M. Nivat, editors, *13th Colloquium on Trees in Algebra and Programming CAAP'88, Lecture Notes in Computer Science*, pages 271–294, Nancy France, March 1988. Springer Verlag. Volume 299.

[7] INMOS. *Transputer Reference Manual.* INMOS Ltd., 1000 Aztec West, Aldmondsbury, Bristol, England, 1986.

[8] INMOS. *The* TRANSPUTERS, *Reference Manual.* Bristol, U. K., inmos edition, 1986.

[9] INMOS. occam *2 Reference Manual.* Prentice Hall, 1988.

[10] L. Kleinrock. *Queueing Systems.* Wiley, 1976.

[11] C. Lavaranne and Y. Sorel. Syndex: un environnement de programmation pour multi-processeur de traitement du signal. Research report 113, INRIA FRANCE, 1989.

[12] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. SIGNAL: a data flow oriented language for real time signal processing. In C. I. Byrnes and A. Lindquist, editors, *Computational and combinatorial methods in systems theory*, pages 419–433. North-Holland, 1986.

[13] D. Pritchard, C. Askew, D. Carpenter, I. Glendinning, A. Hey, and D. Nicole. Practical parallelism using transputer arrays. In *Parle 87* Springer-Verlag, 1989. LNCS 258.

A Implementación

El formato de lo mensajes, así como el código occam del Manejador de Comunicaciones son presentados en este anexo.

A.1 Formato de los mensajes

Un mensaje es una secuencia de octetos, compuesto por:

1. tamaño (en octetos). en occam, es necesario especificar el número de octetos que posee el mensaje.
2. identificación de la ruta.
3. identificación de la señal.
4. el valor de la señal, el cuál es traducido en una secuencia de octetos a la transmisión y reconstruído a la recepción.

Los mensajes son formateados en los buffers de salida. El valor es extraído en los enrutadores de conector.

A.2 El receptor

Su función primordial es separar las tareas de recepción por el conector, que es una operación lenta, de la tarea de enrutamiento. Su estructura es:

```
[maxsize]BYTE message:
BYTE size.mess:
WHILE TRUE
  SEQ
    Linki ? size.mess::message
    Ri.Routi ! link.mess::[message FROM 0 FOR INT size.mess]
```

- Linki significa el conector "i"
- Linki.Routi conceta el receptor de este conector a su enrutador de conector Ri.

A.3 Enrutador de conector

Este proceso contiene la información para el enrutamiento de los mensajes que llegan por ese conector. La información es representada por dos funciones: *gen.route* and *loc.route*. La primera devuelve, para la ruta de un mensaje, el buffer interconector al cuál debe ser enviado el mensaje. En el caso que este procesador sea el destino, la segunda función provee el buffer de entrada, de acuerdo a la identidad de la señal contenida en el mensaje.

El proceso enrutador Routi tiene las siguientes conexiones:

- a los buffers interconectores vía un arreglo de canales "Routi.BiLi[dest]", donde el índice "dest" selecciona el buffer interconector.
- a los buffers de entrada vía un arreglo de canales "Routi.IBi[loc.dest]"; el índice "loc.dest" selecciona el buffer de entrada al cual el valor es enviado.

El código occam es:

```
[maxsize]BYTE message:
BYTE size.mess,dest,loc.dest:
WHILE TRUE
  SEQ
    Ri.Routi? size.mess::message
    dest := gen.route(message[0])
    IF
      dest <> 0
        Routi.BiLi[dest] ! size.mess::[message FROM 0 FOR INT size.mess]
      dest = 0
        SEQ
          loc.dest := loc.route(message[1])
          Routi.IBi[loc.dest] ! (size.mess-2)::[message FROM 0 FOR INT size.mess-2]
```

A.4 Buffers Interconectores

Estos conectan al enrutador de un conector de entrada al multiplex de un conector de salida, como se especifica a continuación:

- al enrutador de conector a través del canal "Routi.BiLi[ind.in]", donde "id.in" es una constante que identifica el elemento del arreglo de canales que lo conecta al enrutador.
- al multiplex de salida a través del canal "B.Mux[ind.out]", donde "ind.out" es una constante que indica el canal que lo conecta al multiplex.

```
[maxsize]BYTE message:
BYTE size.mess:
VAL ind.in IS ...:
VAL ind.out IS ...:
WHILE TRUE
  SEQ
    Routi.BiLi[ind.in] ? size.mess::message
    B.Muxj[ind.out] ! size.mess::[message FROM 0 FOR INT size.mess]
```

A.5 Buffer de entrada

El código occam es:

```
TYPE a.v :
VAL dest IS ...:
WHILE TRUE
  SEQ
    Routi.IBi[dest] ? a.v
    a ! a.v
```

donde

- "Routi.IBi[dest]" conecta el buffer de entrada con el enrutador de conector; la constante "dest" indica el canal del arreglo que lo conecta al enrutador de conector.
- "a" es el canal que conecta el buffer al módulo.

A.6 Buffer de salida

```
TYPE x.v :
VAL l IS ...:
VAL BYTE route IS ...:
VAL BYTE ident IS ...:
VAL BYTE size.sig IS ...:
VAL BYTE size.mes IS ...:
WHILE TRUE
  SEQ
    x ? x.v
    B.Muxi[l] ! size.sig;route;ident; signal.v
```

- "x" es el canal que conecta el módulo al buffer de salida.
- "B.Mux[l]" conceta el buffer al multiplex de salida del conector "i".
- "route" e "ident" son los identificadores de la ruta y de la señal, respectivamente.
- "size.sig" es la cantidad de octetos según el tipo de la señal.
- "size.mess" es el número de octetos del mensaje entero.

A.7 Multiplex de salida

Se implementa de la siguiente manera:

```
VAL INT n IS :
[max.size]BYTEmessage :
BYTE size.mes :
WHILE TRUE
  SEQ
    ALT j=0 FOR n
      B.Mux[j] ? size.mes::message
        SKIP
    linkj ! size.mess::[message FROM 0 FOR INT size.mess]
```

- "n" es el número de canales que conecta al multiplex con los buffers de salida y los buffers interconectores.
- "max.size" es el largo máximo (en octetos) de los mensajes que atraviesan este conector.
- "linkj" es el conector de salida.