HERRAMIENTA PARALELA PARA EL DISEÑO DE AUTÓMATAS DE ESTADOS FINITOS

Juan Fº Sanjuan Estrada[1], J. M. García Donaire [2] y J. A. Alvarez Bermejo[3]
Dpto. Arquitectura de Computadores y Electrónica. Universidad de Almería
[1] jsanjuan@ual.es; [2] julian@ace.ual.es; [3] jaberme@ace.ual.es

RESUMEN

El diseño de autómatas de estados finitos sigue una metodología completamente automatizable, de tal forma que cualquier sistema secuencial síncrono puede ser diseñado a partir de la correcta elaboración de la tabla de transiciones entre estados. Sin embargo, surgen dos problemas importantes cuando se desea obtener un circuito óptimo con el mínimo número de biestables y de puertas lógicas. Por un lado, el problema de la minimización de estados que permite reducir el número de biestables, y por otro lado, el problema de la codificación o asignación de estados que intenta reducir la complejidad del sistema combinacional. Este último es un problema NP-complejo, de tal forma que los requerimientos computacionales crecen exponencialmente a medida que aumenta el número de estados a codificar.

En el presente artículo se muestra una herramienta software, denominada CCEDAF (*Codificador Combinatorio de Estados para el Diseño de Autómatas Finitos*) que minimiza el problema de codificación de estados mediante una solución paralela.

1. INTRODUCCIÓN

Una adecuada codificación de estados permite obtener el circuito secuencial síncrono óptimo, asignando códigos binarios a los estados que describen la máquina de estados finitos, de tal forma que los próximos estados binarios y las funciones de salida se pueden eficientemente implementar con una tecnología determinada. Sin embargo, la evolución de los algoritmos de asignación de estados está íntimamente relacionada a la evolución de las plataformas de implementación.

El primer intento de automatizar los algoritmos de asignación de estados mediante computadores se remonta a la década de los años 60, cuando la tecnología de implementación se basaba en diodos, transistores y puertas sencillas. Posteriormente, Humphrey [1] formuló las "*reglas de adyacencia de estados*" basadas en agrupar 1´s en tablas de Karnaugh para obtener las funciones de salida y los próximos estados, de tal forma que permitan minimizar el número de términos producto en las expresiones suma de productos que describen la parte combinacional del circuito secuencial síncrono [2] [3] [4].

Posteriormente, la tecnología cambió las observaciones formuladas por Humphrey llegaron a ser las bases de las próximas generaciones de algoritmos. En la década de los 80, Micheli introdujo la fase de minimización lógica antes de la codificación de estados en KISS [5]. La fase de minimización tiene la ventaja de que el coste de implementación con tecnología PLA puede ser estimado a partir del número de entradas, salidas y términos productos de las funciones minimizadas. Las funciones minimizadas se pueden implementar con tecnología PLA donde la función de coste (área del circuito) se determina a priori de la asignación, asumiendo que el código posteriormente asignado satisface todas las codificaciones resultantes desde la fase de minimización.

El método implementado en KISS tiene sin embargo varios inconvenientes. En primer lugar, en la fase de minimización lógica, las funciones de próximo estado se consideran independientemente, es decir, la minimización de la función del próximo estado no se tiene en cuenta. Este aspecto es especialmente relevante en los contadores secuenciales síncronos, donde los estados minimizados para un contador de 2p estados tiene 2^p términos producto, mientras el optimo es $O(p^2)$. Además, tampoco se tiene en cuenta la realimentación (*feedback*) del autómata.

La idea correspondiente a la minimización lógica se basaba en la minimización de los implicantes generalizados, que fue propuesta en [5]. Sin embargo, la complejidad computacional del método presentado lo hizo impracticable para autómatas con más de 8 estados.

En 1988, Józwiak publicó el método de máximas adyacencias, denominado MAXAD, orientado a implementaciones de dos niveles lógicos (PLA y circuitos lógicos AND-OR) [6] [7]. Aunque el método usa "adyacencia entre estados", difiere considerablemente de los métodos anteriores basados en este idea. Józwiak ejecuta un sofisticado análisis previo considerando muchos tipos de adyacencia, y usa los resultados del análisis de forma eficiente en la generación del código binario. Comparando los resultados obtenidos con KISS, la maquina diseñada por MAXAD tiene implementaciones con un promedio del 13% menos área PLA y 28% menor del área de realimentación.

Alguno de los inconvenientes de KISS se solucionan en su sucesor NOVA [8]. NOVA implementa una aproximación eficiente y flexible, representando el problema de asignación de estados como un problema de gráfico que se resuelve en varias estrategias heurísticas produciendo resultados superiores y ofreciendo una buena relación calidad / tiempo de ejecución. La mejor estrategia de NOVA, denominada *iohybrid*, produce resultados de calidad comparable a los resultados del método de máxima adyacencia.

En general, los métodos basados en minimización lógica solamente difieren del resto de métodos en el nivel de detalle de las restricciones. Todos estos métodos aplicados a implementaciones de dos niveles, sin embargo, no son aplicables a implementaciones multinivel, excepto para redes de multiplexores. Uno de los primeros métodos de asignación de estados multi-nivel fue MUSTAND [9], diseñado para trabajar con sistemas de síntesis lógica, permitiendo maximizar el número y tamaño de los sub-cubos de expresiones describiendo las funciones de salida. Estos sub-cubos permiten minimizar el número de letras. La maximización del cubo es realizada en el proceso de asignación del código adyacente a alguna pareja de estados seleccionados. La selección de la pareja se realiza de acuerdo a reglas similares a las de Humphrey.

Alguna de las aproximaciones alternativas de interés para codificar estados son los algoritmos genéticos [16] [17], que calculan la calidad de un cromosoma, codifican la maquina con el código representado por el cromosoma y minimizan las funciones lógicas resultantes con ESPRESSO [19]. La calidad de la asignación se representa por el tamaño de la implementación PLA. Desafortunadamente, los algoritmos genéticos básicos son conocidos pobremente por manipular los problemas con funciones de calidad complicadas y costosas en tiempo, como es el caso de las aplicaciones de codificación de estados.

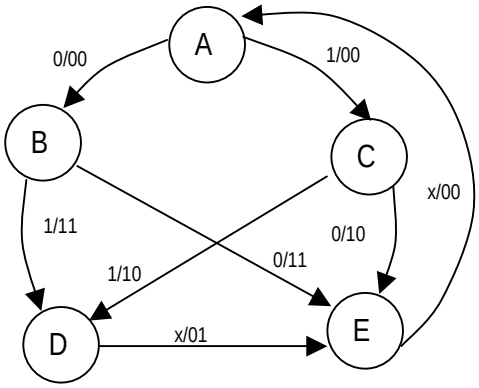

Figura 1.- Diagrama de estados

Estado actual	Próximo estado		Salida	
	I_0		Z_0	Z_1
	0	1		
A	B	C	0	0
B	E	D	1	1
C	E	D	1	0
D	E	E	0	1
E	A	A	0	0

Tabla 1.- Tabla de transiciones

El tiempo de ejecución de los algoritmos genéticos excede en algunos casos a los algoritmos clásicos por un factor de 100 para pequeños ejemplos. Una dirección interesante apuntada en [6] es, sin embargo, la codificación simultanea y selección de tipos de flip-flops usados para almacenar las variables de estado (D o J-K). En algunos casos, la elección de flip-flops JK (los cuales son requeridos por requerir menos funciones de excitación complicadas) reducen la componente combinacional de la máquina de estados finita más del 80%.

Este articulo se estructura de la siguiente forma: en primer lugar, se realiza una breve introducción a los autómatas de estados finitos (sección 2), continuando con una descripción global de la automatización implementada en la herramienta CCEDAF [11] (sección 3). Posteriormente, se comentan las diferentes técnicas de paralelización utilizadas para minimizar los efectos del problema de asignación de estados (sección 4). En la sección 5 se describen brevemente los resultados que ofrece la herramienta de asignación de estados CCEDAF, para finalizar con las mejoras previstas que se implementarán en un futuro próximo (sección 6) y conclusiones finales (sección 7).

2. DISEÑO DE AUTÓMATAS DE ESTADOS FINITOS

La calidad final de un circuito dependerá de la experiencia e intuición del diseñador, además de la complejidad del propio sistema. La experiencia se alcanza con ejemplos y con el propio diseño de autómatas, de tal forma que dos diseñadores pueden crear diferentes circuitos secuenciales síncronos que describen un mismo autómata de estados finitos, de forma que la mejor implementación será aquella cuya función de coste sea menor.

2.1. Diseño del diagrama de flujo y tabla de transiciones

A partir de la descripción textual del sistema a diseñar, que puede ser incompleta y/o incoherente, se crea un diagrama de estados (ver figura 1) que explique de manera gráfica el funcionamiento del autómata. En esta fase, imposible de automatizar, la comprensión, experiencia e intuición del diseñador son fundamentales para un diseño correcto. En la tabla 1 se muestra una tabla de transiciones de una máquina de estados finitos con cinco estados (A, B, C, D, E), una entrada (I_0), y dos salidas (Z_0, Z_1).

2.2. Minimización de estados

De un autómata sólo interesa la salida que va produciendo, no el número de estados que necesita para hacerlo. Por lo tanto, es claro el interés de minimizar la tabla de estados para obtener

el número mínimo de estados sin degradar el autómata. En el mejor de los casos podríamos disminuir el número de biestables necesarios, simplificando tanto el proceso de creación de la tabla de excitación como el circuito final.

Para realizar esta minimización de la tabla de estados existen dos métodos: el método de Huffman [18] y el método de las tablas de implicación [12]. Cabe decir que minimizar supone aplicar un algoritmo tan sencillo como largo y tedioso, que está implementado en la mayoría de los entornos computacionales de diseño digital, incluyendo el BOOLE-DEUSTO [13].

Una vez minimizada la tabla de estados, hay que decidir el número de variables de estado. La cuantía de las variables de estado se establece de forma que haya el menor número posible. Por tanto, si en la tabla a sintetizar hay *m* estados, el número de variables de estado *n* vendrá dado por la relación: $2^{n-1} < m \leq 2^n$

2.3. Codificación o asignación de estados

El coste de la implementación de un circuito secuencial síncrono depende principalmente de la asignación de estados realizada. El problema de encontrar la asociación entre estados y sus correspondientes códigos binarios de forma que el coste de implementación sea mínimo se denomina problema de asignación o codificación de estados.

Estado	Asignaciones	
	Nº 1	Nº 2
A	000	000
B	100	100
C	011	110
D	010	011
E	111	010

Tabla 2.- Posibles asignaciónes

El coste de implementación de una ecuación B_i representada en forma de suma de productos, viene dada por la ecuación (1), donde *k* corresponde al número de términos producto de la ecuación, y siempre que *m* sea mayor a 1.

$$C(B_i) = \sum_{j=0}^{k-1} P_j(B_i) + O(B_i) \quad \text{donde} \quad P_j(B_i) = \begin{cases} m & \text{Si el término } j \text{ de } B_i \text{ tiene } m \text{ variables} \\ 0 & \text{Si el término } j \text{ de } B_i \text{ tiene 1 variable} \end{cases} \quad O(B_i) = \begin{cases} m & \text{Si } B_i \text{ tiene } m \text{ términos} \\ 0 & \text{Si } B_i \text{ tiene 1 término} \end{cases} \qquad (1)$$

Dado un conjunto de ecuaciones booleanas S = { B_0, B_1, ... B_{n-1}}, su coste se calcula con la expresión (2), donde C(B_i) es el coste de la ecuación B_i y k_i es el número de términos producto en la ecuación B_i, siendo *i* mayor que 1.

$$C(S) = \sum_{i=0}^{n-1} \left[C(B_i) - \sum_{j=0}^{k_i-1} R_j(B_i) \right] \quad \text{donde} \quad R_j(B_i) = \begin{cases} P_j(B_i) & \text{Si el término } j \text{ de } B_i \text{ se repite en otro término distinto.} \\ 0 & \end{cases} \qquad (2)$$

El coste de un circuito secuencial síncrono es igual al coste del conjunto de ecuaciones booleanas formadas por las ecuaciones que generan el próximo estado más las ecuaciones que generan el valor de salida del circuito.

En el ejemplo anteriormente presentado, para los dos diferentes asignaciones de estados propuestas en la tabla 1, la asignación nº 1 tiene un coste de 31, mientras que la asignación nº 2 tiene un coste de 13. Este ejemplo demuestra que elegir una asignación de estados apropiada reduce enormemente el coste de la implementación.

Sin embargo, determinar la asignación de estados que corresponde al coste mínimo de un circuito no es un problema trivial. Un conjunto de reglas heurísticas recopiladas a lo largo de los años se han propuesto para permitir implementaciones de circuitos secuenciales síncronos por muchos diseñadores [2] [3]. Antes de presentar estas reglas, mostramos algunas definiciones necesarias:

Atribución de estado, $A(S_i)$: corresponde a la codificación binaria asignada al estados S_i.

Sucesores de un estado, $Suc(S_k)$: es el conjunto de todos los estados sucesores a un estado S_k. Observando la tabla 1, obtenemos los siguientes sucesores:

$Suc(A) = \{B, C\}$ $Suc(B) = \{D, E\}$ $Suc(C) = \{D, E\}$ $Suc(D) = \{E\}$ $Suc(E) = \{A\}$

Predecesores de un estado, $Pred(S_i, I_a)$: corresponde al conjunto de todos los estados predecesores de un estado S_i con una condición de entrada I_a. De tal forma, que un estado S_i es denominado predecesor de un estado S_k si hay una transición desde el estado S_i al estado S_k para cualquier valor de la variable de entrada I_a. Según nuestro ejemplo de la tabla 1, obtendremos los siguientes predecesores:

$Pred(E, I_0 = 0) = \{B, C, D\}$ $Pred(D, I_0 = 1) = \{B, C\}$

Particiones de la salida, $O(Z_k)$: corresponde al conjunto de particiones de una salida Z_k, de tal forma que cada una de las salidas de una máquina de estados finita divide los estados del circuito en dos subconjuntos. Para circuitos tipo Moore, una salida es denotada por $Z_k(S_i)$, mientras que en un circuito tipo Mealy, la salida es $Z_k(S_i, I_a)$, debido a que en una maquina tipo Moore la salida es una función del estado, mientras que en un circuito tipo Mealy la salida también depende de los valores de la entrada. De la tabla 1 obtenemos las siguientes particiones de salida

$O(Z_0) = \{ (B, C), (A, D, E) \}$ $O(Z_1) = \{ (B, D), (A, C, E) \}$

Estados asociados o adyacentes: Los estados S_i y S_j están asociados entre sí cuando se cumple alguna de las siguientes situaciones:

1.- Si ambos estados son sucesores de un estado S_k determinado. Por ejemplo, las siguientes parejas de estados de nuestro ejemplo son adyacentes: (B, C), y (D, E).

2.- Si ambos estados son predecesores de un estado S_x para el mismo valor de entrada. En nuestro ejemplo, no existe ninguna pareja de estados que cumpla esta situación.

3.- Cuando ambos estados pertenecen a la misma partición de una salida Z_m. En nuestro ejemplo, la pareja (A, E) son adyacentes entre sí.

Distancia entre estados, D(Si, Sk): La distancia entre dos estados S_i y S_k se define como la distancia Hamming entre $A(S_i)$ y $A(S_k)$, de tal forma, que la distancia

Hamming entre dos código binarios a y b se define como el número de bits en la misma posición con fase opuesta:

$$D(S_i, S_k) = \sum_{i=1}^{n} S_i \oplus S_j \qquad \text{donde } n \text{ es el número de bits.} \qquad (3)$$

Una vez asentadas las definiciones anteriores, presentamos las reglas heurísticas utilizadas para estudiar la adyacencia entre estados de tal forma que el coste de un circuito secuencial síncrono se reduzca al mínimo cuando la asignación de estados se realice de forma que minimice la distancia entre estados:

1.- Estados que están en el mismo conjunto de sucesores de un estado determinado. Según nuestro ejemplo, destacamos las parejas (B, C) y (D, E).

2.- Estados que están en el mismo conjunto de predecesores de un estado dado para un mismo valor de entrada. Según nuestro ejemplo, no hay parejas de estados que cumplan esta regla.

3.- Estados que están en la misma partición con un mismo valor de salida. Según nuestro ejemplo, destacamos la pareja (A, E).

Observe que las parejas de estados (B, C), (D, E) y (A, E) están asociadas entre sí más frecuentemente que otras parejas. De esta forma, en una buena asignación de estados para la máquina de estados finitos de la tabla 1, la distancia Hamming entre estos estados debería ser pequeña. La asignación nº 1 de la tabla 1 tiene D(B, C) = 3, D(D, E) = 2 y D(A, E) = 3, mientras que la asignación nº 2 tiene D(B, C) = 1, D(D, E) = 1 y D(A, E) = 1. Claramente, la asignación nº 2 tiene una menor distancia Hamming, lo que explica porqué el coste de implementación del circuito con esta asignación es mínima.

3. AUTOMATIZACIÓN

La herramienta CCEDAF – *Codificador Combinatorio de Estados para el Diseño de Autómatas Finitos* – orientada a alumnos de laboratorio de Tecnología de Computadores, permite al diseñador enfocar sus esfuerzos en la fase inicial del diseño del diagrama de estados (imposible de automatizar) y en la fase final de simulación e implementación del sistema digital.

En la figura 2 se muestran las distintas fases que integran la herramienta diseñada, donde se resaltan con línea discontinua, el problema de minimización de estados y el problema de codificación de estados. Con relación al problema de minimización de estados, supone aplicar un algoritmo tan sencillo como largo y tedioso, que se encargue de reducir el número de estados (k), que generalmente consume poco tiempo de computación.

Sin embargo, el problema de codificación de estados consiste en evaluar todos los posibles códigos de n bits, tal que $2^n \leq k$, memorizando aquellas combinaciones que requieran un menor número de puertas. El número de combinaciones total se puede reducir considerablemente si aplicamos la adyacencia entre estados, de tal forma que cuando el número de estados es relativamente alto (por ejemplo, $k \geq 9$ estados), el número de bits necesarios aumenta (en este caso, $n \geq 4$ bits), por lo que el número de combinaciones, teniendo en cuenta las posibles

adyacencias entre estados, se dispara exponencialmente a 10.810.800 codificaciones diferentes, lo que evidentemente se traduce en una enorme carga computacional.

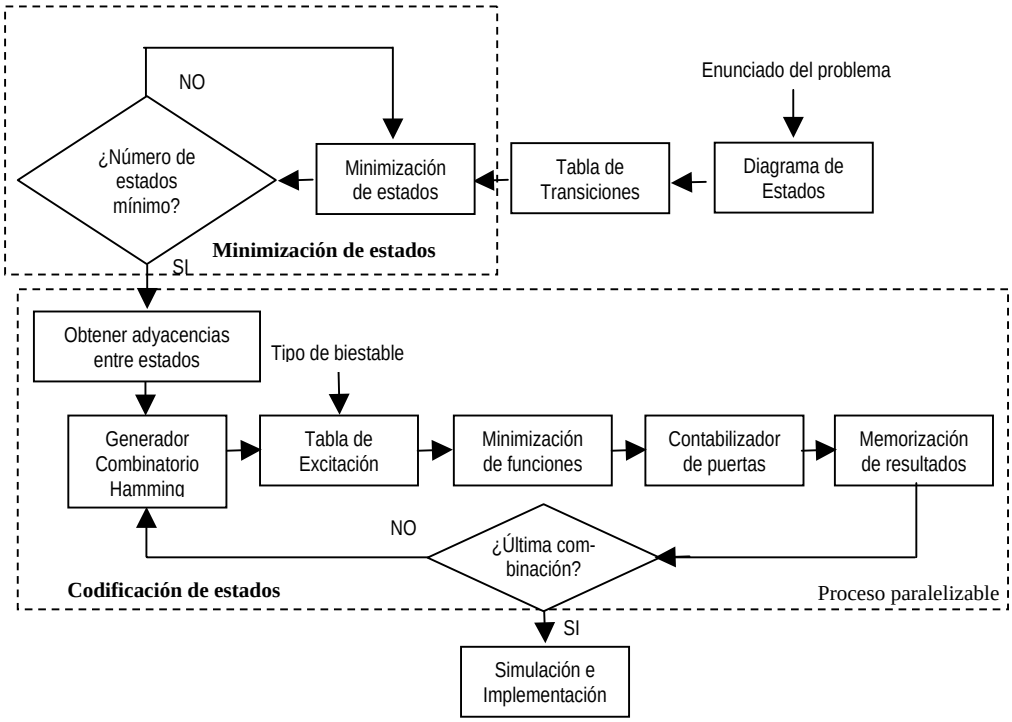

Figura 2.- Esquema general de CCEDAF.

El corazón de CCEDAF es el generador combinatorio Hamming, encargado de medir la distancia Hamming de todas las posibles combinaciones cumpliendo el máximo número de adyacencias entre estados como sea posible, y memorizando aquellas combinaciones con menor distancia Hamming.

Únicamente aquellas combinaciones con menor distancia Hamming son asignadas a los estados, para confeccionar la tabla de excitación y minimizar las funciones según el tipo de biestable seleccionado.

La herramienta CCEDAF ha sido implementada en lenguaje orientado a objetos C++, ejecutándose tanto en sistema operativo Linux como en Windows.

4. PARALELIZACIÓN

El mejor modo de resolver un problema complejo es dividirlo en subproblemas más pequeños, por este motivo una forma de mejorar el rendimiento de la herramienta CCEDAF [10] es dotarlo de la capacidad de evaluar simultáneamente distintas codificaciones, para lo cual se deberá disponer de un conjunto de procesos (o subprogramas) que evaluen un determinado rango de combinaciones y que se ejecuten de forma paralela, de tal forma que el conjunto de todos los procesos abarquen todas las combinaciones posibles.

El algoritmo 1 corresponde a un extracto del código fuente de CCEDAF, donde se muestra la función que contiene el análisis secuencial encargado de evaluar todas las posibles combinaciones.

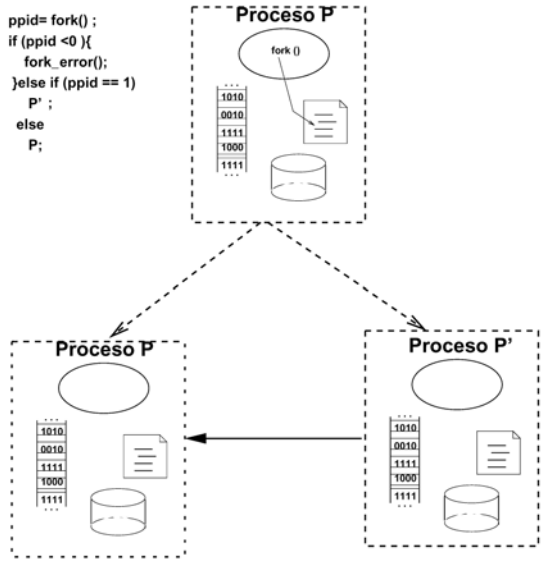

Figura 3.- Esquema de programación de procesos

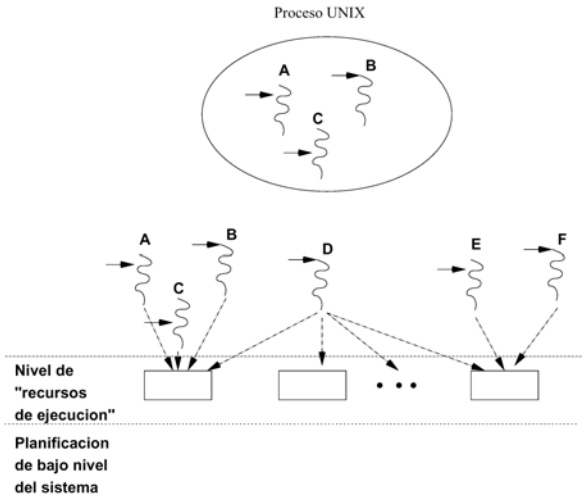

Figura 4.- Esquema de programación de hebras

La ejecución paralela de estos procesos se puede realizar desde dos puntos de vista completamente diferentes: Por un lado, la creación de procesos (o hebras) que se ejecutan al mismo tiempo en el mismo procesador, y por otro lado, ejecutar procesos en distintos procesadores de un cluster heterogéneo.

```
void AnalisisSecuencial() {
    vector<string> funciones, minfunciones; // funciones a minimizar
    int pos=0;
    bool primeravez=true;
    // Inicializar y configurar los atributos de las hebras
    while(GenerarAsignacion(0, primeravez)) {
        Comb_Actual++;
        AsignarEstados(0);
        CrearTablaExcitacion(0);
        CrearFunciones(&funciones, 0);
        Minimizar(funciones, EntBiestable, &minfunciones);
        funciones.erase(funciones.begin(), funciones.end());
        CalcularNumeroPuertas(minfunciones,0);
        // Eliminar las funciones
        minfunciones.erase(minfunciones.begin(),minfunciones.end());
        primeravez=false;
    }
}
```

Algoritmo 1.- Programación secuencial de CCEDAF

4.1. Ejecución pseudo-paralela

En este sentido surge un dilema, puesto que se pueden crear varios procesos que se ejecuten en el mismo procesador o crear varias hebras del mismo proceso que se ejecutan de forma paralela.

La programación de procesos (figura 3) genera copias del programa original, totalmente independiente, proporcionando protección (en el caso de direcciones de memoria comunes), sin embargo, surgen los siguientes inconvenientes: el coste de cambio entre múltiples procesos es elevado, el planificador de procesos suele manejar eficientemente un número limitado de procesos y las variables de sincronización compartidas entre procesos suelen ser lentas.

Una buena alternativa es la programación de hebras (figura 4), donde se generan diferentes secuencias independientes (hebras) de la ejecución de un mismo programa. La interfaz POSIX [14] permite a más de una hebra estar activa dentro de un proceso, del cual se comparten recursos y memoria.

La programación hebrada se ha realizado con hebras POSIX, mostrando en el algoritmo 2 un extracto de la programación hebrada de CCEDAF, donde la función *EnviarHebra()* tiene una estructura muy parecida a la función *AnalisisSecuencial()* del algoritmo 1, con la única salvedad que cada hebra evalúa únicamente aquellas combinaciones cuyas posiciones coinciden con $IdHeb(numheb+1)$, donde *IdHeb* es el número identificador de la hebra.

```
// Permite evaluar todas las combinaciones repartiéndolas entre varias hebras.
void AnalisisMultihebras() {
      char IdHebra[5];
      pthread_t hebra[2]; // declaración de una hebra
      pthread_attr_t atributo;
      int rc, status, j;
      // Inicializar y configurar los atributos de las hebras
      pthread_attr_init(&atributo);
      pthread_attr_setdetachstate(&atributo, PTHREAD_CREATE_JOINABLE);
      for(j=0; j<Numhebras; j++) {
            sprintf(IdHebra,"%d",j);
            pthread_create(&hebra[j], &atributo, EnviarHebra, IdHebra);
      }
      for(j=0; j<Numhebras; j++) { // esperar a que terminen las hebras
            rc = pthread_join(hebra[j], (void **)&status);
             f (rc) {
                  cout << "ERROR; return code from pthread_join() is " << rc;
                  exit(-1);
            }
      }
      // Liberar los atributos
      pthread_attr_destroy(&atributo);
}
```

Algoritmo 2.- Programación hebrada de CCEDAF

4.2. Programación paralela multi-procesador

Los autómatas de estados finitos definidos con un número superior a ocho estados, requieren de un elevado tiempo de ejecución (horas o incluso días). Por este motivo, se precisa de una implementación paralela que distribuya la carga computacional entre distintos procesadores de un cluster heterogéneo.

La implementación paralela se ha programado con librerías MPI – *Message Passing Interface* – [15], de tal forma que diferenciamos entre dos tipos de procesadores; por un lado el procesador principal (procesador 0) encargado de ejecutar de forma secuencial tareas con poca carga computacional, tales como leer la tabla de transiciones del fichero de entrada, minimización de estados y estudiar adyacencias entre estados.

Por otro lado, todos los procesadores, incluido el procesador principal, se encargan de evaluar las combinaciones cuyas posiciones coincidan con $Idproc(numproc+1)$, donde *Idproc* es el número identificador del procesador. Esta metodología permite reducir la comunicación entre procesos, cuello de botella en toda implementación paralela de paso de mensajes, debido a que la única información que los procesadores secundarios requieren del procesador principal es una lista con las parejas de estados que cumplen las reglas de adyacencia entre estados.

Una vez que todos los procesadores, incluido el procesador principal, han terminado de evaluar sus correspondientes combinaciones, devuelven las funciones minimizadas de las combinaciones con menor coste de implementación. Finalmente, el procesador principal selecciona de entre todas las combinaciones recibidas de los procesadores secundarios aquellas cuyo coste de implementación sea menor.

5. FUNCIONAMIENTO

La ejecución de CCEDAF requiere de un fichero de texto que contenga la tabla de transiciones, así como una serie de parámetros que describan el autómata de estados finitos a diseñar. En el algoritmo 3 se puede observar un ejemplo del fichero de entrada que describe el funcio-

namiento de un autómata de estados finitos con 1 entrada, 2 salidas y 5 estados (A, B, C, D, E, y F).

```
#Entradas = 1; // nº de entradas
#Estados = 5; // nº de estados
#Salidas = 2; // nº de salidas
#Elementos = 10; // nº elementos
[0 A B 00] // Tabla de transiciones
[0 B E 11]
[0 C E 10]
[0 D E 01]
[0 E A 00]
[1 A C 00]
[1 B D 11]
[1 C D 10]
[1 D E 01]
[1 E A 00]
```

Algoritmo 3.- Fichero de entrada (ejemplo.txt)

Biestable	AND	OR	NOT
D	5	2	3
	7	1	2
T	6	4	4
	7	4	3
RS	5	1	4
	7	2	3
	8	1	3
JK	6	1	4
	7	1	3
	5	2	4

Tabla 3.- Número de puertas necesarias.

Para la ejecución de la aplicación se debe introducir en la línea de comandos la siguiente orden:

ccedaf *fichero.txt tipo_biestable num_resultados [mensajes] [opción] [numhebras]*

donde *fichero.txt* corresponde al nombre del fichero de texto de entrada encargado de describir la tabla de transiciones de estados. Con relación al *tipo_biestable* se puede elegir entre los cuatro tipos de biestables disponibles: D, T, SR, o JK. Finalmente, con el parámetro *num_resultados* se especifica el número total de los mejores resultados, es decir, aquellas combinaciones que ofrezcan el menor número de puertas posibles.

El resto de los parámetros opcionales están orientados a mejorar el rendimiento de la aplicación; por un lado, *mensajes* indica si se desea visualizar por pantalla los resultados intermedios de todas las combinaciones, su valor por defecto es 0 permitiendo acelerar la ejecución al no visualizar ningún mensaje intermedio. El siguiente parámetro *opción* permite seleccionar el modo de ejecución: 0 - monoprocesador, 1 – multi-hebras, y 2 – multiprocesador. En el caso concreto de seleccionar una ejecución multihebrada, es necesario indicar el número de hebras ejecutadas, a través del parámetro *numhebras*.

Finalmente, recordar que para la ejecución multiprocesador, el usuario debe ejecutar el siguiente comando:

mpirun –np numprocesadores ***ccedaf*** *fichero.txt tipo_biestable num_resultados ....*

6. RESULTADOS

La ejecución secuencial, multihebrada, o multi-procesador de CCEDAF devuelve un fichero de texto donde se incluyen *num_resultados* combinaciones que requieren menor coste para el tipo de biestable seleccionado.

En la tabla 3 se muestran el número de puertas AND, OR y NOT necesarias para implementar las combinaciones con menor coste propuestas por CCEDAF para nuestro ejemplo. El diseñador debería comparar los costes de implementación de las distintas combinaciones, para los distintos tipos de biestable, y seleccionar la opción más adecuada para la simulación y/o implementación. En nuestro caso, la implementación óptima podría ser con biestables tipo SR

más un módulo combinacional con 5 puertas AND, 1 puerta OR y 4 puertas NOT. Sin embargo, este módulo combinacional se puede sustituir por un módulo combinacional programable (ROM, PAL o PLA).

Aunque el tiempo requerido depende de la modalidad de ejecución de CCEDAF escogida, en principio tanto la ejecución secuencial como la ejecución multi-hebrada están orientadas para diseñar autómatas de estados finitos con menos de nueve estados. La ejecución multihebras aumenta considerablemente el rendimiento, respecto de la ejecución secuencial, en ordenadores bi-procesadores, muy comunes actualmente, mientras que la ejecución multiprocesador está más orientada al diseño de autómatas con más de ocho estados.

Si en la ejecución multi-hebrada aumentamos el número de hebras, el rendimiento decrece, observando que el número óptimo de hebras en un sistema monoprocesador es de dos o tres hebras, obteniendo entre un 15% y 20% de reducción en el tiempo de ejecución respecto de la ejecución secuencial. De tal forma que cuando seleccionamos más de tres hebras, el tiempo de ejecución de la implementación multi-hebrada es superior al de la implementación secuencial.

7. MEJORAS FUTURAS

Las modificaciones que se tienen previsto implementar en un futuro próximo van encaminadas en dos importantes aspectos: incrementar la velocidad de ejecución y ampliar la gama de plataformas de implementación.

Respecto a las posibles mejoras destinadas a aumentar la velocidad de ejecución diferenciamos entre dos vías de actuación, por un lado implementar un nueva modalidad de ejecución CCEDAF orientada a un multiprocesamiento paralelo híbrido, que aparte de repartir la carga computacional entre los distintos procesadores del cluster, también sea capaz de crear distintas hebras dentro de un mismo procesador. La otra posible vía de actuación se centra en la generación de códigos binarios, donde se pueden mejorar y ampliar los métodos de restricciones, entre los que destacamos la adyacencia entre estados que permitan reducir el campo de búsqueda.

Finalmente, una posible modificación estructural de mayor envergadura estaría orientada a implementaciones multi-nivel para diferentes tecnologías con la posibilidad de realizar múltiples funciones de coste de compromiso entre área (número de puertas), retardo, potencia, complejidad de interconexión, etc, introduciendo un nuevo tipo de criterio en el proceso de síntesis secuencial, lo que complica enormemente el proceso. Además, cada plataforma de implementación (puertas lógicas, PLAs, PALs, CLBs, CPLDs, o FPGAs) requieren de parámetros que dependen de la plataforma específica de implementación.

8. CONCLUSIONES

Con esta herramienta se pretende que los alumnos de informática, y cualquier diseñador en general, pueda reducir el tiempo dedicado al diseño de autómatas de estados finitos, de tal forma que puedan centralizar sus esfuerzos en la generación del diagrama de transiciones a partir del enunciado del problema, y sobre todo a facilitar la simulación/implementación del circuito secuencial síncrono. Sin embargo, la principal ventaja de que el circuito secuencial

síncrono requiera de un pequeño número de puertas lógicas facilita sobre todo la detección de averías.

10. BIBLIOGRAFÍA

[1] W.S. Humphrey. *Switching circuits with computer applications*. McGrawHill, New-York, 1958.

[2] D. B. Armstrong. *A programmed algorithm for assigning internal codes to sequential machines*. IRE Transactions on Electronic Computers, pages 466-472, August, 1962.

[3] D. J. Comer. *Digital logic and state machine design*. CBS College Publishing, New York, 1984.

[4] T. A. Dolotta and E. J. McCluskey. *The coding of internal states of sequential circuits*. IEEE Trans. on Electronic Computers, pages 549-562, October, 1964.

[5] G. de Micheli, R. K. Brayton, and A. Sangiovanni-Vincentell. *Optimal state assignment for finite state machines*. IEEE Trans. on CAD, pages 269-284, 1985.

[6] L. Józwiak. *Efficient suboptimal state assignment of large sequential machines*. Proc. of EDAC, pages 536-541, 1990.

[7] L. Józwiak. *An efficient heuristic method for state assignment of large sequential machines*. Journal of circuits, systems and computers, 2(1) pages 21-26, 1992.

[8] T. Villa y A. Sangiovanni-Vincentell. *NOVA: state assignment of finite state machines for optimal two-level logic implementation*. IEEE Trans. on CAD, pages 905-924, 1990.

[9] S. Devadas, H. TonyMa, A. R. Newton y A. Sangiovanni-Vincentelli. *MUSTANG: state assignment of finite state machines for optimal multi-nivel logic implementation.* Proc. of Int. Conf. on CAD, pages 16-19, 1987.

[10] J. F. Sanjuan Estrada, I. García Fernández y J. A. Álvarez Bermejo. *Automatización del problema de asignación de estados en el diseño de sistemas secuenciales sincronos*. JENUI 2004.

[11] J. F. Sanjuan Estrada, I. García Fernández y J. A. Álvarez Bermejo. *CCEDAF: Codificador Combinatorio de Estados para el Diseño de Autómatas Finitos*. JENUI 2004.

[12] A. Lloris Ruíz, A. Prieto Espinosa y L. Parrilla Roure. *Sistemas digitales*. McGraw-Hill, 2003.

[13] J. García Zubia, J. Sanz Martínez y B. Sotomayor. *BOOLE-DEUSTO, la aplicación para sistemas digitales*. VII Jornadas de Enseñanza Universitaria de la Informática (JENUI), 2001.

[14] B. Nichols, D. Buttlar y J. Proulx Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. Ed. O'Reilly Nutshell. September, 1996.

[15] M. Snir, S. Otto, S. Huss-Lederman, O. Walker, y J. Dongorra. *MPI: The complete reference*. Machachusetts Institute of Technology. 1996.

[16] A. E. A. Almaini, J. F. Miller, P. Thomson, y S. Billina. *State assignment of finite state machines using a genetic algorithm*. IEEE Proc. on Computers and Digital Techniques, pages 279-286, July 1995.

[17] S. Chattopadhyay and P. Pal Chaudhuri. *Genetic algorithm based approach for integrated state assignment and flipflop selection in finite state machine synthesis*. Proc. of Int. Conf. on VLSI Design, pages 522-527, 1997.

[18] S. H. Unger. *Asynchronous sequential switching circuits*. New York, Wiley-Interscience, 1969.

[19] R. Brayton et al. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, Boston, MA, 1984.