Optimización de carga de datos en un banco de pruebas de aviónica

José Daniel García y Jesús Carretero

Resumen— **Este artículo presenta las técnicas de optimización utilizadas para acelerar la carga de definiciones de señales de un banco de pruebas de aviónica. La carga de datos en formato XML puede presentar problemas de rendimiento cuando se trata de grandes volumenes de datos. Bajo esas condiciones, es necesario buscar alternativas que permitan cargar los datos de forma más eficiente, sin imponer restricciones a los generadores de los datos utilizados como entrada. El artículo analiza posibles optimizaciones y valora las ventajas e inconvenientes de cada solución.**

Palabras clave— **aviónica, optimización, carga de archivos de datos, ADA 95, XML.**

I. Introducción

En el proceso de desarrollo de sistemas de aviónica una parte muy importante es la realización de las pruebas que aseguren el correcto funcionamiento de los sistemas desarrollados. Uno de los mecanismos de verificación que se suele utilizar es la conexión de los sistemas a un banco de pruebas que estimula los sistemas y recoge sus salidas. El nmero de señales que se utilizan en la prueba de un sistema típico suele ser bastante alto (entre 10.000 y 100.000 señales). La construcción de un banco de pruebas con cierto grado de genericidad que permita su reutilización en diversos proyectos implica necesariamente la parametrización del banco en cuanto al tipo y número de señales generadas [3].

La carga en el banco de pruebas del archivo de definición de señales puede suponer una importante degradación en el rendimiento del sistema durante la iniciación del mismo. Debe tenerse en cuenta que un archivo de definición de señales de 100.000 señales puede necesitar un almacenamiento superior a los 200 MB. Por tanto es muy importante, para que el banco de pruebas sea operativo, que la carga de las definiciones de señales sea lo más rápida posible. De lo contrario, la repetición de pruebas implica una espera en el proceso de iniciación del orden de minutos o incluso de horas.

II. Sistema original

La figura 1 muestra una visión general del banco de pruebas. La definición de las pruebas a realizar es una de las principales entradas al banco de pruebas y se representa mediante un conjunto de archivos XML. De estos archivos, el mayor volumen de datos viene representado por un archivo XML que contiene la definición de las señales de entrada y de salida del banco de pruebas. El banco de pruebas está completamente escrito en ADA 95 [1] lo que supone un conjunto de restricciones sobre las alternativas de mejora del rendimiento. El uso de ADA 95 en el sector de la aviónica es una práctica bastante habitual desde mediados de los 90 [6].

Grupo de Arquitectura y Tecnología de los Computadores, Departamento de Informática, Universidad Carlos III de Madrid, E-28911 Leganés, Madrid (España). Correo electrónico: jdaniel,jcarrete@arcos.inf.uc3m.es.

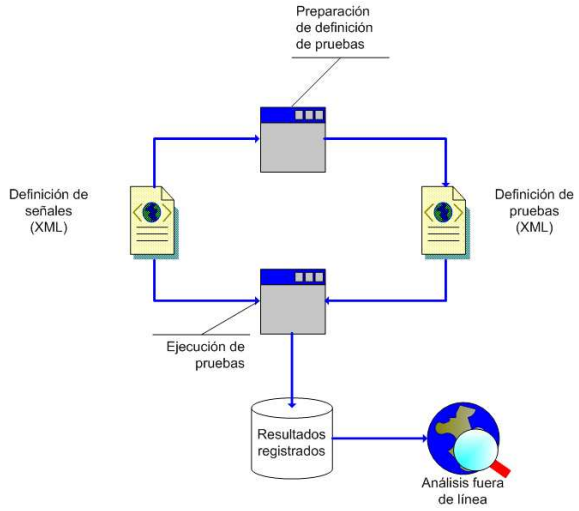

Fig. 1. Interfaces del banco de pruebas

Originalmente, el banco carga las señales mediante el uso de un parser XML. Con esta solución es necesario realizar primero una lectura y análisis del archivo XML, para posteriormente construir en memoria las estructuras de datos necesarias para representar la información [4].

III. Alternativas para la optimización

Para resolver el problema de la mejora del rendimiento de la iniciación del banco de pruebas, se identificaron varias alternativas posibles:

- Mejora del parser XML.
- Preprocesado de los datos XML a otro formato.
- Paralelización de la carga.

A. Mejora del parser

Puesto que el problema de la falta de rendimiento se produce durante la lectura y análisis del archivo XML, la primera alternativa pasa por buscar mejoras en el propio parser. Analizados los productos disponibles para ADA 95, ninguno de los productos estudiados ofreció mejoras sustanciales en el procesado de grandes volumenes de datos. Otra solución habría sido la construcción de un parser ad-hoc o la modificación de uno del que el código fuente estuviese disponible. No obstante, esta solución no parece efectiva en costes.

B. Preprocesado de los datos XML a otro formato

El problema del rendimiento parece inherente a la naturaleza del formato de los datos (XML). Aunque los datos se proporcionan en XML al banco de pruebas, dichos datos cambian con poca frecuencia, por lo que es admisible realizar un preprocesado de los archivos XML, de forma que el banco de pruebas utilice otro formato que permita un tiempo de carga menor.

En la sección IV se describe en detalle esta alternativa, que permite acelerar la carga de los datos. No obstante, debe tenerse en cuenta que los datos deben preprocesarse cada vez que se produce un cambio en los mismos y que esto es un inconveniente de esta elección.

C. Paralelización de la carga

Independientemente del formato en que se almacenen los datos, el proceso de carga de los mismos es un buen candidato para la paralelización. Un algoritmo de carga de datos paralelo, incrementará el porcentaje de uso del procesador. Téngase en cuenta que al tratarse de un proceso de carga habrá un alto número de operaciones de entrada salida, con lo que un algoritmo paralelo debe presentar ventajas con respecto a un algoritmo secuencial. Además, al tratarse de la carga de datos en memoria, cada hilo de ejecución puede ocuparse de la carga de un subconjunto de datos, con lo que la competición por recursos es prácticamente inexistente.

IV. Preprocesado mediante serialización

El uso de un formato ad-hoc permite definirlo de la forma más conveniente para el uso que se le desea dar. En el caso descrito, parece claro que un formato binario ofrece ventajas sobre un formato de texto como XML. Una posibilidad sería la definición de un formato binario específico. No obstante ADA 95 ofrece otra posibilidad mediante el paquete `Ada.Streams` [5]. Mediante este paquete es posible hacer uso de una técnica conocida como serialización. Esta técnica consiste en volcar a un archivo una imagen de una estructura de datos de forma que ésta pueda ser recuperada posteriormente.

Una vez decidido el formato del archivo intermedio, es necesario seleccionar también que datos se van a almacenar en dicho archivo. Básicamente se presentan dos opciones:

- Almacenar el árbol generado por el parser XML.
- Almacenar las estructuras de datos finales.

Cuando se utiliza un parser XML para cargar datos a partir de un archivo en dicho formato, se va generando en memoria un árbol que es una representación en memoria del archivo XML. Posteriormente, la aplicación puede realizar manipulaciones sobre los datos (p. ej. leer los datos del árbol en memoria para rellenar las estructuras de datos que la aplicación necesite utilizar). Por tanto una posibilidad sería la serialización del propio árbol XML. Con esta solución se evitaría tener que realizar un análisis sintáctico sobre el archivo XML en cada ocasión, pero seguiría siendo necesario rellenar las estructuras de datos del programa cada vez.

La otra posibilidad consistiría en almacenar directamente las estructuras de datos finales. Con esta solución, se incrementa el tiempo de preprocesado. A cambio el tiempo de carga de los datos en el sistema queda reducido.

De las dos soluciones propuestas la segunda es preferible a la primera, ya que en el caso que nos ocupa, y dada la baja tasa de modificaciones de las definiciones de señales, es preferible sacrificar el tiempo de preproceso si con ello se consigue mejorar el tiempo de carga de los datos en el sistema.

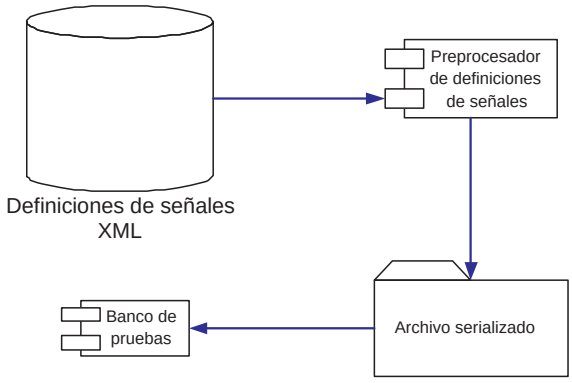

Fig. 2. Preprocesado de las definiciones de señales

V. Implementación de la serialización

La serialización de los datos que posteriormente serán cargados por el banco de pruebas implica la necesidad de que los datos sean susceptibles de ser serializados. Por un lado se necesita un proceso de serialización y, por otra parte, un proceso de lectura de los datos serializados. Partiendo de un programa original que carga los datos directamente de un archivo XML, se pueden generar los dos programas necesarios con pequeñas modificaciones de código.

Además, es necesario hacer que los datos asociados al proceso de serialización y recuperación soporten dicha característica.

```
procedure T'Write(
Stream: access Root_Stream_Type'Class;
Item: in T);
procedure T'Read(
Stream: access Root_Stream_Type'Class;
Item: in T);
procedure T'Output(
Stream: access Root_Stream_Type'Class;
Item: out T);
function T'Input(
Stream: access Root_Stream_Type'Class)
return T;
```

Fig. 3. Atributos utilizados en la serialización.

En ADA 95, todos los tipos no limitados soportan cuatro atributos: `Write`, `Read`, `Output` y `Input` (fig. 3).

`Write` y `Read` permiten almacenar y recuperar, respectivamente un valor de un cierto tipo. `Output` e `Input` tienen el mismo efecto, excepto que en el caso de utilizarse con arrays o registros con discriminantes, permiten adems el almacenamiento y recuperación de información adicional necesaria. En el caso de jerarquías de clases (tipos etiquetados en ADA 95) se pueden utilizar tam-bién `T'Class'Output` y `T'Class'Input`.

Los atributos anteriormente mencionados existen para todos los tipos. No obstante, su implementación por defecto es unicamente útil para tipos triviales. En el resto de los casos se pueden redefinir los atributos que sea necesario para proporcionar una implementación más adecuada a las necesidades.

Un caso típico en el que la implementación por defecto de los atributos de serialización no es adecuado es cuando hay presencia de punteros (tipos `access`). En dicho caso, es necesario establecer alguna estrategia de volcado y recuperación de la información. Para implementar dicha estrategia, puede ser necesario redefinir algún atributo.

```
type Signal_Ptr_Type is
  access all Signal'Class;

procedure Write(
  Stream : access Root_Stream_Type'Class;
  Item : in Signal_Ptr_Type);
for Signal_Ptr_Type'Write use Write;

procedure Read(
  Stream : access Root_Stream_Type'Class;
  Item : out Signal_Ptr_Type);
for Signal_Ptr_Type'Read use Read;
```

Fig. 4. Especificación de la redefinición de los atributos Read y Write.

La redefinición de atributos implica:

1. La especificación de un par de procedimientos que implementen el par de atributos a modificar.
2. La asociación de los procedimientos con los correspondientes atributos.
3. La implementación de los citados procedimientos.

Los pasos 1 y 2 se realizan en la especificación del correspondiente paquete tal y como se muestra en la figura 4.

El paso 3 se realiza en la implementación del correspondiente paquete, tal y como se muestra en la figura 5.

Obsérvese, que con las definiciones expuestas, será necesario que las referencias a las señales se almacenen utilizando el atributo `T'Class'Output` y se recuperen mediante el atribut `T'Class'Input`. De otra forma, no sería posible realizar adecuadamente la serialización en presencia de polimorfismo.

```
procedure Write(
  Stream : access Root_Stream_Type'Class;
  Item : in Signal_Ptr_Type ) is
begin
  if Item = null then
    Boolean'output(Stream, false);
  else
    Boolean'output(Stream, true);
    Signal'Class'output(Stream, Item.all);
  end if;
end Write;

procedure Read(
  Stream : access Root_Stream_Type'Class;
  Item : out Signal_Ptr_Type ) is
    signal_found : Boolean :=
      Boolean'input(Stream);
begin
  if signal_found then
    Item := new Signal'Class'(
      Signal'Class'Input(Stream));
  else
    Item := null;
  end if;
end Read;
```

Fig. 5. Implementación de la redefinición de los atributos Read y Write.

El banco de pruebas almacena las señales en memoria mediante una tabla hash que utiliza listas enlazadas para las colisiones. Una posible definición de dicha estructura de datos se presenta en la figura 6.

```
type Element_Type;
type Pointer_Type is
  access all Element_Type;
type Element_Type is record
    data : Signal_Pointer_Type;
    ident : Identifier_Type;
    next : Pointer_Type;
end record;
type Hash_Table_Type is array(0..MAX-1) OF
  Pointer_Type := (others => null);
```

Fig. 6. Implementación de la redefinición de los atributos Read y Write.

La serialización de esta estructura debe tener en cuenta que el campo data es una referencia polimórfica a objetos de una jerarquía de clases. Por tanto, `Hash_Table_Type'Write` debe realizar llamadas a `'Class'Output` en vez de utilizar `'Output` o `'Write`. Igualmente, `Hash_Table_Type'Read` debe realizar llamadas a `'Class'Input` en vez de utilizar `'Input` o `'Input`. Las figuras 7 y 8 ilustran este hecho.

El último paso consiste en hacer que los corre-

```
procedure Write(
  Stream : access Root_Stream_Type'Class;
  Table : in Hash_Table_Type) is
begin
  for N in 0..(Table'Length -1) loop
    pointer := Table(N);
    if pointer /= null then
      element := pointer.all;
      Integer'output(stream,N);
      Signal_Pointer_Type'Class'Output(
        stream, element.data);
      Identifier_Type'Output(
        stream,element.ident);
      while pointer.next /= null loop
        pointer := pointer.next;
        element := pointer.all;
        Integer'output(stream,N);
        Signal_Pointer_Type'Write(
          stream,element.data);
        Identifier_Type'Output(
          stream,element.ident);
      end loop;
    end if;
  end loop;
  Integer'Output(stream,-1);
end Read;
```

Fig. 7. Serialización de la tabla de señales.

spondientes programas principales realicen la serialización y la carga de los archivos. Esto una vez definida la infraestructura descrita es tan sencillo como realizar las correspondientes llamadas a `Output` y `Input`.

VI. Paralelización de la carga

Aunque la serialización como mecanismo de preprocesado permite obtener un incremento en el rendimiento de la carga de datos, éste proceso necesita obtener un rendimiento los más alto posible. Se puede obtener una mejora en el proceso de carga de datos si se paraleliza el proceso. Ada 95 es un entorno ideal para la paralelización de algoritmos ya que proporciona un conjunto de herramientas más que apropiadas para el desrrollo de programas concurrentes [2].

La carga de datos es un buen candidato para ser paralelizado, puesto que incluye un alto porcentaje de operaciones de entrada/salida. Si el proceso se paraleliza, los tiempos en los que el procesador no se encuentra ocupado, se pueden aprovechar por otras tareas. De esta forma, se pueden aprovechar bastante bien los tiempos de espera por operaciones de entrada salida.

Existen dos posibles alternativas para llevar a cabo la paralelización:

- Cada tarea en ejecución se encarga de cargar un subconjunto de los datos de disco en las estructuras de datos.

```
procedure Read(
  Stream : access Root_Stream_Type'Class;
  Table : out Hash_Table_Type) is
    identifier : Identifier_Type;
    signal : Signal_Pointer_Type;
    pointer : Pointer_Type;
    aux : Pointer_Type;
    N : INTEGER;
begin
  N := Integer'Input(stream);
  while N /= -1 loop
    index := new Element_Type;
    signal :=
      Signal_Pointer_Type'Class'Input(
      stream);
    identifier := Identifier_Type'Input(
      stream);
    pointer.data := data;
    pointer.identifier := identifier;
    pointer.next := null;
    if Table(N) /= null then
      aux := Table(N);
      while aux.next /= null loop
        aux := aux.Next;
      end loop;
      aux.next := pointer;
    else
      Table(N) := pointer;
    end if;
    N := Integer'Input(stream);
  end loop;
end Write;
```

Fig. 8. Carga de la tabla de señales serializada.

- Se utiliza un pool de procesos de forma que cada proceso tiene un ciclo de ejecución en el que realiza una lectura y la correspondiente acutalización en las estructuras de datos.

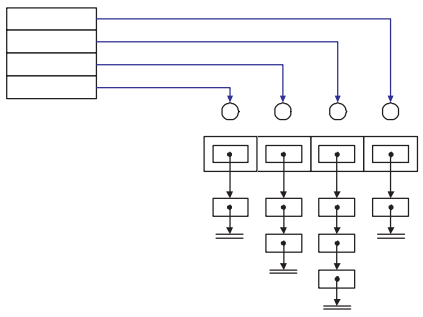

Fig. 9. Paralelización por particionamiento

De las dos alternativas, la primera tiene como ventaja su sencillez. Si se realiza un particionamiento adecuado de los datos, y se realiza una asignación adecuada de las tareas a las particiones de datos, se eliminan automáticamente los problemas de concurrencia. Esto evita tener que realizar tratamientos de exclusión.

En el caso de una estructura como la propuesta en anteriormente (una tabla hash), se puede realizar un particionamiento bastante sencillo, asignando una tarea a cada entrada de la tabla hash (que puede tener una lista enlazada para las colisiones). Es necesario que en el momento de la generación de los datos serializados se almacene información con las posiciones de comienzo de cada una de las particiones dentro del archivo serializado.

```
procedure Deserialize(
st : out Signal_Table) is
  task type EntryLoader(index : Integer);
  type EntryLoaderAccess is access
   EntryLoader;
  task body EntryLoader is
    df : File_Type;
  begin
    Open(df, in_file, "file.dat");
    Set_Index(f,offsets(index));
    st.all(index) :=
      Signal_Table_Node'Input(stream(f));
    Close(f);
  end EntryLoader;
  loaders : array (1..st'Last) of
   LoadEntryAccess;
begin
  LoadOffsets(offsets);
  for i in 1..st'Last loop
    loaders(i) := new EntryLoader(i);
  end loop;
end Deserialize;
```

Fig. 10. Paralelización de la carga.

La figura 9 muestra un esquema de la estrategia de paralelización donde cada tarea se ocupa de una partición del archivo y carga los datos correspondientes en la lista correspondiente a una entrada de la tabla hash.

En la figura 10 se muestra como puede paralelizarse la carga del archivo serializado.

Se han realizado pruebas para diferentes conjuntos de definiciones de señales (entre 10,000 y 50,000 señales). La tabla I muestra una comparación de los tiempos de ejecución expresados en segundos.

TABLA I
Comparación de tiempos de ejecución entre carga secuencial y carga paralela.

N. Señales.	Secuencial	Paralela
10,000	16.974	0.421
20,000	39.086	1.522
30,000	58.834	2.043
40,000	79.615	3.826
50,000	100.144	5.788

Para la realización de las pruebas se han utilizado señales con unas necesidades de almacenamiento de 1KB. Los resultados obtenidos muestran que el ahorro en tiempo de ejecución permite mejorar el tiempo necesario para cargar los datos en el banco de pruebas.

La figura 11 muestra la notable reducción conseguida con la versión paralela del algoritmo de carga de datos. Con esta mejora se ha conseguido el objetivo fundamental que se perseguía en la optimización de la carga de datos: hacer que el tiempo de carga no fuese un inconveniente para la repetición de las pruebas.

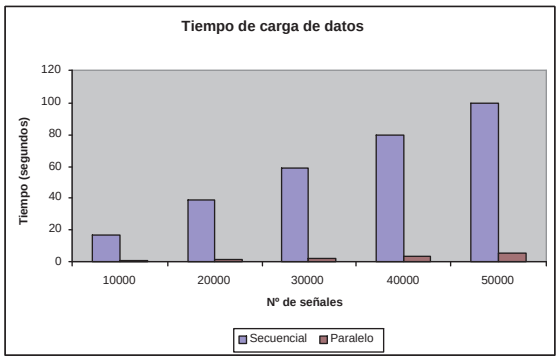

Fig. 11. Ahorro en el tiempo de carga de datos.

VII. Conclusiones

En este artículo se ha presentado un problema que puede darse en diversos sistemas a la hora de cargar grandes volúmenes de datos que se encuentran representados en XML. Aunque XML es un formato que ofrece indudables ventajas, el tiempo necesario para cargar un conjunto de datos XML en un sistema puede ser a veces prohibitivo.

La primera estrategia que se ha presentado consiste en el preprocesado de los datos para almacenarlos en un formato lo más próximo posible a la representación del sistema final. Esta estrategia permite un primer ahorro en el tiempo de carga. Al mismo tiempo que sienta las bases para que se puedan aplicar técnicas de paralelización en el proceso de carga.

La segunda estrategia es la paralelización del proceso de carga. Utilizando un conjunto de hilos de ejecución que leen simultáneamente del mismo archivo serializado, se puede reducir notablemente el tiempo necesario para cargar los datos en memoria.

VIII. Trabajo futuro

En este árticulo se ha explorado la paralelización del conjunto de datos mediante particionamiento que es una alternativa bastante simple. No obstante, existen otras estrategias de paralelización que deben ser exploradas para poder establecer comparaciones entre las distintas alternativas.

Otro aspecto que queda pendiente es el de la adecuada parametrización del algoritmo de carga paralela mediante particionamiento. La identificación de los parámetros que intervienen en el rendimiento y su correcto ajuste, puede dar lugar a

mejores versiones del algoritmo presentado.

Referencias

[1] Barnes, J. *Programming in Ada 95*, Addison-Wesley, 1995
[2] Burns, A. and Wellings, A. *Concurrency in Ada. Second Edition* Cambridge University Press, 1995
[3] J. Carretero, J. Fernndez, J.M. Prez, F. Garcia *Distributed Instrumentation Systems Using CORBA*, Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics, 14-18 July, 2002. Orlando, Florida, EEUU. Pp. 86-91. ISBN: 980-07-8150-1
[4] Ciancarini, P. and Rizzi, A. and Vitali, F. *An Extensible Rendering Engine for XML and HTML*, Computer Networks and ISDN Systems, Vol 30, no 1-7, pp. 225-237, ISSN: 0169-7552
[5] Kienzle, J. and Romanovsky, A. *On Persistent and Reliable Streaming in Ada* International Conference on Reliable Software Technologies - Ada-Europe'2000 June 26-30, 2000. Potsdam, Alemania. Pp. 82-95
[6] Winter, D.C. *Open Systems Ada technology demonstration program* Proceedings of ERA International Avionics Conference and Exhibition 19-20 Nov. 1997. Heathrow, Reino Unido. pp. 433-438. ISSN: 01419331