Un Algoritmo Genético Multi-Objetivo para la Optimización de Memoria Dinámica en Sistemas Empotrados

J.I. Hidalgo[1], D. Atienza[1], S. Belmar[2], C. M. González[2], P. Virseda[2], J. Lanchares[1], F. Fernández[3]

Resumen—**Los sistemas empotrados comerciales han aumentado sus capacidades hasta el punto de que, en la actualidad, pueden implementar nuevas aplicaciones multimedia que hasta hace pocos años estaban reservadas a potentes equipos de sobremesa. Estas aplicaciones tienen en común la necesidad de un complejo e intensivo uso de la memoria. Por ello la optimización de la memoria dinámica es un requerimiento básico al portar estas aplicaciones. Dentro de ella el ajuste de las estructuras dinámicas de datos (EDDs) es una de las partes más importantes a la hora de utilizar un sistema empotrado. En este artículo se presenta un método automático de optimización de EDDs de aplicaciones multimedia orientadas a objetos. En concreto se presenta una implementación de un algoritmo genético multi-objetivo basado en vectores (VEGA), en el que dada una aplicación para un sistema empotrado a optimizar, devuelva las mejores EDDs para las variables dinámicas de la misma. Para ello, nos hemos basado en los resultados obtenidos en los perfiles de ejecución de aplicaciones reales y de resultados teóricos sobre las implementaciones de dichas estructuras de datos. Con esta información se pueden obtener optimizaciones de las implementaciones de las variables usando EDDs orientadas a la reducción de energía, a la reducción en el consumo de memoria y al incremento del rendimiento. Los resultados obtenidos sobre aplicaciones reales muestran que se pueden obtener implementaciones óptimas de manera automática en sólo unos minutos, mientras que las aplicaciones hasta ahora presentadas necesitaban días para obtener una posible implementación.**

Palabras clave—**Memoria Dinámica, Sistemas Empotrados, Optimización, Algoritmos Genéticos Multiobjetivo, Energía, Rendimiento.**

I. INTRODUCCIÓN

Nos encontramos a las puertas de la cuarta generación de telefonía móvil. Esta revolución tecnológica integrará en un único dispositivo sumamente compacto, una gran cantidad de aplicaciones, que van desde las propias de un teléfono como tal, hasta otras que podrían encontrarse instaladas en un PC de sobremesa, como por ejemplo, sistemas de videoconferencia, reproductores de vídeo y sonido, etc. No sólo en los nuevos teléfonos móviles tenemos esta combinación de multimedia y alta integración, sino que las PDAs (Personal Digital Assistant) o las videoconsolas portátiles son otro claro exponente de sistemas empotrados multimedia de altas prestaciones.

Estos dispositivos, a pesar de que el avance tecnológico en cuanto a integración de circuitos integrados ha sido enorme y ha conseguido procesadores idóneos para el tipo de aplicaciones requeridas, siguen adoleciendo de ciertas limitaciones. Los mayores inconvenientes surgen del hecho de que las aplicaciones usadas en los sistemas empotrados suelen provenir de sistemas de sobremesa, con diferentes restricciones, entre las que destaca el uso de memoria, en particular la memoria dinámica. Como ejemplo podemos destacar que un computador de sobremesa dispone de, al menos, entre 512 y 1024 MB de memoria RAM, mientras que un sistema empotrado no suele incorporar más de 64 MB.

Es por ello que una de las principales tareas en el proceso de adaptación de aplicaciones informáticas para sistemas empotrados multimedia es la optimización del sistema de memoria dinámica, eligiendo adecuadamente las Estructuras de Datos Dinámicas (en adelante, EDDs) que mejor se adapten a las necesidades de la aplicación y las limitaciones de los componentes físicos de la plataforma empotrada objetivo.

Para optimizar el uso de memoria dinámica, el programador habitualmente dispondrá de una biblioteca de EDDs (arrays dinámicos, listas enlazadas, etc.) entre las que debe escoger dependiendo de sus necesidades o restricciones en cuanto a rendimiento, uso de memoria y consumo de energía. Normalmente, ésta tarea se realiza mediante la ejecución de una búsqueda exhaustiva para la obtención de un frente de Pareto (conjunto de puntos óptimos de alguna de la métricas que se desean optimizar, como el rendimiento, el consumo energético o el uso de memoria) que abarque todas las posibilidades sobre las que realizar la elección final. Está búsqueda es efectiva pero presenta el inconveniente de un alto coste computacional.

[1] Departamento de Arquitectura de Computadores y Automática. Universidad Complutense de Madrid, {hidalgo, datienza, julandan}@dacya.ucm.es

[2] Facultad de Informática, Universidad Complutense Madrid,

[3] Universidad de Extremadura, fcofdez@unex.es

Los algoritmos evolutivos han demostrado ser una herramienta eficiente para resolver problemas complejos de optimización en tiempos aceptables para distintos tipos de aplicaciones. El problema que tratamos en este trabajo es optimizar el uso de memoria, sin descuidar el rendimiento y el consumo de Energía. Estamos claramente ante un problema multi-objetivo.

El propósito de este trabajo ha sido la implementación de un algoritmo genético multi-objetivo (del tipo VEGA), en el que dada una aplicación para un sistema empotrado a optimizar, devuelva las mejores EDDs para las correspondientes variables dinámicas de dicha aplicación. Para ello, nos hemos basado en los resultados obtenidos en los perfiles de ejecución de aplicaciones reales y de resultados teóricos sobre las implementaciones de dichas estructuras de datos que proporcionan una manera fiable de medir el consumo, el rendimiento y la memoria.

Como podremos comprobar a lo largo del desarrollo de este artículo, los resultados obtenidos por nuestro trabajo demuestran la validez de nuestra aproximación.

El resto del artículo está organizado como se indica a continuación. En la sección II se exponen las características de los sistemas empotrados multimedia actuales que motivan la realización de este trabajo y el trabajo relacionado. En la sección III se expone la metodología completa de optimización en la que trabaja el algoritmo genético multi-objetivo que se presenta, y en la sección IV se detallan los pormenores del mismo. Los resultados experimentales se analizan en la sección V y las conclusiones y posibles líneas futuras de trabajo en la sección VI.

II. Trabajo Previo

Actualmente se considera ampliamente aceptado el hecho de que las aplicaciones multimedia, debido a su alto dinamismo (e.g. el número de objetos renderizados en pantalla puede variar significativamente según el momento de la aplicación), requerirán el uso de memoria dinámica en los nuevos sistemas empotrados de altas prestaciones. Por ello, una línea de investigación que ha cobrado una reciente importancia es la optimización de las estructuras dinámicas de almacenamiento (o DDTs) en sistemas empotrados [8].

En relación a dichas optimizaciones, los algoritmos de propósito general en entornos multimedia tienden a utilizar estructuras de datos que se basan en variantes del tipo abstracto de datos de las tablas ordenadas como estructura básica [18] cuando se intentan crear implementaciones con un buen rendimiento. Según esta idea, la biblioteca de patrones estándar de C++ (en inglés: Standard Template C++ Library o STL) [13], y otras bibliotecas de patrones propuestas recientemente [16][2] suministran una gran cantidad de implementaciones de estructuras de datos complejas para que los diseñadores puedan desarrollar nuevos algoritmos sin preocuparse de dichas implementaciones, con la única restricción de utilizar los métodos de acceso de las tablas para acceder a los datos. Sin embargo, dada su orientación a sistemas de sobremesa, este tipo de bibliotecas se centran en lograr implementaciones que proporcionen un buen rendimiento y no tienen en consideración otras ligaduras de diseño tan importantes como el rendimiento en los sistemas empotrados, tales como la disipación de potencia o el uso de memoria.

En el caso de sistemas empotrados multimedia de altas prestaciones se ha comenzado a estudiar cuáles son los métodos de acceso y las implementaciones de estructuras de datos más convenientes para lograr sistemas con un consumo bajo de energía [5]. Asimismo, varias transformaciones para estructuras de datos que disminuyen el grado de disipación de potencia y simplifican el control en bucles locales a distintas fases de las aplicaciones se han propuesto recientemente [10] [1] [20]. No obstante, este tipo de transformaciones sólo son aplicables a porciones de los códigos de entrada que utilizan estructuras de datos simples (e.g., vectores o vectores de punteros) [6], ya que no pueden identificar en tiempo de compilación los elementos accedidos en implementaciones complejas de estructuras de datos basadas en memoria asignadas dinámicamente, por lo que no son aplicables a las implementaciones existentes en las últimas generaciones de software multimedia.

Finalmente, se han propuesto metodologías de diseño [15] y optimizaciones de las implementaciones de las estructuras de datos para sistemas empotrados [5] que utilizan conocimiento específico del tipo de aplicaciones que se ejecutarán en el sistema final y/o que se basan también en estudios previos de los patrones de acceso a memoria física de los distintos tipos de entradas de datos al sistema (e.g., tamaño de los fotogramas que se deben mostrar en pantalla). Sin embargo, este tipo de técnicas no son aplicables tampoco a las estructuras dinámicas de datos debido a su impredecible comportamiento en tiempo de diseño.

III. Diagrama de Flujo de la Metodología de Optimización.

El entorno propuesto de optimización consta de tres fases, una fase de división y dos fases de optimización. Cada una de ellas está encargada de refinar de manera independiente el hardware empleado en un cierto componente del subsistema software de gestión de memoria dinámica del sistema dedicado (estructuras dinámicas de datos y gestor de memoria dinámica). No se considera el refinamiento de las estructuras de variables estáticas (analizables en tiempo de compilación) ya que para ello existen otras técnicas complementarias (véanse las referencias [6][20][10][15] para más detalles).

Debido a la compleja arquitectura hardware/software que poseen los sistemas dedicados de altas prestaciones, la optimización de todos los elementos que integran el subsistema de memoria dinámica o que actúan con él obliga a aplicar refinamientos a diferentes niveles de abstracción. De lo contrario, los diseños finales no serían óptimos a nivel global, si no solamente a nivel local. Los resultados obtenidos en varios sistemas comerciales tales como una aplicación de renderizado escalable tridimensional o una aplicación de reconstrucción tridimensional, demuestran que la aplicación completa de esta metodología es capaz de lograr mejoras en todas las métricas relevantes (rendimiento, uso de memoria, consumo de energía) de más de tres órdenes de magnitud con respecto al diseño comercial inicial del sistema. Asimismo, la metodología de refinamiento propuesta permite reducir significativamente el tiempo de optimización de dichas aplicaciones, logrando pasar de varios días como sucede actualmente a nivel comercial a apenas unos minutos.

A. Partición algorítmica.

La principal tarea a realizar al principio del flujo de diseño es la disgregación del código original de la aplicación en módulos de menor tamaño que se puedan analizar y optimizar individualmente de una manera eficiente según su carga final sobre el subsistema de memoria del sistema dedicado. Como ejemplo de la necesidad de esta fase, baste indicar que las aplicaciones de última generación para reconstrucción de imágenes en 3D o para el renderizado escalable de objetos complejos basado en triángulos, pueden llegar a ocupar más de medio millón de líneas de código en un lenguaje de alto nivel de la complejidad de C++ [5].

Esta fase de división contempla como entrada el código de la aplicación original a ejecutar por el sistema dedicado, usando un lenguaje de alto nivel de abstracción tipo C++. Este código, aunque todavía muy alejado de la implementación optimizada final, permite la validación funcional de su comportamiento algorítmico.

B. Exploración de las estructuras dinámicas de datos.

Tras la división del sistema completo en sub-algoritmos o tareas básicas, en esta fase se realiza el refinamiento del modo en que la memoria dinámica es reservada y accedida por parte de las aplicaciones o servicios multimedia de usuario existentes en el sistema final. Esta memoria dinámica es utilizada por estructuras creadas en tiempo de ejecución que, debido a su complejidad, deben ser definidas específicamente para cada aplicación si se desea lograr un resultado óptimo.

Se debe caracterizar el uso de memoria y patrón de accesos concretos a memoria dinámica ocasionado por la aplicación particular que se está optimizando. Para ello no basta con realizar un análisis estático del código C++ de la aplicación que se desea optimizar, sino que también es necesario estudiar el comportamiento en tiempo de ejecución de cada una de las variables dinámicas existentes en el sistema. La razón es que el primer estudio estático es suficiente para identificar cuáles son los tamaños de los objetos dinámicos que se van a almacenar en las estructuras de datos, pero no su cantidad real según cada caso de entrada. Por tanto, se debe realizar un estudio de comportamiento para cada entrada representativa del sistema, que haya sido definida y proporcionada por el diseñador de la aplicación original.

Con el fin de obtener el perfil de ejecución de una variable concreta de una manera coherente con los requisitos anteriormente mencionados, se ha diseñado una biblioteca orientada a objetos, implementada en código C++ y reutilizable para cualquier tipo de variable [4]. Esta biblioteca es fácilmente configurable mediante la modificación de una serie de directivas de preprocesado y, dado que no sobrecarga en exceso la aplicación original (entre el 3% y el 15%), es posible estudiar el comportamiento final del sistema con relativa precisión.

Tras la inserción de la biblioteca y ejecución del nuevo código, se obtiene un informe detallado del número de accesos totales, de lectura, peticiones de asignación de memoria, liberación, etc. Además, se obtiene información detallada para cada ámbito definido en el sistema y un informe dividido de la información global relacionada con cada variable.

C. Exploración automática de las estructuras dinámicas de datos.

Con la información obtenida mediante estos perfiles en ejecución, realizamos la ejecución de la aplicación de manera iterativa para el conjunto representativo de valores de entrada, definido por los diseñadores de cada aplicación concreta, utilizando cada una de las EDDs implementadas. Tras esto, el diseñador puede definir cuáles son las restricciones que desea respecto a las tres métricas usadas (energía, memoria y rendimiento), indicando unos valores concretos para el sistema dedicado que se está diseñando. De entre todas las soluciones obtenidas, el diseñador escogerá la que mejor se ajuste a los requerimientos de su diseño actual.

El principal problema de esta metodología es la gran cantidad de tiempo necesario para la exploración exhaustiva de las distintas EDDs [5]. Las soluciones propuestas hasta el momento necesitan un tiempo de exploración del orden de días, lo que la hace realizable para el ámbito de investigación académica, pero la excluye totalmente del mundo comercial en el caso de aplicaciones con gran número de EDDs, donde la fase final de optimización y filtrado de posibles soluciones la realizan habitualmente los diseñadores de manera manual según si propia experiencia.

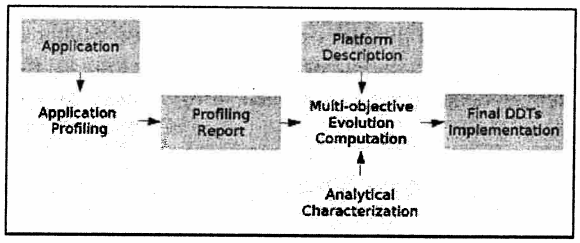

Fig. 1. Flujo de Diseño

La Figura 1 muestra un esquema de las distintas fases (en gris claro) y de las entradas y salidas de cada una (en gris oscuro). A continuación detallamos las fases básicas que se distinguen en la metodología propuesta.

El algoritmo presentado aquí se encarga de realizar la optimización de estas variables y es capaz reobtener los mismos resultados mediante computación evolutiva en un tiempo inferior (del orden de minutos). Los detalles de la implementación se presentan a continuación.

IV. Optimización multi-objetivo de las EDDs

El objetivo marcado es lograr la optimización del uso de memoria dinámica en sistemas empotrados multimedia usando algoritmos evolutivos multi-objetivo. Para ello, nos basamos en optimizar tres ligaduras de diseño de estos sistemas: consumo energético, rendimiento y uso de memoria.

La tarea consiste en seleccionar una determinada implementación de estructura de datos para cada una de las variables dinámicas del programa a optimizar. Para nuestra investigación, usamos una librería de implementaciones de éstas estructuras de datos que contiene 16 posibles implementaciones: Arrays dinámicos, listas enlazadas, listas doblemente enlazadas... (ver TABLA I). Cada una de estas implementaciones está representada por una fila de la tabla según los siguientes acrónimos:

- SLL(AR): Lista enlazada simple de vectores
- SLL(ARO): Lista enlazada simple de vectores de punteros
- DLL(AR):
- DLL(ARO) Lista doblemente enlazada de vectores de punteros
- SLL: Lista enlazada
- SLL(O): Lista enlazada por punteros
- DLL(O):Lista doblemente enlazada con clave explicita
- AR: Vector dinámico disperso
- AR (P): Vector dinámico de punteros

La ecuaciones mostradas en la tabla se obtienen de la caracterización analítica de las distintas EDDs. Con ellas se pueden calcular los valores que intervienen en las funciones de coste. Este análisis se basa en un número de parámetros que se pueden obtener en la fase inicial de profiling. N_e es el número de elementos almacenados en la EDD, s_w es la anchura de palabra en la arquitectura objetivo y s_t es el tamaño del elemento de tipo T y el nivel de agrupamiento de los elementos básicos. (FA).

TABLA I
ESTRUCTURAS DE DATOS ENTRE LAS QUE SE DEBE ELEGIR PARA CADA VARIABLE

DDT implem.	Sequential (NA_{s}) Access Count	Random (NA_{r}) Access Count	Average Size (S_{avr})
SLL(AR)	$3 \times N_r + N_a$	$\frac{N_e}{2 \times N_a} + 3$	$5s_w + \frac{N_e}{N_a}$
SLL(ARO)	$3 \times N_r + N_a$	$\frac{N_e}{2 \times N_a} + \frac{4 \times N_a}{N_r} + 1$	$7s_w + \frac{N_e}{N_a} \times (3s_w + s_T)$
DLL(AR)	$3 \times N_r + N_a$	$\frac{N_e}{4 \times N_a} + 3$	$6s_w + \frac{N_e}{N_a} \times (4s_w + s_T)$
DLL(ARO)	$3 \times N_e + N_a$	$\frac{N_e}{4 \times N_a} + \frac{N_a}{5 \times N_e} + \frac{5}{4}$	$8s_w + \frac{N_e}{N_a} \times (4s_w + s_T)$
SLL	$3 \times N_r$	$\frac{N_e}{2} + 1$	$4s_w + N_e(2s_w + s_T)$
SLL(O)	$3 \times N_e$	$\frac{N_e}{2} + \frac{1}{N_e}$	$6s_w + N_e(2s_w + s_T)$
DLL	$3 \times N_r$	$4N_r + 1$	$5s_w + N_e(3s_w + s_T)$
DLL(O)	$3 \times N_e$	$\frac{N_e}{4} + \frac{3}{N_e} + \frac{1}{4}$	$7s_w + N_e(3s_w + s_T)$
AR	N_a	1	$N_a \times s_T$
AR(P)	$2 \times N_a$	2	$N_a(s_T + s_w)$

A. Codificación Genética de Soluciones

Como es bien sabido, para aplicar correctamente un algoritmo genético es necesario una representación correcta de las soluciones, garantizando que todos los cromosomas sean codificaciones de soluciones reales al problema que se estás tratando. En nuestro caso debemos decidir la implantación más conveniente del conjunto de

variables de un programa. Por lo tanto, para definir cada una de las variables en el cromosoma debemos tener representación de la siguiente información que define la EDD a usar:

- Estructura de Datos (DS): Debemos elegir entre 16 diferentes posibilidades utilizando las EDDs previamente caracterizadas (TABLA I). Si usamos codificación binaria necesitaremos 4 bits para diferenciar las 16 posibilidades.

- Número de elementos (Elements). Hasta 8 elementos (3 bits).

- Número de Niveles de la EDD (Levels): Se pueden especificar hasta 4 niveles (2 bits).

- Campos Básicos (Basic Fields): Raramente habrá soluciones óptimas que tengan más de 7 u 8 campos en una estructura de datos. EN cualquier caso nuestra metodología permite en su codificación la representación de hasta 16 campos, por lo que necesitamos 4 bits para esta información.

En consecuencia, necesitaremos (4+3+2+4)=13 bits para representar la solución propuesta para cada variable, y si una aplicación tiene N variables, cada cromosoma necesita N*13 bits (genes) para representar una implementación. Por ejemplo, la primera aplicación que hemos probado en nuestros resultados experimentales (Simblob, ver sección V) utiliza únicamente dos variables dinámicas. Una solución potencial al problema vendría representada por un cromosoma de 26 bits (ver Fig. 2).

Nuestra implementación actual permite explorar aplicaciones con hasta 40 EDDs al mismo tiempo, lo que puede cubrir todas las aplicaciones reales para sistemas empotrados que nos interesan [5].

0 to 1	2 to 5	6 to 8	9 to 12	13 to 14	15 to 18	19 to 21	22 to 25	Bit positions
Levels	Basic Fields	Elements	DS	Levels	Basic Fields	Elements	DS	Meaning
Variable 1				Variable 2				

Fig. 2. Ejemplo de un cromosoma de 26-bits

D. *Funciones de Fitness*

Para poder evaluar los individuos de cada población es necesario evaluarlos con respecto a los tres objetivos de optimización (rendimiento, memoria y energía).

En primer lugar fragmentamos al individuo en las distintas variables. Para cada variable, calculamos su valor de Rendimiento, Memoria y Energía en la ejecución del programa (con la ayuda de la información obtenida previamente en el perfil en ejecución almacenada en la lista de resultados) y las ecuaciones siguientes:

$$Perf = (NA_r * (3 * (N_r + N_w - 2)/4)) + \\ + (NA_s * ((N_r + N_w - 2)/4)) + (NA_{cd} * 2) \quad (1)$$

$$AvMem = S_{av} \quad (2)$$

$$Energy = ((Npa * Epa) + (N_{rw} * Erw) + \\ + (S_{av} * Eestatica))/1000 \quad (3)$$

donde el Rendimiento viene dado por la ecuación (1). El resultado se mide en accesos a memoria. Donde lecturas (N_r) y escrituras (N_w) son datos obtenidos del profiling y representan el número de lecturas y escrituras de cada variable a optimizar.

NA_r es el número de instrucciones necesarias para realizar un acceso aleatorio, NA_s es el número de instrucciones necesarias para realizar un acceso secuencial, y NA_{cd} es el número de instrucciones para crear o destruir la estructura. Estos tres valores han sido obtenidos de forma teóricas para cada tipo de estructuras de datos.

El uso de memoria se obtiene de (2), el resultado se mide en Bytes y S_{avg} es el uso medio de memoria para una variable dependiendo del tipo de estructura dinámica usada para implementarla. Este dato también ha sido obtenido de forma teórica.

Finalmente el consumo energético se evalúa mediante la ecuación (3), el resultado se mide en nJ (nanoJulios). En esta ecuación N_{pa} es el número de fallos de caché, E_{pa} es el consumo de acceso a memoria principal, N_{rw} es el número de lecturas y escrituras realizadas sobre memoria caché, E_{rw} es el consumo de acceso a memoria caché, S_{av} es el tamaño en Bytes de la variable, y $E_{estatica}$ es el consumo por Byte de memoria.

Ante la imposibilidad de calcular los fallos de caché de manera experimental, realizamos una simplificación basándonos en datos estadísticos mediante los cuales consideramos el número de fallos de caché como el 2% del total de accesos realizados.

Una vez calculados los valores para cada variable presente en la solución, computamos los valores de todas las variables para todas las métricas del individuo, calculamos su valor de evaluación global, dependiendo del tipo de los objetivos a optimizar.

E. *Algoritmo Multi-objetivo*

La optimización Multi-Objetivo se puede definir como el problema de encontrar un vector de variables de decisión que satisfaga un conjunto de restricciones y optimice un vector de funciones

cuyos elementos representan las funciones objetivo. Estas funciones orientan la búsqueda pero suelen estar habitualmente en conflicto como así sucede en nuestro problema. Por lo tanto, el término optimizar significa encontrar una solución que dé valores de todas las funciones objetivo (energía, memoria y rendimiento) aceptables para el diseñador.

En los últimos años se han propuesto una gran cantidad de algoritmos evolutivos multi-objetivo[3][11]. De entre todos ellos una de las primeras propuestas fue la realizada por Schaffer. En [12] propuso una ampliación de un algoritmo genético simple (SGA) [7] denominada *Vector Evaluated Genetic Algorithm (VEGA)*.

La ejecución del algoritmo genético comienza de igual forma que el SGA, generando una población aleatoria. Para cada individuo de la población, se obtiene sus valores de evaluación según lo establecido en la sección anterior. Una vez que ya tenemos esa información para cada individuo, se aplican los operadores genéticos (selección, mutación y cruce) sobre los individuos de la población, con lo cual generamos la siguiente generación.

El Algoritmo Genético de tipo VEGA es básicamente un algoritmo genético simple en el que se cambia el operador de selección. En cada iteración del algoritmo, se realiza un filtrado de individuos dentro de su población mediante una función de selección multi-objetivo. Para ello, lo primero que realiza es la división de la población en tantas subpoblaciones como objetivos queramos optimizar. Después fusionamos los mejores individuos de estas subpoblaciones (Shuffling) y aplicamos los operadores de mutación y cruce (al estilo tradicional) para obtener la siguiente generación de la población.

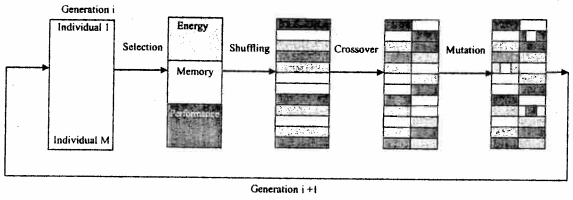

Fig. 3. Algoritmo VEGA

Esta población esta formada por los mejores de la generación anterior (30%), el resultado de mutar los mejores individuos de la anterior generación (20%), el resultado de cruzar los mejores de la generación anterior (40%) y con el objetivo de eliminar la especiación (principal problema del algoritmo genético VEGA), un 10% de individuos nuevos creados de forma aleatoria.

Mediante sucesivas iteraciones del algoritmo (VEGA), conseguimos la evolución de las poblaciones hacia los objetivos y con las restricciones propuestas por el usuario. Una vez que ya tenemos la última población (óptima) decodificamos los individuos y mostramos el resultado obtenido, es decir, las implementaciones óptimas de las estructuras dinámicas de datos (EDD´s) para las variables dinámicas extraídas del perfil en ejecución de la aplicación objetivo a optimizar.

El algoritmo se ha implementado con un interfaz que permite varias opciones. Es posible, por ejemplo, indicar los objetivos de optimización (uno, dos o tres objetivos), imponer restricciones de rendimiento, memoria o uso de energía. En cuanto a la memoria, es posible fijar el tamaño de la memoria caché y de la memoria principal de la arquitectura objetivo.

Además, en la ejecución del algoritmo genético se pueden cambiar algunos parámetros como el tamaño de la población, el número de iteraciones del algoritmo, y el número de variables a optimizar.

Este interfaz muestra también los resultados de la ejecución del algoritmo. Para cada objetivo de optimización se muestran todas las variables y su estructura de datos correspondiente obtenida del algoritmo. Además de ello se muestran los resultados de energía, memoria y consumo para cada objetivo de optimización.

V. Resultados Experimentales

Los resultados obtenidos han sido contrastados con los obtenidos al hacer la optimización de forma manual por los miembros del IMEC, llegando a conclusiones similares en unos tiempos muy inferiores. Las pruebas realizadas en sistemas comerciales tales como una aplicación de renderizado escalable tridimensional o una aplicación de reconstrucción tridimensional, demuestran que la aplicación completa de esta metodología es capaz de lograr mejoras en todas las métricas relevantes (rendimiento, uso de memoria, consumo de energía) de más de tres órdenes de magnitud con respecto al diseño comercial inicial del sistema.

Hemos aplicado la metodología de optimización implementada a tres programas comerciales. Las tres aplicaciones optimizadas han sido:

- Vdrift [17]: Un simulador de conducción en tres dimensiones.
- Simblob [14]: un simulador físico para fluidos sobre superficies sólidas.

- Lilith [19]: un motor de creación de mundos 3D con efectos climáticos, búsqueda de rutas, transiciones entre el día y la noche, etc.

Tras analizar y comparar los resultados con la optimización manual, se ha podido comprobar que gran parte de las variables (un 75-85% de media) poseen los mismos tipos de estructuras de datos que las obtenidas tras un proceso iterativo de optimización manual muy costoso en tiempo (entre 3 días y una semana, según el caso). Por ejemplo, en la aplicación VDrift, de las 43 variables que hay que optimizar 35 se obtienen con la misma estructura base (un 80%), en Simblob son correctas las 3 opciones óptimas de implementación para cada métrica estudiada (energía, rendimiento y uso de memoria) para la variable que dinámica que hay (100%), mientras que para Lilith, un caso con 80 variables dinámicas, los resultados se acercaban al 90%.

Los resultados son aún más convincentes si analizamos la figura. En ella se muestran los accesos a memoria para las distintas estructuras de datos. En ella se puede ver que la aproximación propuesta por nuestra optimización basada en un algoritmo VEGA, obtiene la mejor implementación posible aunque utilice otro conjunto de EDDs distinto que la propuesta optimizada manualmente.

La implementación del entorno gráfico añade funcionalidad y simplicidad a nuestro proyecto, ya que podemos cargar el perfil de la aplicación que se desea optimizar y elegir que objetivos se optimizan, restricciones para cualquiera de ellos, elegir la precisión del algoritmo genético, así como la cantidad de memoria de la arquitectura objetivo a optimizar. También permite almacenar los resultados de la optimización en un archivo de texto, para un posterior estudio de los resultados sin tener que volver a correr el programa.

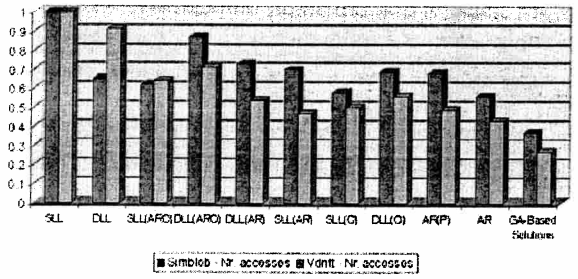

Fig. 4. Accesos a Memoria, para las distintas EDDs y para el conjunto obtenido por el algoritmo genético VEGA (GA-based solutions)

VI. Conclusiones y Trabajo Futuro

En este trabajo se han implementado un conjunto de algoritmos genéticos para la optimización de la memoria dinámica en sistemas empotrados multimedia de última generación. En concreto se han realizado versiones de un algoritmo genético, tanto simple, para un único objetivo, como multi-objetivo (VEGA), para dos o tres objetivos a optimizar, capaz de manejar restricciones en cualquiera de estos objetivos.

También hemos estudiado y analizado la arquitectura de los sistemas empotrados de última generación, para comprender qué estábamos realizando y el por qué de las limitaciones a las que se ven sometidos. Con esta información, y sabiendo que los tiempos necesarios para realizar optimizaciones en estos entornos de forma "manual" es del orden de días, y comprendiendo las reglas del mercado, justificamos la idoneidad de nuestro trabajo, con el cual podemos obtener los resultados en tiempos del orden de minutos.

Con respecto a otras soluciones anteriores, además se han realizado las siguientes mejoras:

- Proveer información más exacta sobre los accesos de lectura/escritura efectivos al algoritmo de optimización.
- Permitir incluir resultados sobre iteradores de variables [16], que en las aplicaciones diseñadas con objetos más recientes son muy habituales para acceder a los datos.
- Reconocer varios formatos de trazas.

Como trabajo futuro sería interesante explorar otras opciones de algoritmos genéticos multi-objetivo, ya que en este trabajo se ha presentado el más simple (VEGA) y sería interesante poder contrastar sus resultados (tiempo de generación de la curva de Pareto de soluciones y precisión en las estimaciones) implementando otros algoritmos.

Por otra parte, la arquitectura objetivo considerada tiene sólo un nivel de caché y otro de memoria principal. Para calcular los fallos de memoria caché, nos hemos basado en datos estadísticos fiables, aunque no del todo exactos. Las mejoras posibles en este caso serían poder elegir una jerarquía de memoria más avanzada, con varios niveles de caché, definiendo su asociatividad.

Una vez definida más en detalle esta arquitectura de memoria, sería positivo usar herramientas que simulasen la ejecución de los programas a optimizar y generasen los fallos de memoria caché en sus distintos niveles de forma más exacta.

Otra inclusión adicional en la jerarquía de memoria simulada podría ser el uso de memorias on-chip controladas por software (memorias Scratchpad [20]), que cada vez son más habituales en los sistemas empotrados debido a sus mejoras en

cuanto al consumo de energía y rendimiento con respecto a las memorias caché.

Actualmente, el sistema reporta las implementaciones óptimas de las variables dinámicas de una aplicación. Sería muy interesante, partiendo del perfil en ejecución y del propio código fuente, tras realizar la optimización y saber los resultados óptimos, automatizar estos cambios en el código fuente, sin que fuera necesaria la actuación manual.

AGRADECIMIENTOS

Para realizar este trabajo se han utilizado los archivos de trazas de ejecuciones de aplicaciones proporcionados por el IMEC (Leuven, Bélgica), mediante una colaboración con el estudiante de doctorado de dicho centro Christophe Poucet. Este trabajo ha sido subvencionado mediante el Proyecto CICYT TIN 2005-5619.

REFERENCIAS

[1] Nikolaos Bellas et al. Architectural and compiler techniques for energy reduction in high-performance microprocessors. IEEETrans. on VLSI, 2000.

[2] C++ Standardisation Comittee. Programming languages – C++ – ISO/IEC 14882. Tech. report, American National Standards Institutes, 1998.

[3] Carlos A. Coello et al. Evolutionary Algorithms for Solving Multi-Objective Problems. Kluwer Academic Publishers, 2002.

[4] C.Poucet et al., Template-Based Semi-Automatic Profiling of Multimedia Applications, Actas de IEEE International Conference on Multimedia and Expo (ICME), 2006",

[5] E.G. Daylight et al. Memory-access-aware data structure transformations for embedded software with dynamic data accesses. IEEETrans. on VLSI Systems, 2004.

[6] S. Debray et al. Compiler techniques for code compaction. ACMTOPLAS, 2000.

[7] Jhon Holland. Adaptation in Natural and Artificial Systems. Univ. of Mitchigan Press, 1975.

[8] Ahmed Jerraya et al. Multiprocessor Systems-on-Chips. Morgan Kaufmann, 2005.

[9] Zbigniew Michalewicz. Genetic Algorithms + data structures = Evolution Programs. Springer-Verlag, 1996.

[10] SMuchnick. Advanced compiler design&implementation. Morgan Kaufmann Publisher, 1997.

[11] Andrzej Osyczka. Multicriteria optimization for engineering design. In Design Optimization, Academic Press, 1985.

[12] J.David Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. Actas de ICGA, 1985.

[13] SGI. Standard template library, 2006. http://www.sgi.com/tech/stl/.

[14] Simblob http://sourceforge.net/projects/simblob

[15] Asim Smailagic et al. Benchmarking an interdisciplinary concurrent design methodology for electronic/mechanical systems. Actas de DAC, 1995.

[16] Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. LNCS,Vol. 2177, 2001.

[17] Vdrift. http://sourceforge.net/projects/vdrift

[18] Derick Wood. Data structures, algorithms, and performance. Addison-Wesley, 1993.

[19] Lilith. http://www.grinninglizard.com/lilith).

[20] M. Kandemir et al, A Compiler Based Approach for Dynamically Managing Scratch-Pad Memories in Embedded Systems, IEEE Transactions on Computer-Aided Design, 2004.