MULTIPATH:
Un sistema para la programación lógica

Tesis Doctoral

Septiembre 1996

Presentada por:

Jordi Tubella Murgadas

Dirigida por:

Antonio González Colás

Universitat Politècnica de Catalunya

Departament d'Arquitectura de Computadors

5

MODELO ARQUITECTÓNICO DE MULTIPATH: INTERPRETACIÓN ABSTRACTA

En el **Modelo Arquitectónico de Multipath** [124] se define la representación de los elementos que constituyen el estado del Modelo de Ejecución de Multipath y la realización de las operaciones que lo modifican. La definición del Modelo Arquitectónico de Multipath se realiza para cada una de las dos fases propuestas por el modelo de ejecución. Este capítulo se centra en la fase de **interpretación abstracta** de un programa Prolog (**MAM-AI**), mientras que la fase de interpretación real de un programa será descrita en el capítulo siguiente.

La interpretación abstracta es necesaria para obtener información del comportamiento de un programa que es utilizada en la posterior fase de interpretación real. La información a calcular en esta fase agrupa a la establecida según el modelo de ejecución más la información necesaria motivada por la definición del propio modelo arquitectónico. Los datos a calcular según la definición concreta del modelo arquitectónico dependen básicamente de la forma de representar los términos Prolog en Multipath así como de los vínculos de las variables existentes en un momento de la ejecución.

La primera sección de este capítulo está dedicada a definir la representación de los términos Prolog y de los vínculos de las variables, con el fin de enumerar posteriormente todos los objetivos a realizar en esta fase. En las siguientes secciones, se detalla la representación del estado de la ejecución y la realización de las operaciones que forman la interpretación abstracta. Por último, se describe la etapa de compilación del programa Prolog inicial junto con la información obtenida hacia el formato utilizado para su representación en la fase de interpretación real del programa.

5.1 Representación de los términos Prolog

El modelo arquitectónico de Multipath soporta cuatro tipos de datos distintos para los términos Prolog: *variables*, *constantes*, *estructuras* y *listas*. En el lenguaje Prolog no se admiten declaraciones de tipos de datos para los términos que se emplean. Por ello, la identificación del tipo de datos se debe realizar en tiempo de ejecución. Cada uno de los diferentes tipos se identifica, durante la interpretación real, por un campo en cada una de las referencias realizadas a términos Prolog. Este **campo** se denomina **tag**.

La descripción concreta de la representación de los términos Prolog y sus referencias se divide en los dos apartados siguientes. En primer lugar, se describen las constantes, estructuras y listas (figura 5.1). Posteriormente, se describe para las variables (figura 5.2). En estas figuras se nombran las distintas zonas de memoria que pueden contener términos Prolog en Multipath. La descripción completa de las zonas de memoria y registros utilizados en la fase de interpretación real de Multipath se realiza en el capítulo siguiente.

5.1.1 Constantes, estructuras y listas

El valor del campo tag en una referencia a constante es `tcon`. Por otra parte, existen dos tipos de términos constantes: *átomos* y *enteros*. Su identificación se realiza mediante otro **campo stag**, que contiene el valor `tatm` para los átomos y el valor `tnum` para los enteros. En caso de una referencia a átomo, el **campo cval** (resto de bits) contiene la dirección de la memoria STRINGS, donde están almacenados los códigos ASCII de los caracteres que forman un átomo. En caso de una referencia a entero, el campo cval contiene directamente su representación en complemento a dos.

Las estructuras en Prolog están formadas por un *functor* (nombre de la estructura) y un cierto número de elementos indicado mediante su *aridad*. Cada uno de estos elementos corresponde a cualquiera de los términos Prolog soportados. En Multipath, una referencia a una

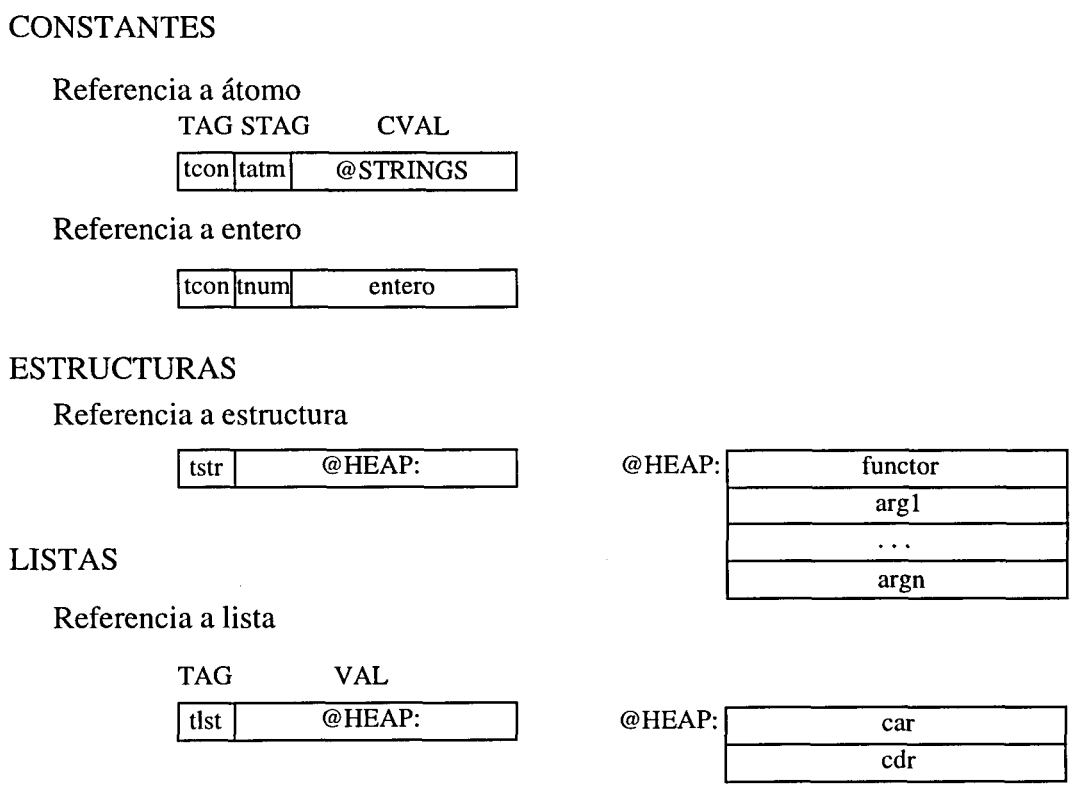


Figura 5.1: *Representación de constantes, estructuras y listas en Multipath.*

estructura contiene el valor `tstr` en el campo tag. El resto de la referencia está formado por el **campo val**. Este campo apunta a la memoria HEAP, donde están almacenados de forma secuencial el nombre de la estructura y su aridad, como una constante atómica, y las referencias a cada uno de los elementos que la forman.

Los términos estructurados que se utilizan en un programa se crean dinámicamente a medida que pueden ser referenciados durante su ejecución. Esta forma de representar las estructuras recibe el nombre de **structure-copying**, y fue definida por primera vez en [13]. En contraposición se encuentra la representación denominada **structure-sharing** [11]. En esta última forma de representar las estructuras se almacena una única vez y de forma estática el esqueleto de las estructuras que aparecen en el programa. El esqueleto identifica el functor, la aridad y los elementos de la estructura que no contienen variables. Durante la creación dinámica de estructuras sólo se almacenan los términos correspondientes a las variables que aparecen en el esqueleto.

Las listas se pueden considerar como un caso particular del tipo estructura. Una lista de

n elementos es equivalente a una estructura de nombre implícito "•" formada por dos elementos: el primer elemento de la estructura (**car**) corresponde al primer elemento de la lista y el segundo elemento de la estructura (**cdr**) corresponde a otra lista formada por los n-1 elementos restantes de la lista. Por ejemplo, la lista [a,b,c] es equivalente a la estructura •(a, •(b, •(c,[]))). El valor del campo tag para referenciar una lista es `tlst`. De forma análoga a las estructuras, el campo cval contiene la dirección de la memoria HEAP donde están almacenadas las referencias a los dos elementos (car y cdr) de la lista. El nombre de la estructura y su aridad no se almacenan, al ser implícitos. El caso particular de una lista vacía ([]) se representa como una constante atómica con un valor especial (NIL) para el puntero asociado al campo cval. Por otra parte, el lenguaje admite la utilización del operador "|" para identificar al segundo argumento de la estructura (cdr). Por ejemplo, la lista [a | b] es equivalente a la estructura •(a, b).

En el sistema PLM se utilizó otra forma de representar las listas denominada **cdr-coding** [36]. En esta codificación de las listas se debe comprobar el valor de un bit de la siguiente posición al car para saber si esa posición contiene una referencia al cdr de la lista o bien ya almacena el car del cdr. Esta forma de almacenar las listas ahorra espacio en memoria pero complica la realización de la unificación, y no ha sido utilizada posteriormente en ningún otro sistema.

5.1.2 Variables

Recordemos que una variable Prolog puede estar vinculada a otras variables pero únicamente puede llegar a estar vinculada a un único término no variable. En este caso, la variable está instanciada. En caso que la variable no esté vinculada a ningún término o a otras variables no instanciadas, la variable está libre. A nivel arquitectónico, una variable corresponde a una posición de memoria cuyo contenido es un único vínculo: sea una referencia a otra variable creada anteriormente, a un término no variable o bien una indicación que la variable no posee ningún vínculo. En caso que una variable esté vinculada a otra variable, se debe acceder a la variable vinculada para poder determinar si la variable inicial está libre o instanciada, operación que se denomina **derreferenciación.**

Una de las principales diferencias de la máquina abstracta propuesta en Multipath con respecto a la WAM estriba en la forma de representar las variables. Existen dos aspectos ortogonales a considerar en la representación de este término Prolog: cómo se accede a una variable desde el programa y cómo se almacena durante la ejecución el vínculo visible de una variable para cada camino recorrido. Cada uno de estos aspectos permite una clasificación de las variables que es descrita en los apartados siguientes.

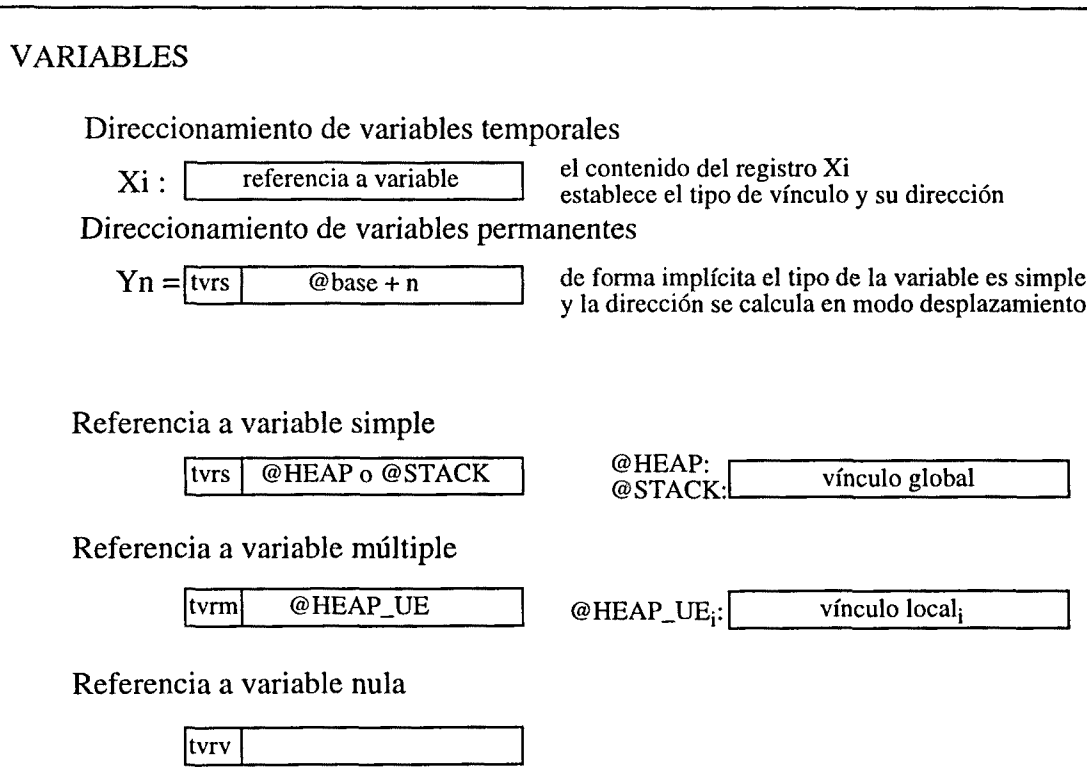


Figura 5.2: *Representación de las variables en Multipath.*

5.1.2.1 Direccionamiento de las variables

Todas las variables que se utilizan en un programa se crean dinámicamente y su ámbito de visibilidad está restringido a la cláusula en que aparecen. Es necesario poder establecer un mecanismo que permita referenciar una variable dentro de la cláusula con el fin de poder acceder a los vínculos obtenidos durante las unificaciones precedentes. Con este objetivo, las variables se clasifican de forma estática, según su utilización en una cláusula, en los siguientes tipos: **anónimas**, **temporales** y **permanentes**.

Las variables anónimas son aquellas que se utilizan una única vez a lo largo de toda la cláusula. En este caso, no es necesario conocer la dirección donde se haya creado ya que nunca serán referenciadas posteriormente en la cláusula.

Las variables temporales son aquellas variables que son utilizadas en más de un predicado de la cláusula (sea la cabecera o algún objetivo del cuerpo) que cumplen la condición que entre la primera y la última utilización no se modifican los registros Xi. Estos registros **Xi** son los

encargados de almacenar la referencia a la variable, es decir, a la posición de memoria donde está almacenado el vínculo de la variable. Ejemplos de variables temporales corresponden a aquellos casos en que una variable se utiliza en la cabecera y en el primer objetivo de la cláusula, o bien, cuando los objetivos existentes hasta su última utilización corresponden a predicados built-in (del tipo *is/2*), que no pueden realizarse mediante procedimientos Prolog.

Las variables permanentes engloban al resto de variables. Debido a que el contenido de un registro Xi puede perderse cuando se activa un objetivo (no se preserva el valor de los registros durante la resolución de un objetivo), la dirección donde está almacenado el vínculo de estas variables debe calcularse respecto a la dirección base de una estructura de datos que permanezca fija durante la ejecución de una cláusula. Esta estructura de datos se denomina entorno (env) y se crea al principio de la ejecución de una cláusula en la memoria STACK mediante la instrucción ALLOCATE. Cada variable permanente de una cláusula se referencia mediante la notación **Yn**, de forma que la dirección del vínculo de la variable se calcula sumando la constante *n* a la dirección base del entorno actual.

5.1.2.2 Almacenamiento de los vínculos de las variables

En el modelo de ejecución los vínculos de las variables (variable/término) están contenidos en el denominado *Entorno de Vínculos Global* (EVG) o en los *Entornos de Vínculos Locales* (EVLs), según el vínculo de la variable sea común a todos los caminos explorados simultáneamente o bien el vínculo sea local a cada uno de esos caminos. A nivel de modelo arquitectónico, las variables se clasifican de forma dinámica en dos tipos: **simples** y **múltiples**, según el número de vínculos que deben almacenar durante la ejecución del programa.

Una variable simple es aquella que durante toda su vida sólo debe almacenar un único vínculo, que es global a todos los caminos que se puedan explorar simultáneamente. Una referencia a variable simple contiene el valor `tvrs` en el campo tag. La referencia al vínculo de la variable está almacenado en la posición de la memoria HEAP o STACK que indique el campo val. Si la variable no posee ningún vínculo, el contenido de la posición de memoria es otra variable simple que apunta a la misma dirección.

Una variable múltiple es aquella que en algún momento de su vida almacena vínculos locales a cada uno de los caminos recorridos simultáneamente. Una referencia a variable múltiple contiene el valor `tvrm` en el campo tag. En este caso, los vínculos locales a cada camino están almacenados en las memorias HEAP_UE. Existe una memoria HEAP_UE asociada a cada camino. El vínculo local a cada camino está almacenado en la misma posición de memoria y viene indicado por el contenido del campo val existente en la referencia a la variable. Si una

variable múltiple no tiene asociado un vínculo local, el contenido de la posición de memoria es otra variable múltiple que apunta a la misma dirección.

Una variable temporal puede ser simple o múltiple según indique el campo tag del registro Xi asociado. Una variable permanente es simple ya que se accede siempre a la memoria STACK para buscar su vínculo global. Por otra parte, cualquier variable simple puede estar vinculada a otra variable múltiple pero nunca al revés: desde el momento que una variable está vinculada a otra múltiple todos los posibles vínculos posteriores a otras variables deben ser múltiples.

Adicionalmente, en la MAM se realiza una optimización consistente en considerar la existencia de otro tipo de variables, denominadas variables **nulas**. Una variable nula (su etiqueta es `tvrv`) puede ser simple o múltiple, pero tiene la característica que el vínculo o los posibles vínculos que se obtengan durante la ejecución no son utilizados posteriormente. La no utilización significa que el valor del vínculo no influye en determinar el flujo de control de la ejecución ni se referencia en predicados con efectos laterales (del tipo `write/1`). En este tipo de variables, el campo val no contiene información. El objetivo al introducir este tipo de variables consiste en reducir la cantidad de memoria necesaria para almacenar los vínculos de variables múltiples y el tiempo de ejecución necesario para el cálculo de dichos vínculos.

5.2 Objetivos de la interpretación abstracta

La interpretación abstracta realiza tres grandes tareas a nivel lógico: la transformación del programa Prolog en otro semánticamente equivalente con un menor número de operaciones a realizar, el cálculo de información necesaria en la interpretación real y la compilación al formato utilizado en dicha fase de interpretación real.

Dentro de estas tres grandes tareas se debe particularizar en las acciones concretas a realizar. A las acciones básicas de la interpretación abstracta definidas en el modelo de ejecución se deben añadir las acciones que son motivadas por la proposición de un modelo arquitectónico concreto. A continuación se recuerdan las acciones motivadas por el propio modelo de ejecución, ya descritas en el capítulo anterior:

- **Determinación de los puntos de entrada a procedimientos.**
- **Inserción de instrucciones de corte en el cuerpo de las cláusulas.**
- **Determinación de la función de indexación para cada punto de entrada.**
- **Determinación del atributo de indeterminismo de cada objetivo.**

El Modelo Arquitectónico de Multipath que se propone en este trabajo requiere que la fase de interpretación abstracta realice una acción adicional. Esta acción se engloba dentro de la tarea de añadir información al programa y consiste en la determinación del tipo dinámico de una variable:

- **Determinación del tipo dinámico para cada variable.**

A continuación se detallan las condiciones que permiten averiguar este tipo y el resultado que se espera obtener con esta acción.

5.2.1 Número de vínculos visibles de una variable

Existen dos causas que provocan la necesidad de que una variable sea múltiple:

(1) El hecho de vincular una variable a un término durante el cálculo de una solución de un objetivo explorado en anchura. Si esta misma variable es accedida posteriormente durante el cálculo de otra solución del objetivo o cuando todas las soluciones se han reunido en un único flujo de control, entonces la variable debe ser múltiple para permitir diferenciar el vínculo visible de la variable en cada uno de los caminos recorridos (ver figura 5.3a).

(2) El hecho de vincular una variable a un término que depende del vínculo que posee otra variable. Si esta última variable posee varios vínculos visibles, entonces la variable inicial debe tener también varios vínculos visibles. Esta causa de creación de variables múltiples puede suceder en cualquier momento de la exploración del

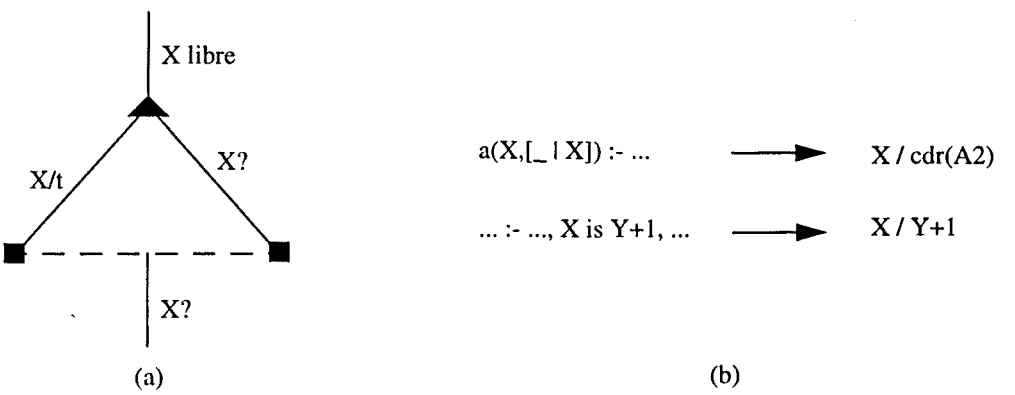


Figura 5.3: *Causas que motivan que el tipo dinámico de una variable sea MÚLTIPLE:*
(a) Acceso a una variable vinculada durante la exploración en anchura de un objetivo.
(b) Vinculación de una variable a un término que es función de otra variable múltiple.

árbol de búsqueda, tanto por la presencia de una variable en la cabecera que se unifica con un elemento de un término estructurado como por la semántica de algunos predicados predefinidos (ver figura 5.3b).

A nivel de modelo de ejecución se establece un criterio óptimo para determinar si el vínculo de una variable es global o bien debe ser local a cada camino (ver subapartados 4.2.2.3, 4.2.2.7 y 4.2.2.9). Este criterio óptimo de determinación del número mínimo de variables múltiples debe aplicarse dinámicamente durante la ejecución del programa. Sin embargo, su realización generaría una serie de inconvenientes a nivel del modelo arquitectónico: la penalización por la comprobación del posible cambio dinámico del tipo de una variable (simple a múltiple, o vicecersa) y la pérdida de orden entre la creación lógica de las variables y la obtención de la dirección que la identifica en su memoria correspondiente. Este último inconveniente provoca que la eliminación de las variables del *Entorno de Vínculos* no pueda realizarse sencillamente siguiendo el orden LIFO, tal como se realiza en la WAM, sino que deberían utilizarse técnicas de recolección de basura más complejas.

La primera decisión fundamental en el Modelo Arquitectónico de Multipath consiste en la **necesidad de conocer el tipo dinámico de una variable (simple, múltiple o nula) en el momento de la creación de la variable** con el fin de evitar los dos inconvenientes comentados en el párrafo anterior. Una vez que una variable ha sido creada no puede modificarse su tipo posteriormente. Esta decisión se favorece de la posibilidad que ofrece la etapa de interpretación abstracta del programa para encontrar situaciones en que se puede asegurar que el tipo de una variable va a ser simple durante toda su vida o bien de la posibilidad de establecer simples condiciones a evaluar en la creación de una variable que permiten asegurar que su tipo debe ser simple. Cuando no pueda asegurarse que el tipo de una variable es simple, se debe crear como múltiple.

En este sentido, el Modelo Arquitectónico de Multipath se basa completamente en el Modelo de Ejecución de Multipath, excepto en el criterio de determinación del número de vínculos de una variable. El criterio que se aplica consiste en obtener información referente al tipo dinámico de las variables durante la fase de interpretación abstracta y utilizar esta información para establecer el tipo definitivo de una variable en el momento de su creación durante la interpretación real. En concreto, la interpretación abstracta clasifica las variables en 4 tipos basándose en la siguiente heurística:

Variables NULAS: Corresponden a aquellas variables que cumplen todas las condiciones siguientes:

- La variable no es visible en ningún predicado con efectos laterales.
- La variable se vincula únicamente a términos básicos, es decir, que no contienen variables libres.
- Una vez la variable está instanciada no es visible en ningún objetivo posterior.

El cumplimiento de estas condiciones garantiza que la variable es nula: en ningún momento es necesario acceder a estas variables para determinar el vínculo de otras variables o para decidir el flujo de control de la ejecución. Las variables anónimas, desde el punto de vista de clasificación estática, son las principales candidatas para ser variables nulas.

Variables SIMPLES: Existen varias alternativas que permiten garantizar que una variable sea simple:

- Una variable aparece en la cabecera de la cláusula y se unifica con un argumento del objetivo que pretende resolver. En este caso, la variable debe almacenar un único vínculo que corresponde a una referencia al argumento, corresponda este argumento a una variable simple o múltiple.
- Una variable que aparece en la cabecera de una cláusula dentro de un término estructurado y el correspondiente argumento del objetivo a resolver ya está instanciado a dicho término. En este caso, el único vínculo de la variable será una referencia al correspondiente elemento del término estructurado que posea el objetivo.
- Una variable instanciada fuera del ámbito de un objetivo explorado en anchura, considerando que la pregunta inicial del programa se explora siempre en profundidad.

Variables S/M: Son variables que cumplen alguna o las dos siguientes condiciones:

- Se crean durante la activación de un objetivo que puede ser explorado en anchura y son vinculadas a términos básicos en todas sus soluciones.
- El número de vínculos de la variable depende únicamente del tipo dinámico de otra u otras variables que son accesibles en el momento de crearla.

En estos casos, la interpretación abstracta genera instrucciones adicionales para detectar en tiempo de ejecución si el objetivo es finalmente explorado en profundidad y/o si el vínculo de cada una de las variables de que depende es único. Si se cumplen estas condiciones en tiempo de ejecución, la variable se crea simple. En caso contrario, la variable debe crearse

múltiple.

Variables MÚLTIPLES: Corresponde al tipo dinámico por defecto que poseen todas las variables del programa y únicamente puede modificarse en caso que se cumplan las condiciones establecidas para los tres casos anteriores.

La misión que debe realizar la interpretación abstracta puede entenderse como la evaluación de una función TYPE para cada variable del programa cuyo dominio de salida especifica el tipo dinámico de la variable (VOID (nula), SINGLE (simple), S/M o MULTIPLE (múltiple)). En caso de ser tipo S/M, se añaden las condiciones que influyen en la determinación del tipo final de la variable durante su creación:

$$TYPE(var) = (VOID,\ SINGLE,\ S/M(traversal,\{var\},\ MULTIPLE)$$

La definición del cuerpo de esta función aplica las heurísticas comentadas anteriormente de clasificación de las variables. Las condiciones que incluyen pueden conocerse a través del resultado de las funciones FUNCT definidas en el capítulo anterior.

5.2.2 Estado de la ejecución y operaciones

En este apartado se resume de forma completa el estado de la ejecución (EJ_{IA}) y las operaciones (OP_{IA}) que deben realizarse en la fase de interpretación abstracta de un programa, teniendo en cuenta el Modelo de Ejecución de Multipath y las consecuencias que ocasiona la definición del Modelo Arquitectónico de Multipath en la fase de interpretación real.

$EJ_{IA} \equiv \{PRG,\ INFO,\ FUNCT\}$

$PRG = \{ep_1, \ldots, ep_i\}$

$EP = \{\ [cl_1, \ldots, cl_j] \mid \{arg,\ \{\{term_1, [cl_{11}, \ldots, cl_{1j}]\}, \ldots, \{term_n, [cl_{n1}, \ldots, cl_{nj}]\}\ \}\ \}$

$CL = [hd,\ gl_1, \ldots, gl_k]$

$HD = [arg_1, \ldots, arg_{hd}]$

$GL = \{ep_k,\ [arg_1, \ldots, arg_{gl}]\}$

$INFO = \{$

$ATTR(gl) = \{NONDET,\ SEMIDET,\ OTHER\}$

$TYPE(var) = \{VOID,\ SINGLE,\ S/M(traversal,\{var\},\ MULTIPLE\}$

$\}$

$FUNCT = \{$

$NSOL\_EP(ep):nsol,\ SUCC\_CL(cl):succ,$

IN_EP_ARG(ep,arg):dterm, IN_GL_ARG(gl,arg):dterm,
OUT_CL_VAR(cl,arg):dterm, IN_GL_VAR(gl,var):dterm,
OUT_CL_ARG(cl,arg):dterm, OUT_EP_ARG(ep,arg):dterm,
INEP_CL_ARG(cl,arg):dterm, INEP_GL_ARG(gl,arg):dterm
}

$$OP_{IA} \equiv \textit{Pre-proceso} \rightarrow \textit{Análisis Global} \rightarrow \textit{Post-proceso} \rightarrow \textit{Compilación}$$

En las secciones posteriores se describe la representación del estado de la ejecución (almacenamiento y codificación) y la realización de las operaciones según se propone en esta fase del Modelo Arquitectónico de Multipath, denominada MAM-AI.

5.3 Representación del estado de la ejecución

Los tres elementos del estado de la ejecución descritos a nivel de modelo de ejecución (PRG, INFO y FUNCT) se almacenan de forma conjunta en diferentes zonas de memoria. Cada una de estas zonas de memoria almacena estructuras de datos del mismo tipo. Dada una zona de memoria *mem*, la estructura de datos que contiene se simboliza como *tmem*.

Los dos siguientes apartados describen, en una visión horizontal, estas zonas de memoria y estructuras de datos. En los tres apartados que les siguen, se da un enfoque vertical de la representación del estado de la ejecución utilizando estos elementos arquitectónicos.

5.3.1 Zonas de memoria

Se utilizan 11 zonas de memoria (EP, CL, GL, VR, TP, RN, DS, RS, DT, RT y AL) durante la fase de interpretación abstracta.

Las **memorias EP, CL, GL y VR** contienen estructuras de datos tEP, tCL, tGL y tVR, respectivamente. Su funcionalidad es almacenar toda la información de un programa referente a puntos de entrada, cláusulas, objetivos y variables.

La **memoria TP** contiene la representación de los términos Prolog correspondientes a los argumentos de las cabeceras de las cláusulas y de los objetivos del programa. Respecto a esta zona de memoria es importante únicamente destacar que cualquier referencia a una variable se representa mediante una referencia a la estructura de datos tVR que almacena su información. Esta zona de memoria no es descrita de forma pormenorizada posteriormente.

El resto de zonas de memoria contienen la representación de las funciones pertenecientes

a FUNCT. A continuación se describe su funcionalidad según representen funciones que evalúan a nsol, succ o dterm.

La **memoria RN** almacena el resultado de evaluar la función NSOL_EP para todos los posibles valores de su dominio de entrada.

Las **memorias DS** y **RS** almacenan la definición del cuerpo y el resultado de la función SUCC_CL para todos los posibles valores de su dominio de entrada.

Las **memorias DT, RT** y **AL** contienen estructuras de datos que almacenan la definición del cuerpo, el resultado y los alias existentes, respectivamente, de todas las funciones que evalúan un elemento del dominio *dterm*. El concepto de alias hace referencia a aquellas variables libres que están vinculadas entre si antes de instanciarse a un término no básico. Una descripción más amplia de la necesidad de establecer los alias existentes en funciones relacionadas se realiza en el subapartado 5.3.5.1.

5.3.2 Estructuras de datos

Este apartado está dedicado a describir los diferentes campos de que constan las estructuras de datos que se almacenan en las zonas de memoria descritas anteriormente. La nomenclatura que se utiliza para enumerar los tipos de datos de los campos sigue el siguiente formato:

nat:	Contiene un elemento de tipo natural.
enum:	Contiene un elemento de un tipo enumerado. Los valores concretos del tipo se muestran en la descripción del campo.
set_N(*tipo*):	Contiene un conjunto de N elementos (no importa el orden de almacenamiento de los elementos) de tipo *tipo*.
$list_N$(*tipo*):	Contiene una lista de N elementos (importa el orden) de tipo *tipo*.
r*tipo*;	Contiene una referencia a un elemento de tipo *tipo*.

La **estructura de datos tEP** está formada por los campos:

NGL: nat	;	número de objetivos padre del punto de entrada.
NA: nat	;	número de argumentos del punto de entrada
GL: set_{NGL}(rtGL)	;	referencias a todos los objetivos padre.
IND: nat	;	indica el número de argumento a partir del cual realizar la indexación (en caso de no realizarse indexación el

valor del campo es NO_IND)

NT: nat ; en caso de indexación, contiene el número de términos distintos que deben comprobarse.

LTERM: $list_{NT}$(TP) ; en caso de indexación, contiene los términos que se deben comprobar en tiempo de ejecución.

NCL: $list_{NT}$(nat) ; especifica el número de cláusulas a probar para cada uno de los términos que se utilizan en la indexación. En caso de no realizarse indexación, la lista está formada por un sólo elemento.

LCL: $list_{NT}(list_{NCL}(rtCL))$; en caso de indexación, contiene la lista de cláusulas asociada a cada término; cuando no se realiza indexación, está formado por un único elemento que especifica la lista de cláusulas candidatas a unificar

NSOL: rtRN ; contiene una referencia al resultado de NSOL_EP.

IN_ARG: $list_{NA}$(rtRT) ; contiene referencias al resultado de la función IN_EP_ARG para cada uno de los argumentos.

IN_ALIAS: rtAL ; referencia a los alias existentes en los argumentos en el inicio de la ejecución del punto de entrada.

OUT_ARG: $list_{NA}$(rtRT) ; resultado de la función OUT_EP_ARG para cada uno de los argumentos de entrada.

OUT_ALIAS: rtAL ; referencia a los alias existentes en los argumentos después de ejecutar el punto de entrada.

La **estructura de datos tCL** está formada por los campos:

EP: rtEP ; referencia al punto de entrada de la cláusula.

NGL: nat ; número de objetivos en el cuerpo.

NVR: nat ; número de variables definidas en la cláusula.

ARG: $list_{NA(ep)}$(rtTP) ; lista de referencias a los términos Prolog correspondientes a los argumentos de la cabecera de la cláusula.

GL: $list_{NGL}$(rtGL) ; lista de referencias a los objetivos del cuerpo.

LVR: $list_{NVR}$(rtVR) ; lista de referencias a la información de las variables.

SUCC_F: rtDS ; referencia a la definición de la función SUCC_CL.

SUCC: rtRS ; referencia al resultado de la función SUCC_CL.

OUT_ARG_F: $list_{NA(ep)}$(rtDT); lista de referencias a las definiciones de las funcio-

nes OUT_CL_ARG, una para cada argumento.

OUT_ARG: $list_{NA(ep)}$(rtRT); lista de referencias al resultado de OUT_CL_ARG.

OUT_ALIAS: rtAL ; referencia a los alias existentes en los argumentos después de ejecutar la cláusula.

INEP_ARG: $list_{NA(ep)}$(rtRT); resultados de las funciones INEP_CL_ARG, uno para cada argumento.

La **estructura de datos tGL** está formada por los campos:

CL: rtCL ; referencia a la cláusula a la que pertenece el objetivo.

NA: nat ; número de argumentos.

ARG: $list_{NA}$(rtTP) ; lista de referencias a los términos Prolog correspondientes a los argumentos del objetivo.

IN_ARG_F: $list_{NA}$(rtDT) ; lista de referencias a las definiciones del cuerpo de las funciones IN_GL_ARG.

IN_ARG: $list_{NA}$(rtRT) ; lista de referencias al resultado de las funciones IN_GL_ARG.

IN_ALIAS: rtAL ; referencias a los alias existentes en los argumentos.

VAR_F: $list_{NVR(CL)}$(rtDT); lista de referencias a las definiciones del cuerpo de las funciones IN_GL_VAR.

VAR: $list_{NVR(CL)}$(rtRT) ; lista de referencias al resultado de las funciones IN_GL_VAR.

VAR_ALIAS: $list_{NVR(CL)}$(rtAL); lista de alias correspondiente a cada variable de la cláusula a que pertenece el objetivo.

INEP_ARG: list(rtRT) ; resultados de las funciones INEP_GL_ARG.

ATTR: enum ; atributo de indeterminismo del objetivo.

La **estructura de datos tVR** está formada por los campos:

CL: rtCL ; referencia a la cláusula en que es visible la variable.

T_STA: enum ; tipo estático de la variable.

DEP: rtRD ; dependencias de la variable.

OUT_F: rtFT ; referencia al cuerpo de la función OUT_CL_VAR.

OUT: rtFT ; referencia al resultado de la función OUT_CL_VAR.

T_DYN: enum ; tipo dinámico de la variable.

El resto de estructuras de datos son descritas en el apartado 5.3.5.

5.3.3 Representación de PRG

PRG se encuentra almacenado de forma distribuida entre las diferentes zonas de memoria que se han descrito anteriormente. Los puntos de entrada, las cláusulas y los objetivos de que consta un programa se acceden a partir de las estructuras de datos almacenadas en EP, CL, GL, VR y TP.

La información relacionada con un punto de entrada se accede mediante una referencia a una estructura de datos tEP. Esta estructura permite conocer el número de argumentos que tiene el punto de entrada, la función de indexación si es definida, y la lista de cláusulas alternativas para resolver los objetivos padre que tiene asociados.

La información relacionada con cada cláusula del programa se accede mediante una referencia a la estructura de datos tCL asociada. Esta estructura permite conocer cada uno de los términos Prolog de los argumentos de la cabecera y la lista de objetivos del cuerpo de la cláusula.

La información relacionada con cada objetivo de un programa se accede mediante una referencia a la estructura de datos tGL asociada. Dicha estructura permite conocer los términos Prolog utilizados como argumentos en el objetivo y la referencia a su punto de entrada.

5.3.4 Representación de INFO

INFO contiene una función ATTR para cada objetivo del programa y una función TYPE para cada variable del programa. El cuerpo de estas funciones se representa mediante código que será ejecutado durante las operaciones de la interpretación abstracta.

El resultado de la función ATTR se almacena en el **campo ATTR** de la estructura tGL asociada a cada objetivo.

El resultado de evaluar la función TYPE se almacena en el **campo T_DYN** de la estructura tVR asociada a cada variable en cuestión.

5.3.5 Representación de FUNCT

Obsérvese que las funciones FUNCT están distribuidas en distintas estructuras de datos según correspondan a información a calcular referente a un punto de entrada, a una cláusula, a un objetivo o a una variable. En primer lugar, es importante recordar que en cada una de estas funciones debe establecerse el dominio del resultado y la definición del cuerpo. Por ello, a nivel arquitectónico se divide la descripción de la representación de estas funciones en dos partes: representación del resultado y representación del cuerpo.

5.3.5.1 Representación del resultado

El resultado de las funciones FUNCT se representa mediante **estructuras de datos tRN, tRS** y **tRT**, según la función evalúe a un elemento del dominio nsol, succ o dterm, respectivamente.

Cada una de estas estructuras de datos almacena dos campos: el **campo RES** contiene el valor del resultado y el **campo STA** contiene el estado en que se encuentra su evaluación. Una función puede estar en 3 estados: (i) DEFINED, corresponde al estado inicial en que todas las funciones están pendientes de ser evaluadas; (ii) EVALUATING, la función está en proceso de evaluarse, y aún necesita el cálculo de otras funciones para su evaluación final; y (iii) EVALUATED, la función ya está calculada. En relación al resultado, a continuación se establece el dominio de salida que poseen los cuatro tipos de funciones.

El dominio de la función SUCC_CL (tipo *succ*) sigue el siguiente formato general:

succ = (sol, (op, i, dterm), ..., (op, i, dterm), side_effects)

cuyo significado se interpreta a partir del elemento *sol*, que puede tomar los valores YES, NO, O UNKNOWN.

- YES: Establece que la cláusula conduce a una solución. Para este valor se permite indicar la conjunción de condiciones que deben cumplir los argumentos de entrada para que la cláusula sea solución de un objetivo. Estas condiciones se establecen mediante un operador *op* (los operadores permitidos se definen en el siguiente subapartado) que, aplicado sobre el argumento *i*, debe evaluar a un dominio de términos concreto *dterm* (ver dominio dterm).

- NO: Asegura que la cláusula va a fracasar en resolver cualquier objetivo. Este valor permite la creación de nuevos puntos de entrada en los cuales se evita la ejecución de este tipo de cláusulas. No obstante, hay que tener en cuenta la existencia de predicados Prolog predefinidos que ocasionan efectos laterales. Por ello, únicamente puede eliminarse la cláusula cuando dicha cláusula está libre de efectos laterales, es decir, cuando *side_effects* está desactivado.

- UNKNOWN: Significa que no se puede asegurar si la cláusula va a tener éxito o va a fracasar en la resolución de un objetivo. Este valor se define para todas las funciones pertenecientes a FUNCT debido a que el nivel de detalle con que se realiza la interpretación abstracta puede no ser lo suficientemente elevado como para poder discernir un valor más exacto.

El dominio de la función NSOL_EP (tipo *nsol*) especifica el número de soluciones que se obtendrán en la satisfacción de cualquier objetivo padre de un punto de entrada. El formato general de los valores pertenecientes a este dominio sigue el mismo patrón que el dominio del tipo succ:

nsol = (num, (..., (op, i, dterm), ...), side_effects)

cuyo significado también se interpreta a partir del elemento *num*, que puede tomar los valores ZERO, ONE, MORE_THAN_ONE o UNKNOWN:

- ZERO: El punto de entrada no conduce a calcular ninguna solución.
- ONE: Únicamente una cláusula del punto de entrada puede tener éxito. Al igual que en el dominio de succ, se permiten indicar las condiciones que deben cumplir los argumentos de entrada para poder obtener dicha solución.
- MORE_THAN_ONE: El punto de entrada puede tener más de una solución. También se permite especificar las condiciones que aseguran que el punto de entrada es indeterminista para una determinada activación de un objetivo padre.

El dominio de las funciones que evalúan a *dterm*, es decir, a un subconjunto del universo de Herbrand que puede poseer un argumento o una variable en tiempo de ejecución real, viene simbolizado por:

dterm = (dt, nb)

donde el elemento *dt* tiene como objetivo identificar el tipo de datos y el elemento *nb* tiene como objetivo especificar si el término es el mismo para todos los caminos recorridos o son visibles términos diferentes en cada camino. Estos dos elementos especifican siempre el caso peor que puede presentar en tiempo de ejecución un argumento o una variable en todas las posibles activaciones de los puntos de entrada, cláusulas y objetivos involucrados. Debe observarse que la definición concreta de los valores que puede tomar el dominio de salida de esta función caracterizan el grado de refinamiento de la información que se obtiene mediante la interpretación abstracta.

Los valores concretos que pueden corresponder a *nb* son:

- SINGLE: En tiempo de ejecución únicamente va a ser visible un término.
- MULTIPLE: En cada uno de los caminos recorridos es visible un término distinto.

Los valores concretos que pueden corresponder a *dt* son:

- ALL: Corresponde al caso general en que el tipo de datos es cualquiera de los cuatro tipos soportados: constantes, estructuras, listas o variables.
- GND: Corresponde a un término que no posee variables libres (término básico).
- VAR: Corresponde a una variable libre.
- CON(VAL): Corresponde a una constante. En este tipo se añade el argumento VAL, que indica el valor que tiene la constante cuando todas las activaciones coinciden de valor o la marca UNKNOWN cuando puede variar el valor de la constante según la activación.
- STR(FUNCTOR,ELEM): Corresponde a una estructura. En este tipo se añade el valor del functor y una especificación del dominio de términos que engloba a todos los elementos de la estructura.
- LST(ELEM): Corresponde a una lista. Se añade una especificación del dominio de términos que poseen los elementos de la lista.

El tipo LST especifica el caso general de una lista que puede contener desde 0 elementos (lista vacía) hasta un cierto número variable de elementos. A su vez, el cdr de la lista puede corresponder al caso general en que corresponde a cualquier tipo de datos soportado. Siempre que sea posible se permite restringir la información de la siguiente manera:

- LST0: El término es una lista vacía.
- LST+: El término es una lista de 1 elemento como mínimo.
- LSTc: El término es una lista en la que todos los cdr corresponden a otra lista que referencia el resto de elementos.
- LST+c: Engloba los dos casos anteriores.

El dominio de términos con que se especifican los elementos de los términos estructurados (ELEM) también contiene una identificación del tipo de datos y del número de vínculos. El tipo de datos puede corresponder a:

- ALL: Caso general.
- GND: Los elementos son básicos.
- VAR, CON, STR, LST: Especifica un tipo de datos fijo para todos los elementos.
- SGND, LGND: Especifica una estructura o una lista con todos sus elementos básicos.

Debe observarse que el hecho de no poder asegurar que un término sea básico (contiene variables libres) dificulta en gran medida el grado de refinamiento en la información que se obtiene mediante la interpretación abstracta. Esto es debido a que cualquier variable libre en un término no básico puede estar vinculada a otras variables del programa y, en consecuencia, cualquier instanciación de estas variables debe ser visible también en el término original. Con el fin de aumentar el grado de refinamiento de la información se añade información referente a todos los **alias** (o elementos vinculados) entre un conjunto de términos relacionados. En este sentido, las funciones que determinan el dominio de términos asociados a todos los argumentos de un punto de entrada, cláusula u objetivo contienen una lista de los alias que aparecen. Esta información se obtiene a través de los campos denominados IN_ALIAS o OUT_ALIAS en las estructuras de datos tEP, tCL o tGL. El formato general de una lista de alias referentes a argumentos es el siguiente:

$$\textbf{alias} = \textbf{set}_{*}(\ (\textbf{op,i}),\ ...,\ (\textbf{op,j})\ )$$

donde cada elemento del conjunto especifica todos los términos vinculados a través de argumentos (i, ..., j). Cada operador *op* permite acceder al propio elemento (i) o, en caso de término compuesto, permite acceder al car(i), al cdr(i), o especificar un elemento cualquiera de un término compuesto elem(i).

La lista de alias también es necesaria en aquellas funciones que determinan el dominio de términos de las variables visibles en una cláusula. Esta información se obtiene mediante el campo VAR_ALIAS en la estructura tGL. En este caso, el formato es idéntico al de la lista de alias entre argumentos, donde los operadores *op* se aplican sobre variables visibles en la cláusula.

5.3.5.2 Representación del cuerpo

La representación concreta de cada una de estas funciones se basa en la distinción de las funciones según dependan o no dependan del código del programa fuente. La dependencia respecto al código fuente significa que el cuerpo de estas funciones se define según los argumentos concretos que tenga una cláusula en su cabecera o en sus objetivos.

El cuerpo de las funciones que no dependen del programa fuente se representan mediante código a ejecutar durante la operación de análisis global o de post-proceso. Puede considerarse el cuerpo de estas funciones como procedimientos cuya misión es la evaluación de la función correspondiente. Las funciones que se engloban en este tipo son: NSOL_EP, IN_EP_ARG y OUT_EP_ARG.

El cuerpo de las funciones que dependen del programa fuente se representa mediante una

estructura de datos que almacena el árbol asociado al cuerpo de la función. Entre las funciones de este tipo se encuentra la función que determina las condiciones de éxito de una cláusula (SUCC_CL), almacenadas en **estructuras de datos tDS**; y el resto de funciones FUNCT no mencionadas en el párrafo anterior, que calculan un subconjunto del universo de Herbrand (tipo *dterm*) asociado a un argumento o una variable, almacenadas en **estructuras de datos tDT.**

En el cuerpo de las funciones FUNCT pueden aparecer constantes (corresponden al dominio de salida de la función), operadores (permiten realizar transformaciones entre constantes del dominio) y otras funciones pertenecientes a FUNCT (según las dependencias que presente el programa). A continuación se detallan los **operadores** permitidos en estas funciones.

El cuerpo de la función NSOL_EP consta únicamente de un operador:

- `or(list(succ)):nsol`. El operador `or` recibe como argumentos las condiciones de éxito de las cláusulas que forman un punto de entrada y calcula el número de soluciones del punto de entrada.

En el cuerpo de las funciones SUCC_CL pueden aparecer los siguientes operadores:

- `unif(dterm,dterm):succ`. Establece las condiciones que deben cumplirse para que los dos argumentos del operador unifiquen.
- `and(succ,succ):succ`. Establece la conjunción de las dos condiciones de éxito indicadas por los dos argumentos del operador.

Los operadores que pueden intervenir en las funciones que evalúan sobre el dominio dterm son los siguientes:

- `unif(dterm,dterm):dterm`. Establece el dominio de términos resultante después de unificar los dos dominios de términos que indican los argumentos del operador.
- `union(list(dterm)):dterm`. Establece el dominio de términos que engloba a la lista de dominios de los argumentos.
- `intersection(list(dterm)):dterm`. Establece el dominio de términos común a toda la lista de dominios de los argumentos.
- `cdr(dterm):dterm`. Retorna el dominio de términos del cdr asociado a la lista que tiene como argumento.
- `car(dterm):dterm`. Retorna el dominio de términos del car asociado a la lista que tiene como argumento.

- `elem(dterm):dterm`. Retorna el dominio de términos de los elementos del término estructurado que tiene como argumento.
- `val(dterm,gl):dterm`. A partir de un dominio de términos, que puede corresponder a un término no básico, evalúa un nuevo dominio de términos aplicando todos los alias de las variables libres que aparecen, obtenidos hasta después de haber ejecutado el objetivo gl.

5.4 Realización del pre-proceso

El objetivo de esta operación es inicializar los elementos que forman el estado de la ejecución.

PRG contiene inicialmente tantos puntos de entrada como procedimientos tiene el programa Prolog fuente. En cada punto de entrada no se realiza indexación y la lista de cláusulas candidatas sigue el orden textual de aparición de cada cláusula del procedimiento.

Las funciones pertenecientes a INFO se inicializan a UNKNOWN o al valor que proporcione el programador, si ha introducido las directivas pertinentes.

Se establece la definición de las funciones FUNCT que dependen del programa Prolog y se inicializa su resultado a DEFINED. En concreto, se reservan las estructuras de datos necesarias en cada una de las memorias e inicializa su contenido según se ha establecido en la definición del cuerpo de las funciones FUNCT. La generación del grafo de cada una de las funciones se realiza mediante un análisis local de las cláusulas que forman el programa.

En el apéndice B se muestra un ejemplo con la definición de funciones FUNCT asociadas a los puntos de entrada, cláusulas y objetivos de un determinado procedimiento.

5.5 Realización del análisis global

El objetivo de esta operación es la evaluación de las funciones FUNCT.

La evaluación de una función FUNCT para unos parámetros de entrada concretos no se realiza hasta que todas las funciones de que depende, conocidas a partir de la definición del cuerpo de la función, están evaluadas. Por otra parte, la evaluación de los operadores que pueden aparecer en la definición de una función se realiza mediante tablas que especifican el valor de su resultado para todos los posibles valores que pueden poseer los operandos.

En consecuencia, la evaluación de todas las funciones FUNCT se realiza a partir de la necesidad de evaluar inicialmente el número de soluciones (función NSOL_EP) de todos los

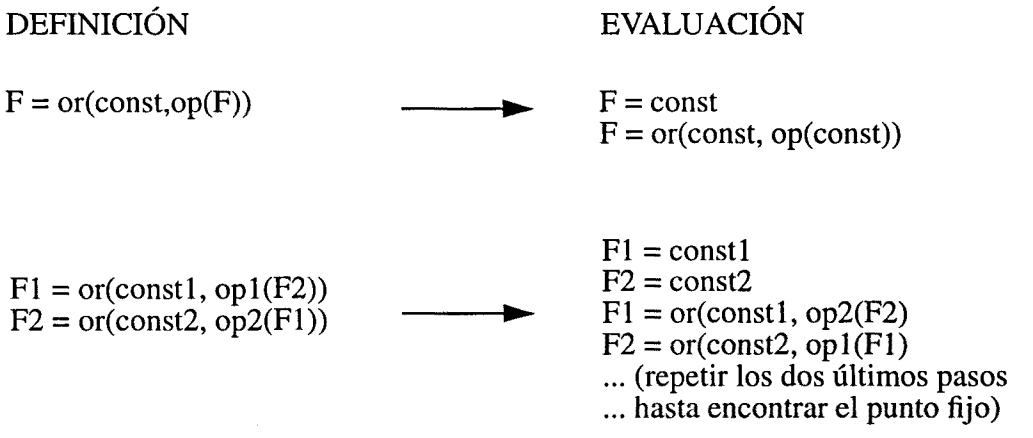


Figura 5.4: *Evaluación de funciones FUNCT con dependencias recursivas.*

objetivos que forman la pregunta del programa. La evaluación de estas funciones implica la evaluación de todas las funciones que dependen de ellas y para todos los posibles parámetros de entrada. Las funciones que no necesitan evaluarse para obtener el resultado de NSOL_EP son INEP_GL_ARG y INEP_CL_ARG. Estas funciones son evaluadas posteriormente.

Un punto importante en la evaluación de las funciones es la detección de posibles ciclos debido a la presencia de recursividad en el programa. Cuando se detectan ciclos cerrados en la definición de una o más funciones, debe realizarse la evaluación de dichas funciones. Para ello, en primer lugar se aplican los valores terminales para la función y, posteriormente, se aplica el operador que los modifica según la recursividad que presente la función. La figura 5.4 muestra dos ejemplos con recursividad en la definición de varias funciones. En el primer caso una función F depende únicamente de si misma, mientras que en el segundo caso dos funciones F1 y F2 dependen entre si. En ambos casos se detalla la secuencia de pasos para realizar su evaluación.

Por último, la evaluación de operadores unif en funciones que evalúan dterm ocasiona como efecto lateral la inclusión de los alias que se generan en la estructura de datos asociada a estas funciones. Se genera un alias siempre que se unifica dos términos que corresponden a variables libres. La función SUCC_CL también puede generar efectos laterales una vez evaluada. Este efecto lateral permite la creación de nuevos puntos de entrada siempre que la función evalúa a NO, es decir, la cláusula no soluciona ninguna activación de un objetivo padre.

En el apéndice B se muestra un ejemplo del resultado de funciones FUNCT asociadas a los puntos de entrada, cláusulas y objetivos de un determinado procedimiento.

5.6 Realización del post-proceso

El objetivo en realizar esta operación consiste en realizar el resto de transformaciones en el programa y calcular la información adicional que debe proporcionarse a la siguiente fase de interpretación real. Para ello, se utiliza el resultado de las funciones evaluadas anteriormente. El orden en que se realizan las acciones durante este post-proceso es el siguiente: inserción de operadores de poda, inserción de funciones de indexación, cálculo del atributo de indeterminismo de los objetivos y cálculo del tipo dinámico de las variables.

La inserción de *operadores de poda* tras la cabecera o un cierto objetivo de una cláusula se realiza siempre que el resultado de las funciones INEP_CL_ARG y INEP_GL_ARG sea incompatible con la función SUCC_CL evaluada para el resto de las cláusulas del punto de entrada. Obsérvese que no es necesario definir ni evaluar estas funciones en la última cláusula de todos los puntos de entrada.

Posteriormente, se determina la conveniencia de insertar una *función de indexación* en cada punto de entrada. Para ello, se utiliza el dominio de términos que poseen los argumentos de entrada (funciones IN_EP_ARG) y las condiciones de éxito de las cláusulas (funciones SUCC_CL). Para poder establecer la función de indexación, deben existir argumentos de entrada, cuyo dominio de términos asegure la eliminación de un cierto número de cláusulas del punto de entrada. Se escoge el argumento que elimina un número promedio de cláusulas mayor. Este número promedio se obtiene calculando las cláusulas que se eliminan para cada posible término del dominio del argumento.

A continuación, se evalúa el *atributo de indeterminismo* de cada objetivo. Este atributo se calcula a partir de la función NSOL_EP asociada al punto de entrada del objetivo. Existe una correspondencia directa entre el resultado de esta función y el atributo. Siempre que la función NSOL_EP evalúa a MORE_THAN_ONE, el atributo vale NONDET. El atributo es SEMIDET cuando se cumplen las siguientes condiciones: la función NSOL_EP evalúa a ONE, existe más de una cláusula alternativa por probar, y las condiciones de éxito de las cláusulas dependen de argumentos con posibilidad de tener términos distintos en los caminos recorridos (campo nb del dominio de términos igual a MULTIPLE). En cualquier otro caso, el atributo es OTHER. Durante esta fase del post-proceso, se afina incrementalmente el valor del campo nb de los dominios de términos por el hecho de establecer que nuevos objetivos pueden ser explorados en anchura. En concreto, desde el momento en que se vincula una variable a un término dentro del ámbito de un objetivo al que se recomienda una exploración en anchura (su atributo es igual a NONDET o SEMIDET), el campo nb de los dominios de términos posteriores en que la variable es visible

pasa a ser MULTIPLE. Este campo también pasa a ser MULTIPLE siempre que se vincula una variable a un término que depende de otro con múltiples vínculos (p. ej., en X/car(Y); X/cdr(Y) y X is Y+1, cuando Y tiene un dominio de términos MULTIPLE).

En último lugar, se evalúa el *tipo dinámico de las variables* del programa. El algoritmo seguido en esta acción sigue el siguiente orden:

- Se localizan aquellas variables que son consideradas NULAS o SIMPLES siguiendo el criterio establecido en el apartado 5.2.1. En estos casos, el tipo de exploración de los objetivos no influye.
- Las variables vinculadas durante la exploración de un objetivo en anchura que se crean en el momento de activar el objetivo poseen el tipo S/M(TRAVERSAL).
- Las variables que se han vinculado a términos que dependen de otras variables múltiples pasan a ser de tipo S/M y se añaden en las condiciones de este tipo las referencias a las variables de que dependen.
- El resto de variables poseen el tipo MULTIPLE.

5.7 Realización de la compilación

La última fase de la interpretación abstracta consiste en la traducción del programa Prolog transformado (PRG) y la información adicional obtenida (INFO):

$$
\begin{aligned}
PRG &= \{ep_1, \ldots, ep_i\} \\
&\quad EP = \{\ [cl_1,\ldots, cl_j] \mid \{arg, \{\{term_1,[cl_{11},\ldots, cl_{1j}]\}, \ldots, \{term_n,[cl_{n1},\ldots, cl_{nj}]\}\ \}\ \} \\
&\quad CL = [hd, gl_1, \ldots, gl_k] \\
&\quad HD = [arg_1, \ldots, arg_{hg}] \\
&\quad GL = \{ep_k, [arg_1, \ldots, arg_{gl}]\} \\
INFO &= \{ \\
&\quad ATTR(gl) = \{NONDET, SEMIDET, OTHER\} \\
&\quad TYPE(var) = \{VOID, SINGLE, S/M(traversal,\{var\}), MULTIPLE\} \\
&\quad \}
\end{aligned}
$$

en el formato adecuado para su posterior interpretación real. Este formato se denomina MAM (Máquina Abstracta de Multipath) y está constituido por un conjunto de instrucciones, directamente interpretables, cuya semántica permite realizar las operaciones del modelo de ejecución de la interpretación real. Las instrucciones MAM se dividen en instrucciones de control (en la tabla 5.1) e instrucciones de unificación (en la tabla 5.2). Los siguientes apartados están dedi-

cados a identificar los elementos considerados hasta ahora durante la interpretación abstracta (PRG e INFO) con sus correspondientes instrucciones MAM. Esta descripción se realiza para un punto de entrada del programa. Téngase en cuenta que un programa está formado por un conjunto de puntos de entrada.

5.7.1 Indexación

Un punto de entrada contiene opcionalmente una función de indexación que, aplicada sobre uno de los argumentos del objetivo padre, indica la lista de cláusulas susceptibles de poder unificar.

La función de indexación se especifica con las instrucciones englobadas en el tipo INDEXACIÓN. El formato general de un punto de entrada es el siguiente:

CÓDIGO DE OPERACIÓN	ARGUMENTOS
INDEXACION	
switch-on-term	Xi, Lv/F, Lc/F, Ll/F, Ls/F
switch-on-constant	NumTc, LTc
switch-on-structure	NumTs, LTs
SELECCIÓN DE CLÁUSULAS	
try-me-else	Lr
retry-me-else	Lr
trust-me	
try	Lcl
retry	Lcl
trust	Lcl
ACTIVACIÓN DE OBJETIVOS	
call	Lep, N
call-nondet	Lep, N
call-semidet	Lep, N
exec	Lep
exec-nondet	Lep
exec-semidet	Lep
allocate	
deallocate	
proceed	
escape	Built-in
exit	

Tabla 5.1: *Instrucciones de CONTROL*

Lep: SWITCH-ON-TERM (Xi, Lvar/FAIL, Lcnt/FAIL, Llst/FAIL, Lstr/FAIL)
Lvar: lista de cláusulas
Llst: lista de cláusulas
Lcnt: SWITCH-ON-CONSTANT (NumTc, LTc)
Ltc: . . .
{ constante$_i$, Lcon$_i$ }
. . .
Lcon$_i$: lista de cláusulas
Lstr: SWITCH-ON-STRUCTURE (NumTs, LTs)
Lts: . . .
{ functor$_i$, Lstr$_i$ }
. . .
Lstr$_i$: lista de cláusulas

donde la primera instrucción especifica el argumento sobre el que se realiza la indexación y las etiquetas de las listas de cláusulas a tratar según el tipo de datos que posea este argumento. Los tipos de datos que se consideran son variables, constantes, listas y estructuras. También se permite especificar la posibilidad de fracaso para un tipo de datos en concreto. Si el tipo de datos es constante o estructura, existen dos instrucciones adicionales que permiten especificar la lista de cláusulas para cada constante o nombre de estructura (functor) específico en que se puede diferenciar un número diferente de cláusulas a unificar.

Es conveniente recordar que la indexación es una optimización que también se aplica en la WAM, pero que no se introdujo en la descripción hecha en el capítulo 2 para facilitar su resumen y centrarse únicamente en los aspectos específicos de realización del modelo de ejecución convencional de Prolog. La diferencia existente en la indexación realizada en la MAM respecto a la WAM radica en la posibilidad de aplicarla sobre aquel argumento que permite eliminar un mayor número de cláusulas. En la WAM, la indexación se aplica implícitamente sobre el primer argumento.

5.7.2 Lista de cláusulas

La lista de cláusulas (referenciada por la etiqueta de salida de la indexación o por la etiqueta del punto de entrada en caso de no realizar la indexación) se identifica mediante instrucciones del tipo SELECCIÓN DE CLÁUSULAS. El formato general para representar una lista de cláusulas admite dos posibilidades:

Label: TRY-ME-ELSE (Lr1)
Lcl$_1$: cláusula 1

```
            . . .
Lrj:        RETRY-ME-ELSE (Lri)
Lcl_i:      cláusula i
            . . .
            TRUST-ME
Lcl_n:      cláusula n
```

o bien

```
Label       TRY (Lcl_1)
            . . .
            RETRY (Lcl_i)
            . . .
            TRUSTn (Lcl_n)
. . .
Lcl_i:      cláusula i
. . .
```

donde la primera cláusula candidata a unificar se especifica con la instrucción TRY-ME-ELSE o TRY. La diferencia entre ambas instrucciones consiste en identificar la situación en que el código de la cláusula esté almacenado de forma contigua a la instrucción (con TRY-ME-ELSE) o en cualquier otra dirección de la memoria (con TRY). Esta última opción permite compartir el código de una cláusula en más de un punto de entrada. La última cláusula se especifica con la instrucción TRUST-ME o TRUST y las restantes cláusulas con instrucciones RETRY-ME-ELSE o RETRY.

5.7.3 Cláusula

Cada cláusula se identifica por una etiqueta (Lcl_i), y está formada por instrucciones que representan la cabecera y el cuerpo de la cláusula. Adicionalmente, se introducen instrucciones ALLOCATE, DEALLOCATE y PROCEED. La estructura general de una cláusula es:

```
Lcl_i:      ALLOCATE
            cabecera de la cláusula
            objetivo 1 del cuerpo de la cláusula
            . . .
            objetivo j-1 del cuerpo de la cláusula
            DEALLOCATE
            objetivo j del cuerpo de la cláusula
```

Las instrucciones ALLOCATE y DEALLOCATE sirven para reservar y eliminar el espacio

de memoria (denominado entorno) necesario para almacenar las variables permanentes. Por ello, estas instrucciones sólo deben usarse cuando la cláusula contiene variables permanentes. Por otra parte, en la MAM, al igual que en la WAM, también se realiza la optimización de la última llamada (LCO) consistente en eliminar el entorno de una cláusula antes de activar el último objetivo de su cuerpo. El objetivo de esta optimización es reducir el espacio de memoria necesario para los procedimientos con recursividad de cola, de forma que se requiere un espacio similar al que se utilizaría en un procedimiento iterativo.

Sin embargo, esta optimización introduce un posible conflicto en aquellas variables permanentes que son libres y que se almacenan en un entorno que es eliminado por causa de esta optimización. En todas los accesos a estas variables o a variables que tengan como vínculo a estas variables se produciría una referencia a una posición de memoria inexistente. Este tipo de variables permanentes se denominan inseguras. Existen dos situaciones que pueden provocar el acceso a una variable insegura:

- La referencia a una variable permanente en un argumento del último objetivo de la cláusula.
- La referencia, en un elemento de un término estructurado, a una variable que puede estar vinculada a un término creado antes de ejecutar la cláusula.

Las referencias a estas variables se realiza con instrucciones especiales, que serán descritas más adelante.

Cuando la cláusula contiene únicamente una cabecera, la estructura del código MAM es:

Lcl_i: **cabecera de la cláusula**
PROCEED

5.7.4 Cabecera de una cláusula

La especificación de los argumentos de la cabecera de una cláusula se realiza mediante instrucciones del tipo UNF. En la tabla 5.2 se enumeran todas las instrucciones de este tipo. Existen distintas instrucciones UNF para representar los posibles términos de los argumentos que pueden aparecer en la cabecera. A su vez, existen instrucciones diferentes para representar un mismo término con el objetivo de especificar la información detectada durante el análisis global del programa acerca del tipo de datos de los argumentos de los objetivos padre.

La información que se ha obtenido mediante la interpretación abstracta y que va a ser usada en la generación de código MAM es:

CÓDIGO DE OPERACIÓN	ARGUMENTOS
UNF	
unf_i_{ix/y}	Xi, {Xij/Yn}
unf_i_create-var-{x/y}	Xi, {Xj/Yn}
unf_i[/-tv]_con	Xi, Cnt
unf_ix[/-tnv/-tl/-tv]_ref-lst	Xij
unf_ix[/-tnv/-tv]_ref-str-ftr	Xij, Ftr
UNF_ARG: modo READ o WRITE	
unf/create-arg_{ix/y/uns-ix/uns-y}	{Xij/Yn/Xij/Yn}
unf/create-arg_create-vr[s/m/x/v]-{x/y/a}	{Xj/Yn/k}
unf/create-arg_cnt	Cnt
UNF_ARG: modo READ	
unf-arg_{ix/y/uns-ix/uns-y}	{Xij/Yn/Xij/Yn}
unf-arg_create-var-{x/y/a}	{Xj/Yn/k}
unf-arg[/-tv]_con	Cnt
unf-arg[/-tnv/-tl/-tv]_ref-lst	
unf-arg[/-tnv/-tv]_ref-str-ftr	Ftr
PUT_I	
put_{ix/y/uns-y}_i	{Xij/Yn/Yn}, Xi
put_create-vr[s/m/x]-{x/y}_i	{Xj/Yn}, Xi
put_con_i	Cnt, Xi
put_ref-lst_i	Xi
put_ref-str-ftr_i	Ftr, Xi
set-s/m	Bool, Nx, Ny, ..., Xj, ..., Yn,...
CREATE_ARG: modo WRITE	
create-arg_{ix/y/uns-ix/uns-y}	{Xij/Yn/Xij/Yn}
create-arg_create-vr[s/m/x/v]-{x/y/a}	{Xi/Yn/k}
create-arg_con	Cnt
create-arg_ref-lst	
create-arg_ref-str-ftr	Ftr

Tabla 5.2: *Instrucciones de UNIFICACIÓN*

- Tipo estático de las variables que aparecen en la cláusula. Según este tipo, las variables se clasifican en anónimas, temporales y permanentes.
- Tipo dinámico de las variables. Según este tipo, las variables pueden ser nulas, simples, s/m o múltiples.
- Dominio de términos asociado a los argumentos reales de entrada a la cláusula. El dominio de términos tal como se ha definido en la interpretación abstracta intenta

concretar el tipo de datos y especificar si el término es básico. Los valores que se utilizan durante la compilación son: término general, variable libre, término instanciado (no variable libre), y lista no vacía.

La información especificada en el segundo punto debe conocerse necesariamente ya que el modelo arquitectónico establece que el tipo dinámico de una variable ha de decidirse en el momento de su creación. Sin embargo, la información especificada en el tercer punto es tratada como una optimización del código generado. Por el hecho de restringir el dominio de términos asociado a los argumentos reales es posible simplificar el algoritmo general de unificación. El criterio utilizado durante la compilación consiste en utilizar instrucciones específicas para anotar información calculada en la interpretación abstracta siempre que repercutan en disminuir el grado de complejidad del algoritmo de unificación general.

Para describir el código MAM asociado a la cabecera de cláusula, se particulariza con la siguiente cabecera, que pretende abarcar los diferentes casos que pueden producirse:

h(A, 1, [B|C], B, s(a,[1,2])) :- . . .

Primer argumento (A)

El primer argumento corresponde a una variable. El tipo dinámico de la variable es simple ya que esta variable únicamente debe almacenar un vínculo hacia el correspondiente argumento de entrada. Según el tipo estático de la variable, las posibles codificaciones son las siguientes. Si la variable es anónima (se podría haber escrito como "_"), no hace falta ninguna instrucción para representarla. Si la variable es temporal, se representa como

unf_i_create-var-x (X1, Xj)

teniendo en cuenta que esta instrucción no sería necesaria si el primer argumento no se modifica antes de utilizar la variable A en el primer objetivo del cuerpo de la cláusula. Si la variable es permanente se representa como

unf_i_create-var-y (X1, Yn)

En adelante, la notación utilizada para indicar la posibilidad que una variable sea temporal o permanente consiste en agrupar entre llaves las diferentes opciones. De esta forma, la representación general del primer argumento, teniendo en cuenta que la variable que se crea puede ser temporal o permanente, es

unf_i_create-var-{x/y} (X1, {Xj/Yn})

Segundo argumento (1)

El segundo argumento es una constante numérica. El código MAM depende de la información obtenida acerca del dominio de términos del argumento de entrada asociado.

Si este argumento corresponde a cualquier término Prolog, se representa como

unf_i_con (X2, &1)

Si el argumento es una variable libre, se representa como

unf_i-tv_con (X2, &1)

La notación utilizada para simbolizar distintas alternativas del código de operación que comparten los mismos operandos consiste en agrupar las alternativas entre corchetes. De esta forma, la representación general del segundo argumento es

unf_i[/-tv]_con (X2, &1)

La ventaja de poder identificar la segunda situación consiste en asegurar que la unificación nunca puede fracasar.

Tercer argumento ([B|C])

El tercer argumento corresponde a una lista en la que tanto su car como su cdr corresponden a dos variables.

El siguiente código MAM refleja la situación general, sin información adicional acerca del dominio de términos del argumento de entrada correspondiente:

unf_ix_ref-lst (X3)
unf/create-arg_create-vrs-{x/y} ({Xj/Yn})
unf/create-arg_create-vr[s/m/x/v]-{x/y/a} ({Xj/Yn/1})

donde la primera instrucción especifica la unificación del tercer argumento con una referencia a una lista y las dos siguientes instrucciones realizan la unificación con el car y el cdr de la lista. La unificación de los elementos de términos estructurados se efectúa con instrucciones UNF/CREATE-ARG. Estas instrucciones tratan los dos posibles modos de unificación de elementos de términos estructurados, según si el correspondiente elemento del argumento de entrada ya está creado (modo READ) o si el elemento debe crearse debido a que el argumento de entrada corresponde a una variable libre (modo WRITE) [80].

Siguiendo con el ejemplo, el primer elemento (car) corresponde a una variable simple ya

que aparece también en otro argumento de la cabecera, en concreto, el cuarto. En caso que la unificación de la lista se efectúe en modo WRITE, la variable a crear poseerá un único vínculo que corresponde al término que posea el cuarto argumento de entrada.

En el segundo argumento (cdr) debe considerarse la posibilidad de tener que crear una variable si la unificación se realiza en modo WRITE. En este momento, se utiliza la información obtenida acerca del tipo dinámico de la variable. En el ejemplo se ha especificado la notación general ([s/m/v/x]), que considera las cuatro posibilidades: la variable es simple (s); la variable es múltiple (m); la variable es nula (v) o la variable es s/m (x). En este último caso, las condiciones se comprueban antes de crear la variable mediante instrucciones SET-S/M, que determinan el tipo concreto (simple o múltiple) que debe poseer la variable en tiempo de ejecución. Esta instrucción tiene como argumentos el número de variables temporales y permanentes de que depende la variable, así como sus referencias, y un booleano que indica si depende del tipo de exploración del siguiente objetivo a ejecutar.

Independientemente del tipo dinámico de la variable, ésta puede ser temporal, permanente o anónima, según la clasificación estática. La notación utilizada en el código de operación es {x/y/a}. Cuando la variable es temporal, el operando de la instrucción especifica un registro Xj; cuando es permanente, el operando es Yn; y cuando es anónima, el operando (k) indica el número de elementos consecutivos del término estructurado que se deben unificar.

Si el dominio de términos de este argumento del objetivo padre corresponde a una lista no vacía (tl) o a un término no variable (tnv), se asegura que el modo de unificación es READ. Existen instrucciones específicas de unificación de argumentos cuando el modo es READ. En estas instrucciones no es necesario especificar el tipo dinámico de las variables ya que se calcula siempre en tiempo de ejecución. Otra ventaja adicional que se obtiene cuando el término es una lista no vacía, consiste en asegurar que la unificación con una referencia a lista nunca puede fracasar. En estos dos casos, la representación de este tercer argumento se realizaría con las siguientes instrucciones:

unf_ix[-tl/-tnv]_ref-lst (X3)
unf-arg_create-var-{x/y/a} (Xj/Yn/1)
unf-arg_create-var-{x/y/a} (Xj/Yn/1)

Si el dominio de términos de este argumento del objetivo padre es una variable libre (tv), se garantiza que el modo de unificación es WRITE y, por tanto, se deben crear los elementos de la lista. Por otra parte, cuando el modo de unificación es WRITE se asegura que nunca va a fracasar la unificación de la variable libre correspondiente al argumento real con el término

estructurado de la cabecera. Esta información es importante a la hora de realizar las operaciones de unificación en el modelo propuesto ya que evita tener que sincronizarse con la finalización de las unificaciones que, de forma concurrente, se ejecutan para cada camino recorrido. En esta situación, la codificación se realiza con las siguientes instrucciones:

unf_ix-tv_ref-lst (X3)
create-arg_create-vrs-{x/y/a} (Xj/Yn/1)
create-arg_create-vr[s/m/x/v]-{x/y/a} (Xj/Yn/1)

Cuarto argumento (B)

El cuarto argumento de la cabecera ejemplo, corresponde a una variable existente también en la cabecera que ya ha sido unificada previamente. El tratamiento general es el siguiente:

unf_i_{ix/y} (X4, {Xj/Yn})

donde la variable temporal o permanente {Xj/Yn} debe coincidir con la variable que se haya utilizado en la instrucción anterior de unificación.

Existe otra posibilidad de codificar la misma situación con una instrucción menos. Respecto a la descripción realizada para el tercer argumento, puede evitarse la instrucción anterior si se substituye la instrucción de creación de la variable del car por una de estas tres instrucciones:

unf/create-arg_uns-ix (X4), unf-arg_uns-ix (X4) o create-arg_uns-ix (X4)

según se trate del modo general de unificación, del modo READ o del modo WRITE, respectivamente. En este caso, debe tenerse en cuenta que la variable apuntada por X4 es insegura: puede hacer referencia a una variable permanente que está libre, almacenada en un entorno eliminado o que puede ser eliminado posteriormente debido a la optimización de la recursividad de cola. La información referente al dominio de términos también se utiliza en estos casos. Existen tres instrucciones análogas a las anteriores que son utilizadas cuando el dominio del argumento de entrada no corresponde a una variable libre y, por tanto, no hace falta gestionarla como insegura:

unf/create-arg_ix (X4), unf-arg_ix (X4) o create-arg_ix (X4)

Quinto argumento (s(a, [1, 2]))

El quinto y último argumento de la cabecera corresponde a una estructura. Sin tener información específica del dominio de términos del argumento de entrada, su codificación es:

unf_ix_ref-str-ftr (X5, s/2)
unf/create-arg_con (a)
unf/create-arg_create-vrs-x (X6)
unf_ix_ref-lst (X6)
unf/create-arg_con (&1)
unf/create-arg_create-vrs-x (X7)
unf_ix_ref-lst (X7)
unf/create-arg_con (&2)
unf/create-arg_con ([])

Cuando el argumento de entrada no corresponde a una variable libre, se especifica el modo READ de unificación. Si a su vez, algún elemento de la estructura vuelve a ser otro elemento estructurado, se permite indicar el modo de unificación si es conocido:

unf_ix-tnv_ref-str-ftr (X5, s/2)
unf-arg_con (a)
unf-arg-tl_ref-lst
unf-arg_con (&1)
unf-arg-tl_ref-lst
unf-arg_con (&2)
unf-arg_con ([])

Si el argumento corresponde a una variable libre, se utilizan instrucciones de unificación en modo WRITE. Cualquier otra estructura anidada también puede especificarse de forma segura en modo WRITE:

unf_ix-tv_ref-str-ftr (X5, s/2)
create-arg_con (a)
create-arg_ref-lst
create-arg_con (&1)
create-arg_ref-lst
create-arg_con (&2)
create-arg_con ([])

Para finalizar este apartado de descripción de la codificación de la cabecera de una cláusula, nótese que el orden de representación de sus argumentos no tiene que realizarse forzosamente en el orden de escritura. Es posible reordenar el código de cada argumento si ello puede conducir a disminuir el tiempo de ejecución del programa. Las únicas dependencias que hay que respetar son:

- La instrucción de creación de una variable debe preceder a cualquier instrucción de

acceso a la misma variable.

- Las instrucciones que manipulan los elementos de términos estructurados deben seguir el orden textual de dichos elementos.

5.7.5 Cuerpo de una cláusula

La especificación de cada uno de los predicados u objetivos del cuerpo de una cláusula se realiza mediante las instrucciones del tipo PUT, que especifican sus argumentos, y de la instrucción CALL, que especifica el punto de entrada correspondiente para resolver el objetivo. En el último objetivo del cuerpo de la cláusula, la instrucción CALL se debe substituir por la instrucción EXEC.

Existen distintas instrucciones PUT para representar los posibles términos de los argumentos. La tabla 5.2 muestra todas las instrucciones de tipo PUT existentes en la MAM. Las instrucciones son análogas a las utilizadas en la especificación de la cabecera, pero con la simplificación que ahora siempre se deben crear los términos, por lo que las unificaciones son en modo WRITE.

A continuación se describe brevemente la representación de un argumento i del cuerpo de una cláusula según los cuatros tipo de datos que son soportados.

Si un argumento corresponde a una variable, cabe diferenciar si es la primera vez que se utiliza (por tanto se crea la variable):

put_create-vr[s/m/x]-{x/y}_i (Xj/Yn, Xi)

donde el tipo de la variable puede simple (s), múltiple (m) o s/m (x).

Si la variable ya ha sido utilizada previamente o aparece en la cabecera de la cláusula (con lo cual no se debe crear):

put_{ix/y}_i ({Xj/Yn}, Xi)

Si se referencia una variable permanente en el último objetivo de la cláusula, debe utilizarse la instrucción que considera el caso de una variable insegura:

put_uns-y_i (Yn, Xi)

En esta última situación, puede utilizarse la instrucción que referencia una variable permanente siempre que el dominio de términos de la variable indique que no es libre.

Si un argumento corresponde a un término constante se representa:

put_con_i (constante, Xi)

Si un argumento corresponde a una estructura se representa:

put_ref-str-ftr_i (functor, Xi)
create-arg_... (elem 1)
create-arg_...
create-arg_... (elem aridad)

Si un argumento corresponde a una lista se representa:

put_ref-lst_i (Xi)
create_arg_... (car)
create_arg_... (cdr)

Cuando el último elemento de una lista o estructura es otro término estructurado puede utilizarse la instrucción:

create-arg_ref-lst o create-arg_ref-str-ftr

para iniciar la descripción de los elementos de la estructura anidada.

Por último, la especificación del punto de entrada correspondiente al objetivo se realiza de la forma:

call[/-nondet/-semidet] (Lep, N)

donde se observa que el código de operación especifica el atributo de indeterminismo del objetivo (si no se indica nada, se sobreentiende que es OTHER) y el segundo parámetro de la instrucción (N) indica el número de variables permanentes existentes en ese momento en el entorno. Se utiliza la misma optimización que en la WAM (environment trimming) consistente en numerar las variables permanentes en orden creciente según el objetivo en que se utilizan por última vez: cuanto antes deja de utilizarse una variable, mayor es su número. De esta forma, puede disminuir dinámicamente el espacio necesario para almacenar el entorno de una cláusula.

El apéndice B contiene el código MAM completo para uno de los programas benchmark que se han utilizado en la evaluación del sistema Multipath.

5.8 Resumen y contribuciones

En este capítulo se ha descrito la fase de interpretación abstracta del Modelo Arquitectónico de Multipath. La misión de esta fase es realizar un análisis global de determinismo y de tipos de datos del programa de cara a transformar el código original y calcular información dinámica.

Este proceso permite optimizar la ejecución real del programa. Se ha descrito también la codificación de la salida de esta fase mediante instrucciones MAM. El programa obtenido constituye la entrada de la siguiente fase de ejecución, denominada interpretación real.

Las principales contribuciones que se aportan en esta fase del modelo arquitectónico son las siguientes:

- Definición de un dominio abstracto de tipos de datos de los argumentos y variables de un programa, junto con un dominio abstracto de determinismo de los objetivos, que permiten realizar las acciones propuestas por el modelo de ejecución.
- Definición de las funciones caracterizadoras del análisis de determinismo y de tipos de datos del programa. Se representan a partir de una compilación abstracta de las cláusulas que constituyen el programa.
- Evaluación recursiva de todas las funciones caracterizadoras.
- Definición de las instrucciones MAM y la representación de los términos Prolog. De forma coherente, estos elementos arquitectónicos están integrados dentro de la fase de interpretación real de un programa, pero necesitan ser conocidos durante la interpretación abstracta.

6

MODELO ARQUITECTÓNICO DE MULTIPATH: INTERPRETACIÓN REAL

El modelo de ejecución Multipath descrito en el capítulo 4 establece dos fases de ejecución: una interpretación abstracta del programa y una posterior interpretación real del programa. El modelo arquitectónico de la primera fase ha sido descrito en el capítulo 5. En este capítulo se describe el modelo arquitectónico correspondiente a la fase de interpretación real de un programa.

El modelo arquitectónico de la interpretación real en el sistema Multipath se basa en el propuesto por la WAM, al cual se le añaden los elementos propios que facilitan la realización del modelo de ejecución. Este modelo arquitectónico se denomina **Máquina Abstracta de Multipath (MAM)** [124].

La primera característica diferenciadora del modelo propuesto en la MAM es la definición de dos tipos de *motores* que colaboran en la ejecución de un programa. La principal ventaja de definir estos dos tipos de motores radica en la posibilidad de hacer factible tanto una rea-

lización secuencial como una realización paralela del sistema Multipath, en función de los recursos que se disponga. El tipo de sincronización entre las tareas a ejecutar por los dos motores y el acceso remoto a estructuras de datos cumplen una serie de restricciones que facilitan la realización concreta del sistema en un amplio abanico de arquitecturas paralelas.

En las próximas secciones se introducen los dos tipos de motores existentes en la MAM, caracterizados por los elementos arquitectónicos que los componen (instrucciones, zonas de memoria, estructuras de datos y registros), así como los diferentes estados en que se pueden encontrar. Recuérdese que los términos Prolog soportados se han descrito en el capítulo 5 con el fin de justificar las acciones a realizar durante la fase de interpretación abstracta. Posteriormente, se describe la representación utilizada en la MAM del estado de la ejecución y la realización las distintas operaciones que lo actualizan. Esta descripción del modelo arquitectónico de la MAM está enfocada a introducir la necesidad de sus elementos arquitectónicos y ubicarlos correctamente en el motor pertinente.

6.1 Motores

El modelo de ejecución de Multipath permite la explotación del denominado paralelismo de caminos por la existencia de operaciones a realizar para cada uno de los datos visibles en los caminos recorridos simultáneamente. El modelo arquitectónico introduce el concepto de **motores** (*engines*) como el mecanismo necesario para poder ejecutar las tareas concurrentes existentes en Multipath y proporciona la potencialidad necesaria para permitir realizaciones concretas del sistema tanto en arquitecturas secuenciales como en arquitecturas paralelas. Existen dos tipos de motores (principal y de unificación) que son descritos en los dos apartados siguientes.

6.1.1 Motor Principal (ME)

El **Motor Principal**, denominado **ME**, tiene la responsabilidad de controlar la ejecución de todas las operaciones descritas en el modelo de ejecución. Además, ejecuta totalmente las funciones que determinan la exploración del árbol de búsqueda y las operaciones que afectan de forma global a varios caminos de este árbol de búsqueda. Existe un ME en la MAM.

El ME se caracteriza por las zonas de memoria, estructuras de datos y registros que gestiona (figura 6.1), así como las instrucciones que es capaz de ejecutar (tabla 5.1 y tabla 5.2). En las siguientes secciones se describe de forma detallada la necesidad de cada una de estas zonas de memoria. Una primera descripción del uso de estas **zonas de memoria** es la siguiente: las memorias CODE y STRINGS almacenan las instrucciones de que consta un programa; la

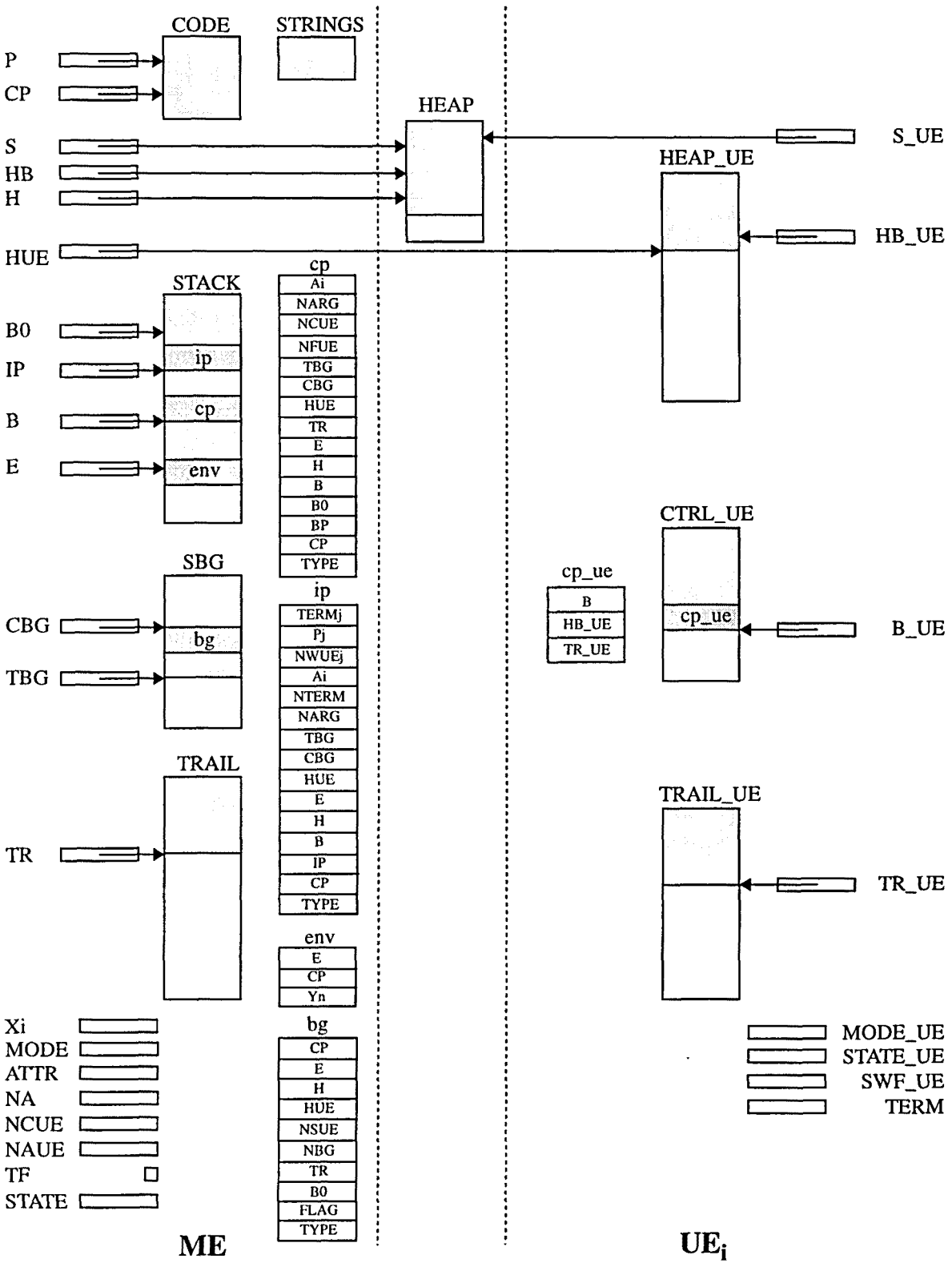


Figura 6.1: *Elementos arquitectónicos de la MAM distribuidos según pertenezcan al ME o a un UE.*

memoria HEAP almacena términos Prolog estructurados, junto con variables temporales y variables inseguras; la memoria STACK almacena las estructuras de datos que representan a los puntos de selección (cp) y a los puntos de indexación (ip), y también almacena las variables permanentes en las estructuras de datos denominadas entornos (env); la memoria TRAIL almacena direcciones de variables vinculadas que han sido creadas anteriormente a un punto de selección con alternativas pendientes; y la memoria SBG almacena estructuras de datos denominadas bg, que contienen información asociada a cada objetivo en anchura.

Por otra parte, el ME puede estar en dos **estados**: RUNNING o SUSPENDED. El registro STATE indica el estado en que se encuentra el ME. Inicialmente, el estado del ME es RUNNING. Las causas que pueden provocar un cambio de estado se describen en las próximas secciones.

6.1.2 Motor de Unificación (UE)

Un **Motor de Unificación**, denominado **UE**, tiene la responsabilidad de ejecutar aquellas operaciones que afectan de forma local a los caminos del árbol de búsqueda. Estas operaciones son indicadas mediante comandos por el ME. El número de UEs es un parámetro del modelo arquitectónico denominado **NUM_UE**. Los UEs no tienen una visión completa del programa que se ejecuta. Únicamente están a la espera de recibir comandos y procesarlos adecuadamente para retornar el resultado al ME.

Las zonas de memoria, estructuras de datos y registros que gestionan los UEs se muestran en la figura 6.1. En una primera visión, cada UE dispone de tres **zonas de memoria**: HEAP_UE, que almacena los vínculos de variables múltiples; CTRL_UE, que almacena estructuras de datos, denominadas cp_ue, para gestionar los puntos de selección; y TRAIL_UE, que almacena las direcciones de variables múltiples creadas antes de cualquier cp_ue que posea el UE.

El número de caminos que gestiona cada UE es una decisión arquitectónica que influye notablemente en el rendimiento del sistema. En un primer prototipo se comprobó la pérdida de rendimiento si cada UE gestiona más de un camino simultáneamente. Por ello, cada UE es responsable de un único camino y, en consecuencia, el parámetro NUM_UE coincide con el parámetro NUM_CAMINOS definido en el modelo de ejecución.

El registro STATE_UE almacena el estado de un UE. Los **estados** en que se puede encontrar un UE son:

- **CURRENT**. El UE está activo y es el responsable de gestionar uno de los caminos que se recorren simultáneamente.

- **SOLUTION(bg)**. El UE es el encargado de gestionar un camino que representa una solución calculada previamente de un objetivo en anchura (bg). El objetivo en anchura se identifica por el contenido del registro SWF_UE.
- **WAITING(ip,term)**. El UE se asocia a un término Prolog (term) de un punto de indexación (ip) que aún falta por explorar. El punto de indexación se identifica a partir del registro SWF_UE. El término Prolog se identifica mediante el registro TERM_UE.
- **FAILED(cp)**. El UE se asocia a un camino que ha fracasado pero que tiene puntos de selección con alternativas aún por explorar. El punto de selección más joven (cp) está almacenado en el registro SWF_UE.
- **AVAILABLE**. Correspondiente a un UE que está disponible para ser utilizado en la exploración en anchura de otro camino del árbol de búsqueda.

En todos los estados, excepto CURRENT, un UE está inactivo o suspendido, en el sentido que no corresponde a ninguno de los caminos que se recorren simultáneamente. Inicialmente, existe un UE en estado CURRENT, correspondiente al único camino que se explora en el momento de comenzar la ejecución del programa, y NUM_UE-1 UEs en estado AVAILABLE, es decir, disponibles para poder ser utilizados posteriormente en un recorrido en anchura. Las causas que pueden provocar un cambio de estado se describen en las siguientes secciones.

El ME necesita conocer el número de UEs que están en cada uno de los diferentes estados para poder decidir el tipo de exploración del árbol. Por ello, existen dos registros NCUE y NAUE que indican el número de UEs en estado CURRENT y AVAILABLE, respectivamente; y 3 campos (NFUE en un cp; NWUE en cada alternativa de un ip; NSUE en un bg) que indican el número de UEs en estado FAILED(cp), WAITING(ip,term) y SOLUTION(bg), respectivamente.

6.2 Representación del estado de la ejecución en la MAM

El estado de la ejecución en Multipath se caracteriza por los siguientes elementos:

$$EJ_{IR} \equiv \{BD,\ OAA,\ OPA,\ EAV,\ AP,\ COA\}$$

A continuación se describen los elementos arquitectónicos que lo representan y se indica su ubicación en el motor pertinente.

6.2.1 Representación de BD

La Base de Datos (BD) está constituida por la salida proporcionada por la interpretación abstracta. Esta información era denominada PRG e INFO a nivel de modelo de ejecución:

$$
\begin{aligned}
PRG &= \{ep_1, \ldots, ep_i\} \\
&\quad EP = \{\ [cl_1,\ldots,\ cl_j] \setminus \{arg,\ \{\{term_1,[cl_{11},\ldots,\ cl_{1j}]\},\ \ldots,\ \{term_n,[cl_{n1},\ldots,\ cl_{nj}]\}\ \}\ \} \\
&\quad CL = [hd,\ gl_1,\ \ldots,\ gl_k] \\
&\quad HD = [arg_1,\ \ldots,\ arg_{hd}] \\
&\quad GL = \{ep_k,\ [arg_1,\ \ldots,\ arg_{gl}]\} \\
INFO &= \{ \\
&\quad ATTR(gl) = \{NONDET,\ SEMIDET,\ OTHER\} \\
&\quad TYPE(var) = \{VOID,\ SINGLE,\ S/M(traversal,\ \{var\}),\ MULTIPLE\} \\
&\quad \}
\end{aligned}
$$

En la MAM, al igual que en la WAM, la BD es activa, por lo que en su representación se utilizan instrucciones que son interpretadas para realizar las operaciones del modelo de ejecución. Las instrucciones MAM han sido descritas en el capítulo 5. La BD se almacena en las memorias CODE y STRINGS pertenecientes al ME. La memoria CODE almacena los puntos de entrada a procedimientos, cláusulas y objetivos del programa. La memoria STRINGS almacena los códigos ASCII correspondientes a las constantes atómicas existentes en el programa.

6.2.2 Representación de EAV

El Entorno Actual de Vínculos (EAV) es el elemento que simboliza, en el modelo de ejecución, todos los vínculos de las variables visibles en los caminos que se recorren simultáneamente y que se han obtenido en las operaciones de unificación realizadas desde el inicio del programa hasta el momento actual de la ejecución.

A nivel de modelo de ejecución, EAV se considera dividido en dos partes: un Entorno de Vínculos Global (EVG), que representa los vínculos visibles globalmente en todos los caminos; y un conjunto de Entornos de Vínculos Locales (EVL), formado por los vínculos locales a cada camino que se recorre:

$$EAV \equiv \{\{EVL_1,\ \ldots,\ \{\ldots,\ var/vínculo,\ \ldots\},\ \ldots,\ EVL_{NCA}\},\ EVG\}$$

A nivel de modelo arquitectónico, debe especificarse cómo se representan los vínculos del EAV y cómo se obtiene la substitución de una variable. La figura 6.2 muestra una representación gráfica de las variables visibles en un momento del programa y sus vínculos, que constituyen a nivel lógico el hasta ahora mencionado Entorno Actual de Vínculos.

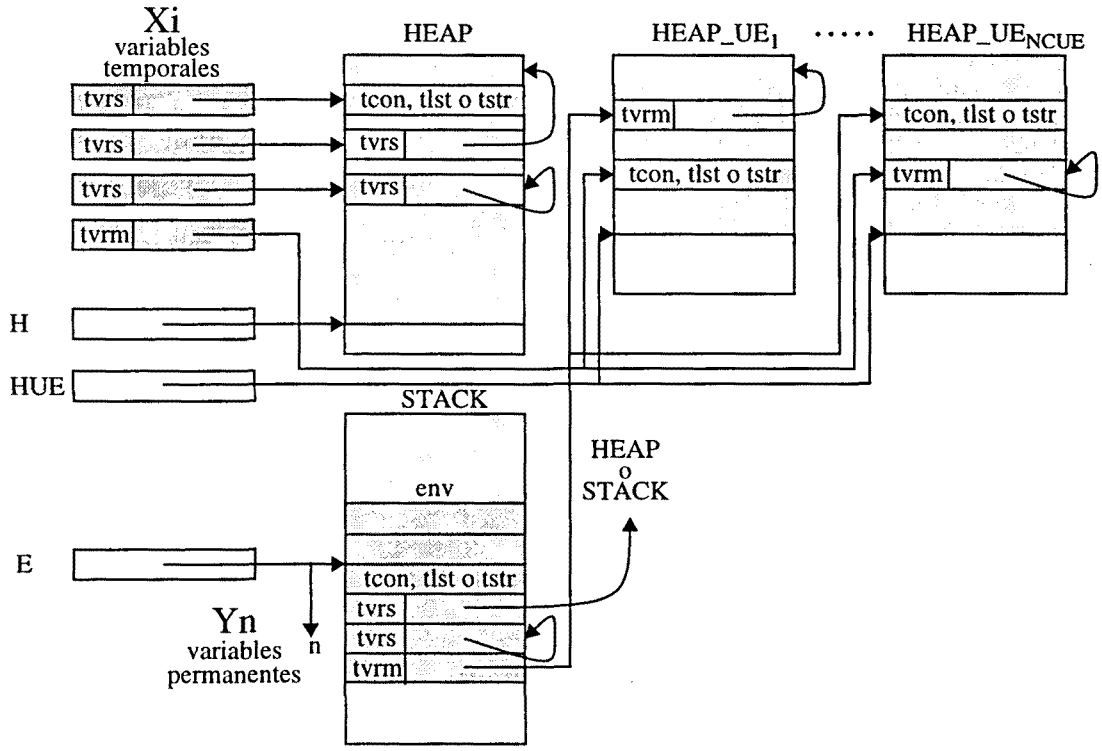


Figura 6.2: *Representación de EAV en la MAM.*

Debe observarse que cuando dos variables están vinculadas, solamente se almacena una referencia de una variable hacia la otra. Por otra parte, cuando una variable está vinculada a un término no variable, el término se almacena en aquella variable con la que esté vinculada que aún no posea un vínculo. Por ello, sólo existe un vínculo por variable visible en cada camino. De esta forma, si una variable es simple, debe guardarse un único vínculo, que es global a todos los caminos; y si una variable es múltiple, debe poder almacenarse un vínculo distinto para cada camino recorrido simultáneamente. A continuación se describe cómo se identifica una variable del Entorno de Vínculos y dónde se almacena su/s vínculo/s según sea simple o múltiple.

Las variables visibles en un cierto instante son las variables temporales y las variables permanentes. No se considera en este apartado a las variables nulas ya que sus vínculos no se utilizan nunca. La referencia a una variable temporal está contenida en un **registro Xi**. Si el campo tag del registro indica que es una variable simple, la dirección (campo val) corresponde a la memoria HEAP. Si el tag indica que es una variable múltiple, la dirección corresponde a las memorias HEAP_UE. Esta dirección de memoria contiene el vínculo de la variable. En

caso de ser múltiple, los distintos vínculos se almacenan en la misma posición de todas las memorias HEAP_UE.

La dirección de una variable permanente (Yn) se calcula mediante la suma de la dirección base del entorno actual (contenido del registro E) más el desplazamiento, que corresponde con el número de la variable permanente. Si la variable es simple (se detecta porque su campo tag es distinto de tvrm), el contenido de esta posición de memoria almacena el vínculo. Si la variable es múltiple, el campo val almacena la dirección de las memorias HEAP_UE donde están almacenados los vínculos correspondientes.

Si una variable no está vinculada con ningún término, el contenido de su dirección corresponde a una variable que apunta a la misma dirección de memoria. En la MAM se cumple que cuando una variable simple apunta a otra, siempre se almacena el puntero en la variable con dirección mayor, considerando el espacio de direcciones de la memoria STACK más alto que el de la memoria HEAP. De esta manera se evita que pueda haber referencias incorrectas a variables creadas anteriormente cuando se elimina el espacio asignado por las variables más jovenes. Por su parte, cuando una variable múltiple apunta a otra, también se almacena el puntero en la que tiene una dirección mayor. Una variable múltiple nunca puede apuntar a una simple, pero sí al revés. En este caso, siempre la variable simple es más joven que la múltiple.

Las zonas de memoria que representan a EAV se caracterizan por el espacio utilizado desde el inicio del programa hasta el momento actual de la ejecución. El registro H apunta a la primera posición libre de la memoria HEAP, mientras que el registro HUE apunta a la primera posición libre de las memorias HEAP_UE. Por su parte, el registro E apunta al entorno actual y limita el espacio de la memoria STACK que se utiliza en la representación del Entorno de Vínculos. Las memorias HEAP_UE que contienen los vínculos de las variables múltiples son aquellas que pertenecen a los UEs que están en estado CURRENT. El número de UEs en este estado se almacena en el registro NCUE.

Durante la operación de unificación, es preciso substituir los vínculos de las variables que intervienen. Para conocer si la variable está instanciada a un término o si la variable es libre, se aplica la operación de derreferenciación.

Un aspecto importante de la MAM es la determinación del motor responsable de acceder a las variables según se trate de una variable simple o múltiple. En este sentido, el ME es el responsable de gestionar las variables simples, mientras que las variables múltiples son gestionadas por los UEs. Los registros H, HUE y E son accedidos únicamente por el ME. Cualquier acceso a una variable múltiple provoca el envío de un comando hacia los UEs en estado

CURRENT para que terminen la tarea que debe realizarse. En el comando se especifica la acción y la/s dirección/es de las variables múltiples implicadas.

El acceso a cada una de las zonas de memoria que representa a EAV cumple una serie de restricciones: la memoria STACK únicamente es accedida por el ME; la memoria HEAP es accedida por el ME (para lectura o escritura) y por los diferentes UEs (únicamente para lectura), y las memorias HEAP_UE únicamente son accedidas por los UEs.

6.2.2.1 Representación del EAV en los UEs

Otra cuestión importante a determinar viene provocada por el hecho de que los UEs se crean dinámicamente a lo largo de la ejecución de un programa y, por tanto, hay que decidir la representación de la parte del EAV que es visible para el nuevo UE. Este aspecto también se considera en aquellos sistemas que explotan el paralelismo O, cuando se crean varios procesos para intentar resolver un objetivo, cada uno de estos procesos con una cláusula alternativa diferente.

Existen tres grandes alternativas para resolver esta cuestión en los sistemas orientados a paralelismo O [33]:

- **Compartición de Entornos.**
  Un proceso comparte los vínculos comunes con otros procesos creados posteriormente. La técnica más utilizada basada en esta alternativa es Hash-Windows [9].

- **Inicialización de Entornos.**
  En el momento de creación de un proceso se inicializa su Entorno de Vínculos con todos aquellos que le son visibles. La técnica más utilizada basada en esta alternativa es Binding Arrays [133].

- **Recálculo de Entornos.**
  Todos los procesos tratan de resolver el programa desde el principio, por lo que cada uno de ellos calcula su Entorno de Vínculos de forma independiente. Sistemas que utilizan esta idea son: Prolog Multi-Sequential Machine [2] y su sucesor BC-machine [3].

La tercera alternativa no se puede aplicar en la MAM ya que este modelo arquitectónico está orientado tanto a una ejecución secuencial como paralela. No tiene sentido en una ejecución secuencial que cada UE vuelva a ejecutar las instrucciones MAM para obtener los vínculos visibles. Sin embargo, las dos primeras alternativas pueden llegar a ser realizadas en la MAM.

Las ventajas e inconvenientes de estas alternativas se indican a continuación.

La compartición de entornos evita la penalización en cuanto a tiempo de ejecución para realizar la inicialización de las zonas de memoria pertinentes. Sin embargo, tiene el inconveniente de que el tiempo de acceso a una variable no es constante al tener que comprobar si una variable está ubicada en el espacio de direcciones propio o en el de otro proceso. La obtención del vínculo de una variable puede suponer tener que recorrer una cadena de procesos predecesores para encontrar el valor válido. Esta alternativa está orientada hacia aquellos sistemas con memoria compartida.

La inicialización de entornos posee la ventaja de permitir un acceso en tiempo constante a los vínculos de las variables pero posee el inconveniente de la penalización en tiempo de ejecución durante la inicialización de un proceso. En esta alternativa, cada proceso accede únicamente a su memoria local de vínculos, excepto en la inicialización en que debe accederse a la memoria de otro u otros procesos. Esta alternativa puede realizarse en sistemas de memoria compartida o distribuida.

Concretando, en la MAM se realiza la inicialización de entornos basada en la copia de todos los vínculos de variables múltiples obtenidos desde el inicio del programa hasta el momento de la creación de un UE. La creación dinámica de un UE se produce cuando se decide recorrer en anchura una alternativa del árbol de búsqueda después de haber encontrado una solución. Las acciones que se efectúan en este momento son las siguientes:

- Se reserva un UE_j disponible (en estado AVAILABLE).
- Supongamos que UE_i es el UE que estaba en estado CURRENT en el momento de crear el punto de selección asociado a la alternativa a recorrer. Se copia el Entorno de Vínculos Local (EVL, almacenado en la memoria HEAP_UE) perteneciente a UE_i sobre el EVL perteneciente a UE_j.
- A partir de este momento UE_j es responsable de recorrer el mismo camino que llevaba hasta ahora UE_i.
- UE_i restaura el estado de su EVL con el contenido existente en el momento de crear el punto de selección (operación de backtracking). A partir de este momento, UE_i es responsable de recorrer el camino asociado a la alternativa que se pretende explorar.

La figura 6.3 muestra gráficamente las acciones que se realizan durante la inicialización de un nuevo UE.

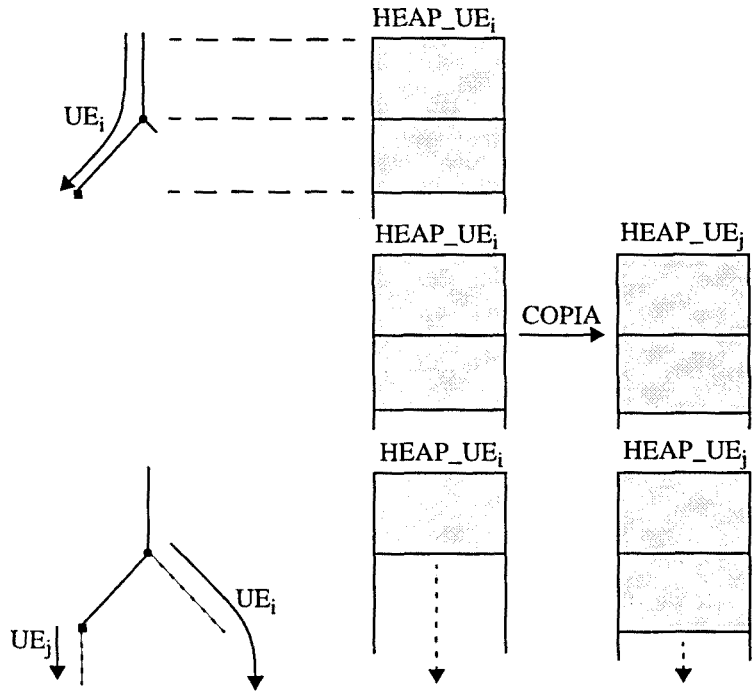


Figura 6.3: *Inicialización de UEs.*

6.2.3 Representación de OAA y COA

En el modelo de ejecución, OAA corresponde al objetivo en anchura que se está intentado resolver en un momento determinado, y se identifica por la lista de objetivos a resolver tras la resolución del objetivo. Por su parte, COA corresponde al conjunto de todos los objetivos en anchura que están activos en un momento de la ejecución. Para cada uno de estos objetivos (OA) se mantiene información sobre todas las soluciones calculadas (EV) y el siguiente objetivo en anchura que hay que intentar resolver cuando se comience a ejecutar su continuación (OAS):

$$OAA \equiv [gl_1, \ldots, gl_{oaa}]$$
$$COA \equiv \{\ldots, \{OA, EV, OAS\}, \ldots\}$$

En la MAM existe una nueva zona de memoria que representa a COA, perteneciente al ME y que se denomina SBG. Esta memoria contiene estructuras de datos, denominadas bg, que almacenan la información que caracteriza a un objetivo en anchura, es decir, OA, EV y OAS. Cada uno de estas estructuras bg se identifica por la dirección donde se encuentra almacenada en la memoria SBG. Por otra parte, la figura 6.4 esquematiza los elementos arquitectónicos de la MAM que representan a COA y OAA. A continuación se describen estos elementos arquitectónicos.

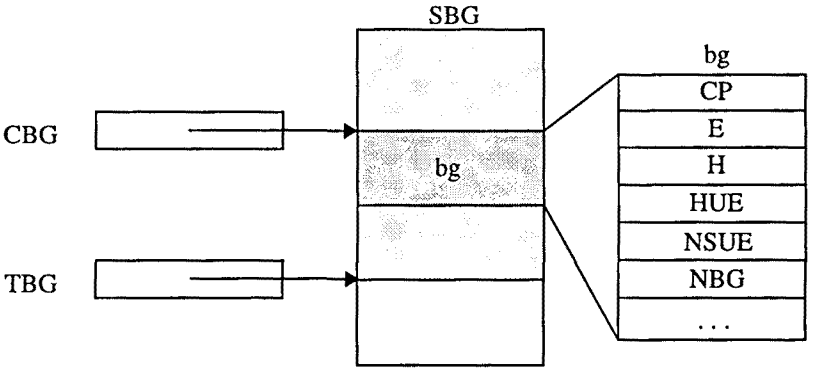


Figura 6.4: *Representación de OAA y COA en la MAM.*

Empezando por OA, la lista de continuación de un objetivo (OP) se representa mediante su dirección de inicio a la memoria de código (campo CP) y las siguientes direcciones de continuación almacenadas en los entornos (campo env.CP) de la memoria STACK. El campo E de un bg contiene la dirección del primer entorno.

Los Entornos de Vínculos (EV) correspondientes a las soluciones ya calculadas son gestionados por los UEs en estado SOLUTION. La representación de estos Entornos coincide con la realizada en el apartado anterior para los Entornos Actuales. En la estructura bg se almacena el espacio ocupado por las zonas de memoria que lo representan, mediante los campos H, HUE y E. Para conocer el número de UEs que se encuentran en este estado se utiliza el campo NSUE.

El Objetivo en Anchura Siguiente (OAS) se representa mediante el campo NBG que contiene la dirección base de la información asociada a este objetivo.

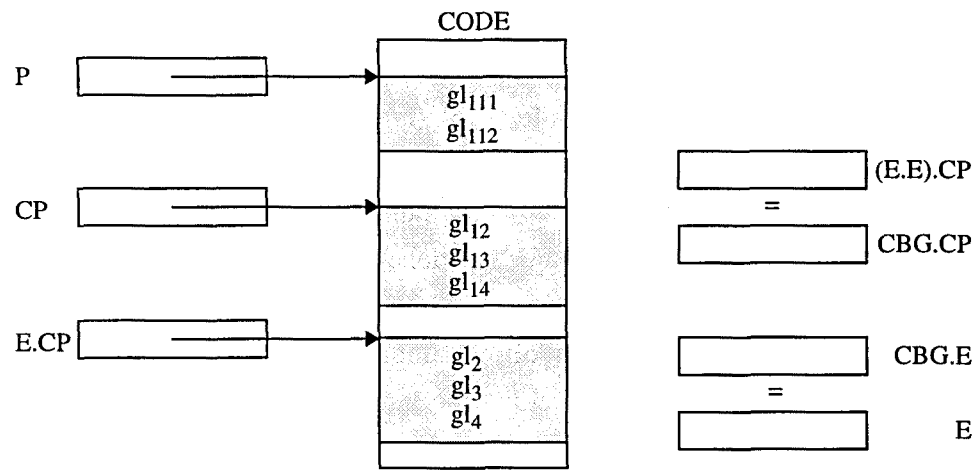
El resto de campos de la estructura bg se utilizan en la realización de las operaciones del modelo arquitectónico y no corresponden a la representación de COA.

El Objetivo en Anchura Actual OAA se representa mediante el registro CBG, que contiene la dirección base dentro de SBG de la estructura bg asociada a este objetivo.

El registro TBG delimita todos los objetivos en anchura que son visibles en un momento de la ejecución. Este registro apunta a la primera posición libre de SBG.

6.2.4 Representación de OPA

OPA es la denominación utilizada en el modelo de ejecución que indica la lista de objetivos que faltan por resolver para solucionar el objetivo en anchura actual. Cada objetivo se identifica



OPA = [gl_{111}, gl_{112}, gl_{12}, gl_{13}, gl_{14}, gl_2, gl_3, gl_4]

Figura 6.5: *Ejemplo de representación de un OPA en la MAM.*

por el punto de entrada asociado, los argumentos, y el atributo de indeterminismo (α) asociados:

$$OPA \equiv [\ldots, \{ep_k, [arg_1, \ldots, arg_m], \alpha\}, \ldots]$$

En la MAM esta lista viene implícitamente representada a partir del registro P, que apunta a la dirección de la memoria de código que contiene el primer objetivo; el registro CP, que apunta al primer objetivo de continuación; y el registro E, que apunta al primer entorno (env) de la memoria STACK, a partir del cual y en el campo E, se pueden obtener el resto de direcciones de continuación.

Como que los entornos (env's) de la STACK están encadenados desde el inicio del programa, se debe establecer la condición que determina la finalización de OPA, es decir, el momento en que se soluciona el objetivo en anchura actual (OAA). La finalización de OPA se produce cuando una dirección de continuación coincide con la dirección de inicio de los Objetivos Pendientes (OP) que identifican al objetivo en anchura actual (OAA), y que está almacenada en el campo CP del elemento apuntado por CBG.

En la figura 6.5 se muestran los elementos arquitectónicos que se utilizan en la representación de OPA.

6.2.5 Representación de AP

Las Alternativas Pendientes AP que establece el modelo de ejecución corresponden a puntos

de selección o puntos de indexación:

$$AP \equiv (\ldots, \{cp, OA, OP, EV\}, \ldots, \{ip, OA, OP, [\ldots, \{term_i, EV_i\}, \ldots]\},\ldots)$$

En la MAM existen dos estructuras de datos (cp e ip, ver la figura 6.6) que contienen toda la información que identifica a estos dos puntos. Estas estructuras de datos se almacenan en la memoria STACK. Ambos puntos tienen en común los campos que representan OA y OP. OA se identifica mediante el campo CBG que contiene la dirección base del elemento correspondiente de SBG. OP se identifica mediante los campos asociados a los registros Xi (corresponden a los argumentos del primer objetivo), el campo CP y la dirección del primer entorno guardada en el campo E. El punto de selección o de indexación más joven se accede mediante el registro B.

Un punto de selección se almacena en la estructura de datos cp. Además de la información anterior, contiene el Entorno de Vínculos (EV). El EV de un punto de selección está asociado a UEs que estaban en estado CURRENT en el momento de su creación. Este EV puede haber sido modificado en algún momento posterior de la ejecución. Por ello, en un cp se almacena el espacio ocupado por cada zona de memoria del EV en el momento de su creación (en los campos H, E y HUE) más la información que permite recuperar este estado en un momento posterior.

Para poder recuperar posteriormente el estado de un EV, se almacenan todos los vínculos que se producen desde el momento de la creación del punto de selección en una zona de memoria denominada TRAIL. Esta zona de memoria también está dividida en una parte global y una parte local. La parte global, direcciones de variables simples, se almacena en la memoria TRAIL perteneciente al ME. La parte local, direcciones de variables múltiples, se almacena en memorias TRAIL_UE, cada una de ellas perteneciente a un UE. Para restaurar el estado de EV existente en el momento de crear un cp se deshacen todos los vínculos producidos desde ese momento hasta el instante de la ejecución en que se quiere recuperar su estado.

Los vínculos de variables simples producidos desde la creación de un cp hasta el momento actual de la ejecución están comprendidos entre la dirección almacenada en el campo TR del cp y la dirección actual de la memoria TRAIL, que contiene el registro TR. Por otra parte, se introduce una nueva estructura de datos, denominada cp_ue, que es almacenada en la memoria CTRL_UE de un UE. De esta forma, los vínculos de variables múltiples producidos en cada UE son todos aquellos existentes entre la dirección indicada por el campo TR_UE de un cp_ue y la dirección de la cima de la memoria TRAIL_UE, que contiene el registro TR_UE.

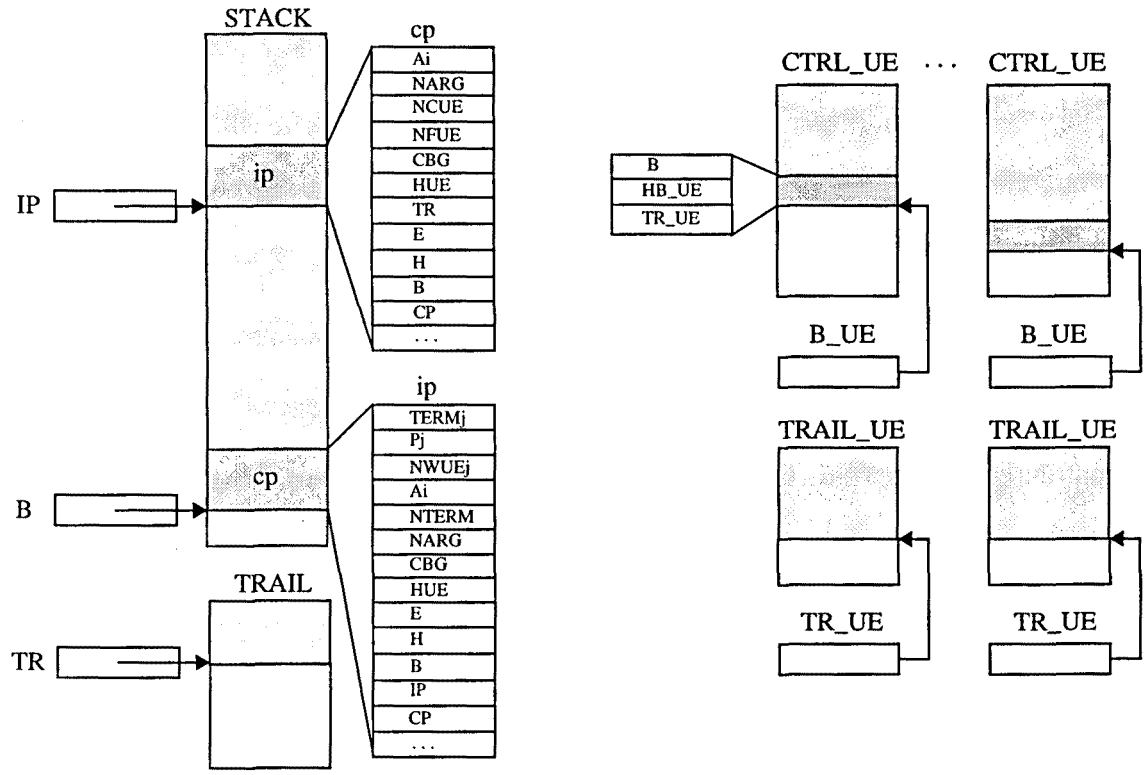


Figura 6.6: *Representación de AP en la MAM.*

Esto significa que los UEs que fracasan en la ejecución y que mantienen algún punto de selección con alternativas pendientes no pueden liberarse (pasar a estado AVAILABLE), sino que pasan a un estado FAILED, preveyendo que, posteriormente, el ME determine que es necesario recuperar el estado existente de un EVL en el momento de la creación de un punto de selección.

En resumen, un EV de un punto de indexación se identifica mediante el campo NCUE almacenado en un cp, que indica el número de EVL existentes en el momento de la creación del punto de selección; y el campo NFUE, que indica el número de UEs en estado FAILED en un momento concreto de la ejecución. El resto de EV se encuentran en estado CURRENT y, por tanto, se conoce el número concreto mediante el registro NCUE. Para poder obtener el estado del EV en el momento de crear el cp se deben deshacer todos los vínculos producidos en los UEs en estado CURRENT y en estado FAILED(cp), y se restauran los punteros a la cima de las zonas de memoria que contienen las variables.

Un punto de indexación se almacena en la estructura de datos ip. El registro IP apunta al ip más joven. Además de la información común con los puntos de selección, contiene una

lista formada por los diferentes términos que faltan por tratar, así como los Entornos de Vínculos asociados a cada término en el momento de crear el punto de indexación. Los UEs responsables de gestionar los EVL de estos Entornos de Vínculos se encuentran en el estado WAITING(ip,term), suspendidos esperando que en algún momento el ME decida explorar su alternativa correspondiente.

La información específica de un ip corresponde al número de alternativas o términos por tratar, que se guarda en el campo NTERM. Además, para cada alternativa contiene el término asociado al argumento indexado (campo TERMj), la dirección a la memoria CODE de la lista de cláusulas susceptible de unificar (campo Pj), el espacio de las zonas de memoria que almacenan los EV (campos H, HUE y E) y el campo NWUEj que indica el número de UEs que se encuentran en estado WAITING.

6.3 Realización de las operaciones en la MAM

En esta sección se describe la realización, mediante la Máquina Abstracta de Multipath (MAM), de las operaciones definidas en el modelo de ejecución.

6.3.1 Realización de Selección de Objetivo

Es la primera operación que se define a nivel lógico en el Modelo de Ejecución de Multipath, y consiste en seleccionar el primer objetivo de OPA.

Teniendo en cuenta el Modelo Arquitectónico, en el momento de iniciar esta operación, el registro P contiene la dirección de la memoria CODE donde se almacenan las instrucciones que representan este objetivo. La selección del objetivo se consigue ejecutando dichas instrucciones. En concreto, las instrucciones de tipo PUT_I crean los términos Prolog correspondientes a los argumentos del objetivo y permiten referenciarlos a través de los registros Xi. Por otro lado, la instrucción CALL (o EXEC) determina la dirección del punto de entrada asociado al objetivo. La información referente al atributo de indeterminismo está incluida en el código de operación de esta última instrucción. Este atributo se almacena en el registro ATTR.

La ejecución de la instrucción CALL también modifica el registro CP para que contenga la dirección de continuación del objetivo, establece el número de argumentos del procedimiento (que se almacena en el registro NA), y determina el tamaño actual del entorno, que no se guarda en ningún registro al poder obtenerse directamente de la memoria CODE a través del registro CP. Si la instrucción de activación del punto de entrada corresponde a EXEC se realizan las mismas acciones excepto la modificación del registro CP.

El paralelismo de caminos que exhibe Multipath ya es posible explotarlo en la MAM durante esta primera operación cuando se debe crear una variable múltiple. Existen instrucciones que provocan la creación incondicional de estas variables (PUT_CREATE-VRM o CREATE-ARG_CREATE-VRM, según la variable corresponda a un argumento de un objetivo o pertenezca a un elemento de un término estructurado, respectivamente) e instrucciones de creación condicional (PUT_CREATE-VRX o CREATE-ARG_CREATE-VRX).

Este último tipo de instrucciones consultan un bit de condición (TF) que indica el tipo que debe tener la variable (simple o múltiple). Este bit de condición se activa mediante la instrucción SET-S/M que evalúa dicho tipo en función de las dependencias que posee la variable. El criterio que se aplica es el siguiente:

- Si una variable depende del tipo de exploración que se va a realizar del próximo objetivo, se comprueba si el número de UEs en estado AVAILABLE es menor que el número de UEs en estado CURRENT. El cumplimiento de esta condición garantiza que el objetivo va a ser explorado en profundidad y, por tanto, la variable puede ser simple.
- Si una variable depende del tipo que posee otra u otras variables, la variable a crear es simple si se recorre un único camino del árbol de búsqueda. Aunque una variable múltiple no puede pasar a simple una vez ya está creada, todas las variables nuevas, que se vinculan a variables múltiples con un único vínculo local, sí pueden crearse simples ya que estas unificaciones se gestionan con comandos especiales.

Cuando se encuentra una instrucción de creación de una variable múltiple, el ME envía un comando UE-INI-VRM a los UEs en estado CURRENT, que ocasiona la inicialización de una variable múltiple libre en su memoria HEAP_UE correspondiente. El ME no necesita ningún resultado por parte de los UEs tras la ejecución de este comando.

6.3.2 Realización de Selección de Cláusula Inicial

La Selección de la Cláusula Inicial consta de dos suboperaciones independientes. En primer lugar, la determinación de la lista de cláusulas susceptibles de unificar (función de indexación) y, posteriormente, la selección de la primera cláusula perteneciente a esta lista. Las instrucciones que intervienen en esta operación son las de INDEXACIÓN (SWITCH-ON-TERM, SWITCH-ON-CONSTANT y SWITCH-ON-STRUCTURE) y la instrucción TRY (o su simétrica TRY-ME-ELSE).

La dirección de la lista de cláusulas coincide con la dirección del punto de entrada si no

```
INDEXACIÓN (Xi, Lv, Ll, Ltc, Lts) retorna @/FAIL {
        Td ← deref(Xi);
        opción tag(Td) {
        <tvrs>:  si Lv = FAIL entonces comando(UE-FAIL); fsi;
                 retorna Lv;
        <tlst>:  si Ll = FAIL entonces
                          comando(UE-FAIL);
                 sino
                          mode ← READ; binding ← SINGLE; S ← val(Td);
                 fsi;
                 retorna Ll;
        <tcon>:  si ∃ Ti ∈ Ltc | match(Td, Ti) entonces
                          retorna @(Ti)
                 sino
                          comando(UE-FAIL);
                          retorna FAIL;
                 fsi;
        <tstr>:  si ∃ Ti ∈ Lts | match(val(Td), Ti) entonces
                          mode ← READ; binding ← SINGLE; S ← val(Td)+1;
                          retorna @(Ti)
                 sino
                          comando(UE-FAIL);
                          retorna FAIL;
                 fsi;
        <tvrm>:  NewIp ← preallocate_ip();
                 comando(UE-SWITCH(Td,Ns,NewIp));
                 Lista ← genera_lista_de_términos_diferentes();
                 NumTerm ← elem(Lista);
                 si NumTerm = 0 entonces
                          retorna FAIL;
                 sino
                          si NumTerm > 1 entonces
                                   crea_ip(cdr(Lista));
                                   crea_bg();
                          fsi;
                          X ← car(Lista);
                          comando(UE-W2C(NewIP, Term)); fsi;
                          si estructura(X) entonces
                                   calcular(mode, binding, S);
                          fsi;
                          retorna @(X)
                 fsi
        fopción
}
```

Algoritmo 6.1: *Indexación.*

contiene instrucciones de INDEXACIÓN. En caso de realizar indexación, el comportamiento de sus instrucciones es el siguiente (ver algoritmo 6.1).

En primer lugar se derreferencia el argumento indicado en la instrucción SWITCH-ON-TERM. En caso de no corresponder a una variable múltiple, se obtiene la dirección de la lista de cláusulas asociada al término obtenido o se produce un fracaso en la ejecución, en función

de si el término pertenece a la lista de tipos unificables o no.

En caso que la derreferenciación obtenga una variable múltiple, el ME necesita conocer el número de vínculos distintos que hay en todos los caminos y el número de caminos en los cuales se obtiene el mismo vínculo para el argumento indexado. Para ello, envía un comando UE-SWITCH a los UEs en estado CURRENT. Este comando puede ejecutarse en paralelo pero el ME necesita sincronizarse con su finalización. Por ello, el ME pasa a estado SUSPENDED y permanece en él mientras no hayan acabado todos los UEs. El comando UE-SWITCH ordena para cada UE que obtenga la derreferenciación de la variable múltiple, comprueba si es un término unificable y cambie de estado según las siguientes condiciones. Si el término no es permitido, ocasiona un fracaso local: el UE pasa a estado FAILED, si posee algún punto de selección, o pasa a estado AVAILABLE, en caso contrario. Si el término es permitido, el UE pasa a estado WAITING, en espera de que el ME decida pasar a explorar la alternativa asociado a dicho término.

Cuando finaliza el comando, el ME vuelve a estado RUNNING y evalúa el número de términos (o alternativas) distintos. Si el número de alternativas es 0, se produce un fracaso; si existe una alternativa, se obtiene la dirección de la lista de cláusulas almacenada en una instrucción de INDEXACIÓN. Si hay más de una alternativa, se crea un punto de indexación (ip), ordenando los términos según el orden: variable, lista, constantes y estructuras. Se obtiene el primer término según este orden y a continuación se envía un comando UE-W2C, que provoca el cambio de estado WAITING a CURRENT en los UEs correspondientes.

Una vez realizada la indexación se debe determinar la dirección de la primera cláusula. La dirección de esta cláusula coincide con la dirección de la lista de cláusulas si no existe la instrucción TRY, es decir, cuando sólo hay una cláusula candidata. En caso de haber más de una cláusula candidata, el comportamiento de la instrucción TRY (o TRY-ME) es el siguiente: crea un punto de selección (cp) en la memoria STACK; envía el comando UE-CREATE-CP, que no requiere sincronización; comprueba si debe crearse un nuevo objetivo en anchura (bg) en la memoria SBG y, por último, obtiene la dirección de la cláusula (en caso de la instrucción TRY es su parámetro y en caso de TRY-ME es la dirección de la siguiente instrucción).

El comando UE-CREATE-CP tiene como objetivo crear un punto de selección local (estructura de datos cp_ue) en la memoria CTRL_UE de cada UE en estado CURRENT. La existencia de un punto de selección local en un UE posibilita que el ME envíe, posteriormente, un comando a dicho UE para que restaure el contenido de su Entorno de Vínculos Local con el fin de explorar otra alternativa del punto de selección.

Por último, se indican las causas que provocan la creación de un nuevo objetivo en anchura (bg) durante esta operación de Selección de Cláusula Inicial:

- Siempre que se haya creado un punto de indexación.
- Cuando se crea un punto de selección con atributo NONDET y el número de UEs en estado AVAILABLE es mayor o igual que el número de caminos actuales.
- Cuando se crea un punto de selección con atributo SEMIDET y el número de caminos actuales es mayor que uno.

La segunda causa restringe la exploración en anchura en el momento de empezar la ejecución de un objetivo indeterminista si no existen recursos suficientes (número de UEs en estado AVAILABLE en este momento) que permitan obtener un mínimo de dos soluciones por camino de entrada al objetivo. La tercera causa evita la exploración en anchura de un objetivo determinista si sólo se recorre un camino.

Por otra parte, un objetivo en anchura (bg) posee el campo TYPE que proporciona información relativa a la posibilidad o no de explorar en anchura cualquier alternativa de un punto de selección que pertenezca al ámbito de dicho objetivo en anchura. Este campo puede tener dos valores:

- ONLY_IP: El objetivo en anchura se ha creado por la existencia de un punto de indexación pero la exploración de las alternativas de sus puntos de selección debe realizarse en profundidad.
- ALL_CP/IP: Valor que posee este campo en cualquier otra situación.

El valor ONLY_IP permite la exploración en anchura de las alternativas de un punto de indexación pero no de los puntos de selección que se creen durante la satisfacción del objetivo. Debe tenerse en cuenta que el tipo dinámico de las variables se determina únicamente a partir de los puntos de selección a los que se recomienda una exploración en anchura. Si se explorara en anchura un punto de selección al que se recomienda su exploración en profundidad, podría obtenerse un tipo dinámico de las variables inconsistente en el momento de su vinculación. Por su parte, el valor ALL_CP/IP permite la exploración en anchura de cualquier alternativa, pertenezca a un punto de indexación o de selección.

6.3.3 Realización de la Unificación

Una vez concluida la operación de determinación de la cláusula candidata a resolver un objetivo,

se realiza la unificación entre el objetivo y la cabecera de dicha cláusula. En este momento, los registros Xi permiten acceder a los argumentos que posee el objetivo a resolver (argumentos formales) y el registro P apunta a instrucciones de tipo UNF que especifican los argumentos de la cabecera de la cláusula (argumentos reales).

La función de este apartado es describir el comportamiento de estas instrucciones, considerando en primer lugar la realización del proceso de unificación general; posteriormente, la realización de la unificación cuando se conoce el tipo de uno de los dos términos; y finalmente, la realización de optimizaciones en la unificación por conocer información calculada durante la interpretación abstracta del programa.

6.3.3.1 Unificación general

Las instrucciones UNF_I_{IX/Y} realizan la unificación general entre dos términos cualesquiera. En todas estas instrucciones, el primer operando (I) simboliza el argumento formal accesible a partir de un registro Xi. Las dos posibilidades que existen para el segundo operando simbolizan otro argumento formal o una variable temporal (IX), o una variable permanente (Y).

El algoritmo iterativo utilizado en la realización de una unificación general utiliza una zona de memoria, denominada PDL, que es gestionada en forma de pila y contiene los términos que faltan por unificar (ver algoritmo 6.2).

La unificación en Multipath no realiza la comprobación denominada *occurs check*, consistente en verificar que no se trata de unificar una variable libre con un término estructurado en el cual aparezca la misma variable. En este situación (poco frecuente), surge una recurrencia que ocasionaría un bucle en el momento de derreferenciar la variable. Pocos sistemas gestionan esta comprobación por la penalización que supone, y por ello, recae en el programador la responsabilidad de asegurar la no existencia de este tipo de unificaciones.

El ME es el motor responsable de iniciar la unificación, derreferenciando los dos términos de partida para obtener los vínculos que poseen. Cuando los términos a unificar corresponden a dos variables múltiples o a una variable múltiple y un término instanciado, y el número de caminos recorridos es mayor que uno, el ME envía un comando de unificación para que los UEs acaben de realizar la unificación de forma local a cada camino. En cualquier otro caso, el ME efectúa la unificación de forma completa. Esto incluye el caso en que se unifica una variable simple libre con los términos asociados a una variable múltiple, en el que el ME simplemente vincula la variable simple a la variable múltiple.

Los vínculos de variable simples obtenidos durante la unificación se almacenan en aquella

```
UNIFICACIÓN (T1, T2) retorna BOOLEANO {
        push(PDL, T1); push(PDL, T2);
        mientras no vacío(PDL) y NCUE > 0 hacer
                Td1 ← deref(pop(PDL));
                si tag(Td1) = <tvrm> y NCUE=1 entonces comando(UE-DEREF-1(Td1)); fsi;
                Td2 ← deref(pop(PDL));
                si tag(Td1) = <tvrm> y NCUE=1 entonces comando(UE-DEREF-1(Td2)); fsi;
                si tag(Td1) = <tvrv> o tag(Td2) = <tvrv> entonces /* no hacer nada */
                sino si tag(Td1) = <tvrs> o tag(Td2) = <tvrs> entonces
                        bind(Td1,Td2);
                sino si tag(Td1) = <tvrm> o tag(Td2) = <tvrm> entonces
                        si tag(Td1)=<tcon> o tag(Td2) = <tcon> entonces
                                comando(UE-UNF_T_CON(Td1,Td2));
                        sino
                                comando(UE-UNF_T_T(Td1,Td2));
                        fsi; fsi; fsi;
                sino si tag(Td1) = tag(Td2) entonces
                        si tag(Td1) = <tcon> entonces
                                si Td1 <> Td2 entonces
                                        comando(UE-FAIL);
                                fsi;
                        sino si tag(Td1) = <tlst> entonces
                                push(PDL, load(value(Td1)); push(PDL, load(value(Td2));
                                push(PDL, load(value(Td1)+1); push(PDL, load(value(Td2)+1));
                        sino si tag(Td1) = <tstr> entonces
                                si load(value(Td1) <> load(value(Td2)) entonces
                                        comando(UE-FAIL);
                                sino
                                        ∀ i:1..aridad
                                                push(PDL, load(value(Td1)+i));
                                                push(PDL, load(value(Td2)+i));
                                fsi;
                        fsi; fsi; fsi;
                fsi; fsi; fsi; fsi;
        fmientras;
        retorna (NCUE > 0);
}
```

Algoritmo 6.2: *Unificación general realizada por el ME.*

variable simple a la que esté vinculada que aún no almacene ningún vínculo. En caso de vincular dos variables simples libres, el vínculo se almacena en la variable que posea una dirección mayor. Estas variables están almacenadas en la memoria HEAP o STACK.

Un aspecto importante en esta operación de unificación es la explotación de paralelismo de caminos. El modelo de ejecución permite la ejecución paralela de todas aquellas unificaciones en que aparezcan variables múltiples. En la MAM, la ejecución concurrente es posible en el momento en que el ME envía un comando de unificación a los UEs en estado CURRENT. La unificación local puede realizarse en paralelo en los distintos UEs.

```
UNIFICACIÓN_UE (T1, T2) retorna BOOLEANO {
        push(PDL_UE, T1); push(PDL_UE, T2);
        mientras no vacío(PDL_UE) hacer
                Td1 ← deref_ue(pop(PDL_UE));
                Td2 ← deref_ue(pop(PDL_UE));
                si tag(Td1) = <tvrv> o tag(Td2) = <tvrv> entonces continuar; fsi;
                si tag(Td1) = <tvrm> o tag(Td2) = <tvrm> entonces
                        bind_ue(Td1,Td2);
                sino si tag(Td1) = tag(Td2) entonces
                        si tag(Td1) = <tcon> entonces
                                si Td1 <> Td2 entonces
                                        retorna FALSO;
                                fsi;
                        sino si tag(Td1) = <tlst> entonces
                                push(PDL_UE, load(value(Td1));
                                push(PDL_UE, load(value(Td2));
                                push(PDL_UE, load(value(Td1)+1);
                                push(PDL_UE, load(value(Td2)+1));
                        sino si tag(Td1) = <tstr> entonces
                                si load(value(Td1) <> load(value(Td2)) entonces
                                        retorna FALSO;
                                sino
                                        ∀ i:1..aridad
                                                push(PDL_UE, load(value(Td1)+i));
                                                push(PDL_UE, load(value(Td2)+i));
                                fsi;
                        fsi; fsi; fsi;
                sino retorna FALSO
                fsi; fsi;
        fmientras;
        retorna CIERTO ;
}
```

Algoritmo 6.3: *Unificación general realizada por un UE.*

Los comandos que el ME puede enviar a los UEs en estado CURRENT son:

- UE-UNF_T_T, cuando se requiere realizar la unificación general de dos variables múltiples de forma local en cada camino, o bien entre una variable múltiple y un término estructurado. El algoritmo 6.3 muestra la unificación general que se realiza de forma local a cada camino en un UE.

- UE-UNF_T_CON, cuando se debe unificar una variable múltiple con un término constante.

  En estos dos comandos, el ME necesita conocer el resultado de la unificación: si ha tenido éxito (el UE correspondiente permanece en estado CURRENT) o si fracasa la unificación local (el UE pasa a estado FAILED si tiene algún cp activo o a AVAILABLE en caso contrario)

- UE-DEREF-1, cuando sólo se explora un camino del árbol de búsqueda y se quiere unificar una variable múltiple.
  En este caso, el ME necesita obtener el vínculo de la variable múltiple. Después de obtenerlo, el ME prosigue con la unificación. Este comportamiento es necesario por la optimización realizada en la determinación del tipo dinámico de las variables que dependen del tipo de otras variables. Cuando el número de caminos recorridos es uno (NCUE=1), estas variables se crean como simples. Esto obliga a hacer un tratamiento especial de las variables múltiples por parte del ME cuando NCUE=1 para evitar que ninguna variable múltiple deba tener que vincularse a otra variable simple. Esta circunstancia está prohibida en la MAM.

- UE-FAIL, cuando el ME detecta un fracaso global durante la unificación.
  Este comando tiene la misión de forzar un fracaso en cada uno de los caminos actuales. El ME necesita saber cuantos UEs pasan a estado AVAILABLE y cuantos a estado FAILED.

Los vínculos de variables múltiples obtenidos durante la unificación local en un camino se almacenan en la memoria HEAP_UE asociada a ese camino. En este caso, los vínculos de variables libres siempre se almacenan en aquella variable que posea una dirección más alta.

La unificación finaliza con éxito si al finalizar el proceso existe como mínimo un UE en estado CURRENT.

6.3.3.2 Casos particulares de unificación

No se utiliza el algoritmo general de unificación cuando un argumento formal de la cabecera de la cláusula corresponde a un término no variable o a una variable que aún no está creada. En este caso, la interpretación abstracta ha generado instrucciones específicas para tratar cada uno de los posibles casos (ver sección 5.8.4 dedicada a la compilación de la cabecera de una cláusula).

A continuación se describe el comportamiento de las instrucciones que realizan la unificación de un argumento real con una variable que debe crearse, una constante, una lista o una estructura.

Las instrucciones UNF_I_CREATE-VAR-{X/Y} realizan la unificación de un argumento real (I) con una variable, que se debe crear, que puede ser temporal (X) o permanente (Y). Esta unificación siempre tiene éxito y se crea una variable simple que contiene un vínculo hacia el otro término a unificar, independientemente del tipo que posea.

La instrucción UNF_I_CON realiza la unificación de un argumento real (I) con un término constante (CON) perteneciente a la cabecera de la cláusula. Esta instrucción vincula el argumento a la constante, si el argumento es una variable libre; o comprueba si los dos términos coinciden, si el argumento no es una variable libre.

La unificación de un término estructurado se realiza con instrucciones específicas para comprobar el tipo de datos del término a unificar e instrucciones para realizar la unificación de todos los elementos que forman el término estructurado de la cabecera de la cláusula. Estas instrucciones siempre deben ser ejecutadas en orden secuencial.

Las instrucciones UNF_IX_REF-LST o UNF_IX_REF-STR-FTR (según el argumento de la cabecera sea una lista o una estructura) realizan la comprobación del tipo de datos del argumento real. Cuando el término es una estructura, adicionalmente se comprueba que los dos functors coincidan. Se envían comandos a los UEs cuando el argumento real derreferencia a una variable múltiple. Por otra parte, estas instrucciones son las encargadas de inicializar 3 registros que son utilizados en las instrucciones posteriores de unificación de los elementos. Estos registros son los siguientes:

Registro BINDING:

Tiene la función de indicar el número de vínculos visibles que posee el argumento real. Contiene el valor SINGLE cuando el argumento real posee un único vínculo visible en todos los camino recorridos y el valor MULTIPLE cuando posee un vínculo distinto en cada uno de los caminos.

Registro MODE:

Tiene la función de indicar el modo de unificación del término estructurado. Contiene el valor READ cuando el argumento real corresponde a un término estructurado ya creado y el valor WRITE cuando el argumento real corresponde a una variable libre. En este último caso, la unificación debe crear los distintos elementos en la memoria HEAP.

Registro S:

Tiene la función de indicar la dirección de la memoria HEAP en que se almacena el siguiente elemento del término estructurado. Este registro tiene significado cuando el modo de unificación es READ.

Cuando el registro BINDING contiene el valor MULTIPLE, los UEs son los encargados de realizar la unificación de cada uno de los vínculos locales. En este sentido, cada UE posee dos registros: MODE_UE, que identifica el modo de unificación en el camino asociado; y S_UE,

que contiene la dirección de la memoria HEAP asociada al siguiente elemento a unificar.

Las instrucciones que realizan la unificación de los elementos del término estructurado pertenecen al tipo UNF-ARG y son las siguientes:

UNF/CREATE-ARG_{IX/Y/UNS-IX/UNS-Y}:

Realizan la unificación del elemento correspondiente del término estructurado con una variable. Esta variable puede ser otro argumento de la cabecera o una variable temporal (IX), una variable permanente (Y), o bien, los dos casos anteriores cuando la variable tiene posibilidades de ser insegura (UNS-IX o UNS-Y). Recuérdese que una variable es insegura cuando es libre y reside en un entorno (env) de la memoria STACK que es eliminado por motivo de la optimización de recursividad de cola. Para evitar referencias a zonas de memoria eliminadas, las instrucciones UNS-{IX/Y} no permiten la existencia de vínculos de la memoria HEAP a la memoria STACK.

UNF/CREATE-ARG_CREATE-VR[S/M/X/V]-{X/Y/A}

Realizan la unificación del elemento correspondiente con una variable que debe crearse. Esta variable puede ser temporal (X), permanente (Y), o anónima (A). En este último caso, el parámetro permite especificar una constante que indica el número de variables anónimas que deben crearse.

Si el modo de unificación es READ, el tipo dinámico de la variable a crear viene determinado por el número de vínculos visibles en el elemento del argumento real. Si el modo de unificación es WRITE, el tipo de la variable viene indicado en el propio código de operación.

UNF/CREATE-ARG_CNT:

Realiza la unificación de un elemento del argumento real con una constante.

Todas estas instrucciones acceden a los registros MODE, BINDING y S. Se envían comandos a los UEs en estado CURRENT para completar la unificación cuando el registro BINDING contiene el valor MULTIPLE o bien, en caso contrario, cuando al derreferenciar un término se obtiene una variable múltiple. Obsérvese que cuando un elemento del término estructurado es otro término estructurado, este último debe crearse antes de la unificación el elemento estructurado. Posteriormente, se utiliza una instrucción UNF/CREATE-ARG_IX para unificar dicho elemento a partir de una variable temporal que permite acceder a su valor.

6.3.3.3 Optimizaciones adicionales

A las instrucciones de unificación descritas anteriormente se les puede añadir información calculada en la interpretación abstracta. El comportamiento de estas instrucciones se describe a continuación.

UNF_I-TV_CON:

El argumento real corresponde a una variable libre (I-TV). En este caso, la unificación no puede fracasar nunca.

UNF_I-{TNV/TL/TV}_REF-LST:

El argumento real corresponde a un término no variable (I-TNV); a una lista (I-TL); o a una variable libre (I_TV). En el primer caso, el modo de unificación siempre será READ. En el segundo caso, la unificación siempre tiene éxito. En el último caso, el modo de unificación es WRITE y siempre tiene éxito la unificación.

UNF_I-{TNV/TV}_REF-STR-FTR:

El argumento real corresponde a un término no variable (I-TNV), implicando que el modo es READ; o bien, corresponde a una variable libre (I-TV), implicando modo WRITE y que la unificación no fracasa nunca.

Por otra parte, las instrucciones de unificación de elementos estructurados también admiten la posibilidad de considerar un modo de unificación establecido de forma estática por la interpretación abstracta. La unificación de elementos en modo READ se consigue con las instrucciones UNF-ARG. La unificación de elementos en modo WRITE se consigue con instrucciones del tipo CREATE-ARG. Las características principales son:

- No necesitan consultar el registro MODE
- Se añaden instrucciones de unificación del último elemento de un término estructurado con otros término estructurado. Estas instrucciones permiten unificar secuencialmente términos estructurados anidados.

Todas instrucciones también envían los comandos oportunos a los UEs en estado CURRENT siguiendo el mismo criterio que el utilizado en la unificación general.

6.3.4 Realización de Inferencia

El Modelo de Ejecución establece que esta operación debe añadir en EAV los vínculos obtenidos en la Unificación y substituir el objetivo que ha unificado con éxito por los objetivos del cuerpo

de la cláusula.

En el Modelo Arquitectónico, esta operación se realiza incluida en las operaciones previas. En concreto, los vínculos se añaden incrementalmente durante la unificación. Por otro lado, la substitución del objetivo por el cuerpo de la cláusula se consigue en el momento que finalizan las instrucciones UNF ya que el registro P pasa a apuntar al primer objetivo del cuerpo. Además, el registro CP apunta a la dirección de continuación, acción realizada durante la ejecución de la instrucción CALL en la Selección de Objetivo.

Si durante la Unificación se ejecuta la instrucción ALLOCATE, la dirección de continuación también se almacena en el entorno (env) actual. De esta forma, al finalizar la cláusula se podrá restaurar esta dirección en caso que se modifique por la ejecución de alguna instrucción CALL durante la selección de un objetivo del cuerpo de esta cláusula.

6.3.5 Realización de Solución?

Según el Modelo de Ejecución esta función determina si se ha solucionado un objetivo en anchura, comprobando si OPA es una lista vacía después de haber realizado una inferencia.

En el Modelo Arquitectónico, la posibilidad de encontrar una solución se detecta al finalizar la ejecución de los objetivos del cuerpo de una cláusula mediante la instrucción PROCEED. Esta instrucción sólo aparece en las cláusulas que representan hechos (no contienen objetivos en su cuerpo), después de haber realizado la unificación de la cabecera.

La ejecución de la instrucción PROCEED únicamente modifica el registro P a partir del contenido del registro CP para que pase a apuntar a la dirección de la memoria CODE del primer objetivo de la lista de continuación.

Para detectar si OPA es una lista vacía se comprueba si los registros CP y E, que identifican la lista de continuación del programa contienen las mismas direcciones que los campos CP y E de la estructura bg asociado al objetivo en anchura actual, que identifica la lista de continuación de dicho objetivo (algoritmo 6.4).

En caso de haber encontrado una solución, el Modelo de Ejecución establece que los

```
SOLUCIÓN? () retorna BOOLEANO {
        retorna( (CBG->CP = CP) y (CBG->E = E) )
}
```

Algoritmo 6.4: *Condición de Solución.*

Entornos Actuales de Vínculos (EAV) pasan a ser Entornos de Vínculos (EV) correspondientes a soluciones del objetivo en anchura. En el modelo arquitectónico, esta operación se puede realizar mediante un comando de cambio de estado de forma que los UEs en estado CURRENT pasen a estado SOLUTION. Sin embargo, esta operación no se efectúa en este momento sino que el envío del comando se retrasa hasta las operaciones de Avance o de Retroceso. De esta forma, se ahorran cambios de estado innecesarios si un UE que ha encontrado una solución es requerido a volver a cambiar de estado posteriormente durante la operación de Avance.

6.3.6 Realización de Exploración en Anchura?

Esta función determina, tras encontrar una solución a un objetivo en anchura, si es posible realizar una exploración en anchura del árbol de búsqueda tal como determina el modelo de ejecución. El algoritmo 6.5 refleja estas condiciones, que son detalladas a continuación.

En primer lugar, se almacena en el objetivo en anchura actual (apuntado por CBG) las cimas de las memorias HEAP, HEAP_UE y TRAIL. Estos valores son necesarios para poder realizar posteriormente la recolección de basura.

Posteriormente, se consulta el cp o ip más joven existente en la memoria STACK. Si la creación de este punto fue anterior a la creación del objetivo en anchura del cual se ha encontrado una solución, no se puede realizar la exploración en anchura. Esta condición consiste en comparar el campo BG del punto de selección o indexación con el registro CBG. Los valores que contienen son las direcciones base de las estructuras bg asociadas al objetivo en anchura de la alternativa y al objetivo en anchura de la solución. Las estructuras bg asociadas a objetivos más jovenes siempre se crean en direcciones más altas.

En caso que la alternativa pueda conducir a una nueva solución del mismo objetivo en

```
EXPLORACIÓN_EN_ANCHURA () retorna BOOLEANO {
        CBG->H ← H;
        CBG->HUE ← max(CBG->HUE, HUE);
        CBG->TR ← TR;
        si B = NIL retorna( FALSO ) fsi;
        si B->CBG < CBG retorna( FALSO ) fsi;
        si B.TYPE = CP y B.CBG.TYPE = ALL_CP/IP y NAUE >= B.NCUE-B.NFUE
                retorna( CIERTO )
        fsi;
        si IP <> NIL entonces retorna( CIERTO ) fsi;
        retorna( FALSO );
}
```

Algoritmo 6.5: *Condición de Exploración en Anchura?.*

anchura, es preciso diferenciar la posibilidad que esta alternativa pertenezca a un cp o a un ip. Si pertenece a un ip siempre se permite la exploración en anchura. Si pertenece a un cp, se permite si su objetivo asociado es de tipo ALL_CP/IP y existen como mínimo el mismo número de UEs disponibles (en estado AVAILABLE) como UEs existentes en el momento de creación del cp (campo B.NCUE) menos los UEs que han fracasado desde la creación hasta el momento actual (campo B.NFUE). Por último, es posible también la exploración en anchura del punto de indexación (ip) más joven, aunque existan puntos de selección más jovenes que éste.

6.3.7 Realización de Avance

La operación de Avance se realiza cuando se determina que, tras la obtención de una solución de un objetivo en anchura, no es posible realizar una exploración en anchura.

En esta operación (ver algoritmo 6.6), el ME envía un comando UE-S2C, que ocasiona que todos los UEs que estaban en estado SOLUTION pasen a estado CURRENT, para proseguir, junto con los UEs que habían encontrado la solución, la exploración en profundidad de la continuación del objetivo en anchura. Este comando no precisa sincronización.

Posteriormente, se modifican los registros P y E para que apunten a la continuación del objetivo; también se actualizan los registros H y HUE para que pasen a apuntar a la primera posición libre de las memorias HEAP y HEAP_UE; se eliminan todos los objetivos en anchura de SBG sin posibilidad de encontrar más soluciones (recolección de basura), y se actualiza CBG para que apunte al siguiente objetivo en anchura.

6.3.8 Realización de Exploración en Profundidad?

La condición que determina si se realiza una exploración en profundidad se evalúa tras obtener un fracaso en una unificación.

```
AVANCE () retorna NULO {
        comando( UE-S2C );
        P ← CBG->P;
        E ← CBG->E:
        H ← CBG->H;
        HUE ← CBG->HUE;
        CBG->FLAG ← FALSO;
        si B <> NIL y B->CBG < CBG entonces TBG ← CBG fsi;
        CBG ← CBG->NBG;
}
```

Algoritmo 6.6: *Avance.*

```
EXPLORACIÓN_EN_PROFUNDIDAD? () retorna BOOLEANO {
        YBG ← bg in SBG | bg es el más joven con soluciones;
        si B = NIL y YBG = NIL entonces FIN_PROGRAMA fsi;
        si YBG <> NIL entonces
                H ← YBG->H;
                untrail(YBG->TR);
        sino
                H ← B->H;
                untrail(B->TR);
        fsi;
        si B = NIL retorna( FALSO ); fsi;
        si B->CBG < YBG retorna( CIERTO ); fsi;
        si B.TYPE=CP y B->CBG->TYPE=ALL_CP/IP y NAUE >= B->NCUE-B.NFUE
                entonces retorna( FALSO )
        fsi;
        si IP <> NIL entonces retorna( FALSO ) fsi;
        retorna( CIERTO );
}
```

Algoritmo 6.7: *Condición de Exploración en Profundidad?.*

Las acciones que se definen en el modelo arquitectónico en la evaluación de esta condición se muestran en el algoritmo 6.7. De forma resumida, pueden enumerarse de la siguiente manera.

En primer lugar, se busca en la memoria SBG el objetivo más joven con soluciones (denominado YBG en el algoritmo) y se realiza la operación de recolección de basura: todas aquellas zonas de memoria reservadas posteriormente a dicho objetivo YBG pueden ser recuperadas.

A partir de este momento, se actúa como en la operación de exploración en anchura: si la alternativa más joven de STACK está asociada a un objetivo más joven que YBG pasa a realizarse la operación de avance, es decir, se explora en profundidad. En caso contrario, si la alternativa corresponde a un ip o a un cp de tipo ALL_CP/IP con el número de UEs AVAILABLE necesario, puede ser explorada en anchura. También retrocede si existe alguna alternativa en la memoria STACK que corresponda a un ip. En caso que no se cumpla ninguna de estas alternativas, procede con la operación de avance.

6.3.9 Realización de Retroceso

Esta operación debe realizar el retroceso en el árbol de búsqueda. Corresponde a la operación de backtracking en el modelo convencional. En el Modelo de Ejecución de Multipath tiene como objetivo adicional permitir la exploración en anchura de otras ramas del árbol aun cuando no se ha llegado a las hojas del árbol de búsqueda.

La realización de esta operación exhibe un comportamiento distinto según la alternativa que se ha decidido explorar pertenezca a un punto de selección o a un punto de indexación (ver algoritmo 6.8).

Si la alternativa elegida corresponde a un punto de selección (cp), las acciones se resumen en enviar un comando UE-BACKTRACK, un comando UE-BACKWARD-CURRENT, un comando UE-BACKWARD-SOLUTION y, finalmente, restaurar los registros necesarios del cp.

El comando UE-BACKTRACK va dirigido a los UEs en estado FAILED y tiene como objetivo que los UEs deshagan los vínculos realizados desde la creación del punto de selección. Los UEs pasan a estado CURRENT ya que serán los responsables de recorrer la nueva alternativa.

El comando UE-BACKWARD-CURRENT va dirigido a los UEs en estado CURRENT. Ante este comando, un UE comprueba si debe recorrer la nueva alternativa o no, es decir, si tiene visible el cp hacia el que se retrocede. En caso negativo, el UE pasa a estado SOLUTION del objetivo en anchura actual. En caso positivo, reserva un UE en estado AVAILABLE. En este momento, se copia el Entorno de Vínculos hacia el nuevo UE y éste último pasa a estado SOLUTION del objetivo solucionado. El UE original realiza la operación de backtracking y pasa a estado CURRENT, como responsable de recorrer la alternativa elegida.

El comando UE-BACKWARD-SOLUTION va dirigido a los UEs en estado SOLUTION. Puede suceder que un UE en este estado deba recorrer la alternativa elegida. Esta posibilidad

```
RETROCESO (B) retorna NULO {
        si B->TYPE = CP entonces (* punto de selección *)
                comando( UE-BACKTRACK );
                si NCUE>0 entonces comando( UE-BACKWARD-CURRENT ) fsi;
                si CBG->FLAG entonces comando( UE-BACKWARD-SOLUTION ) fsi;
                restaurar de cp(Ai, P, E, CP, CBG, TBG, HUE)
        sino (* punto de indexación *)
                comando( UE-C2S );
                comando( UE-W2C );
                si ip interno entonces CBG->FLAG ← CIERTO
                sino restaurar TBG
                fsi;
                restaurar de ip( Ai, P, E, CP, CBG, HUE);
                si ultima alternativa entonces eliminar_ip(); fsi;
                selección de cláusula inicial
        fsi
}
```

Algoritmo 6.8: *Retroceso.*

es debida a que las alternativas de la memoria STACK no se exploran en orden LIFO: cualquier punto de indexación puede ser explorado aunque existan puntos de selección más jóvenes por explorar. En esta situación, pueden existir UEs que están en estado SOLUTION pero son responsables de explorar otras alternativas pendientes. En este caso, se procede como en el anterior comando. Se reserva un nuevo UE, se le copia el Entorno de Vínculos, el nuevo UE reemplaza al UE original como solución del objetivo, y el UE original realiza backtracking para pasar a explorar la nueva alternativa.

La siguiente operación a realizar después del Retroceso a un punto de selección es la Selección de Siguiente Cláusula.

Si la alternativa elegida corresponde a un punto de indexación (ip), las acciones se resumen en enviar un comando UE-C2S, un comando UE-W2C, y restaurar los registros pertinentes del punto de indexación.

El comando UE-C2S provoca que los UEs en estado CURRENT pasen a estado SOLUTION del objetivo solucionado. El comando UE-W2C va dirigido a los UEs en estado WAITING responsables de recorrer la siguiente alternativa del ip. Estos UEs pasa a estado CURRENT.

Posteriormente, el ME pasa a determinar la dirección donde está almacenada la primera cláusula de la alternativa elegida del ip. Puede suceder que el término tenga una única cláusula, con lo cual, la dirección está almacenada en el campo P del ip; o bien, puede tener más de una cláusula candidata, con lo que la dirección del campo P apunta a una instrucción TRY. Esta instrucción crea un nuevo cp en la memoria STACK y establece la dirección de inicio de la primera cláusula.

La siguiente operación a realizar después del Retroceso a un punto de indexación es la Unificación.

6.3.10 Realización de Selección de Siguiente Cláusula

El campo BP del punto de selección más joven (accedido mediante el registro B) apunta a la dirección de la siguiente cláusula. Esta cláusula está encabezada por una instrucción RETRY o TRUST.

La ejecución de la instrucción RETRY modifica el registro P para que apunte a la siguiente cláusula y el campo BP del cp para que contenga la dirección de la próxima cláusula.

La instrucción TRUST elimina el cp ya que la cláusula elegida es la última susceptible

de unificar. En este caso, se envía un comando UE-REMOVE-CP a los UEs en estado CURRENT para que también eliminen su punto de selección local (estructura de datos cp_ue).

6.4 Sincronización de los comandos

El Modelo Arquitectónico de Multipath establece que cualquier operación que deba realizarse a nivel local relacionada con un camino específico del árbol de búsqueda se indica a nivel de comandos. El ME es el responsable de enviar los comandos. A su vez, cada comando es recibido y ejecutado por los UEs que se encuentran en el estado requerido por el comando.

En este apartado se resumen en dos tablas todos los comandos MAM. La tabla 6.1 contiene los comandos que afectan a tareas de control de flujo del recorrido de un árbol de búsqueda, mientras que la tabla 6.2 agrupa a todos los comandos utilizados en la unificación de términos Prolog. Para cada comando se indica el código de operación, los argumentos, el estado de los UEs hacia los cuales va dirigido y el resultado que espera el ME.

El modelo arquitectónico deja abierta la posible realización de Multipath tanto en un sistema secuencial como en un sistema paralelo. En un sistema secuencial un comando se ejecuta en el mismo momento en que se detecta su necesidad, realizando la tarea que conlleva asociada de forma consecutiva para todos los UEs que se encuentran en el estado adecuado.

En el contexto de un sistema paralelo, el modelo arquitectónico ofrece la posibilidad de explotar el paralelismo de caminos definido en el modelo de ejecución ya que la tarea asociada a un comando puede ejecutarse de forma concurrente en todos los UEs que se encuentran en

CÓDIGO DE OPERACIÓN	ARGUMENTOS	ESTADO INICIAL	RESULTADOS
INDEXACIÓN			
ue-switch	Vrm, IdSw, IP	CURRENT	State, Term, Mode, S
INDETERMINISMO			
ue-create-cp	B	CURRENT	
ue-remove-cp		CURRENT	
ue-backtrack	B	FAILED(B)	
ue-backward-current	CBG, B	CURRENT	State
ue-backward-solution	CBG, B	SOLUTION(CBG)	State
CAMBIO DE ESTADO			
ue-s2c	CBG	SOLUTION(CBG)	
ue-c2s	CBG	CURRENT	
ue-w2c	IP, TERM	WAITING(IP,TERM)	

Tabla 6.1: *Comandos de CONTROL*

CÓDIGO DE OPERACIÓN	ARGUMENTOS	ESTADO INICIAL	RESULTADOS
ue-unf_t_t	Vrm, Vrm	CURRENT	State
ue-unf_t_con	Vrm, Con	CURRENT	State
ue-unf_tv_con	Vrm, Con	CURRENT	
ue-unf_t_ref-lst	Vrm, Lst	CURRENT	State, Mode
ue-unf_tl_ref-lst	Vrm	CURRENT	
ue-unf_tv_ref-lst	Vrm, Lst	CURRENT	
ue-unf-1_t_ref-lst	Vrm, Lst	CURRENT	State, Mode, S
ue-unf-1_tl_ref-lst	Vrm	CURRENT	S
ue-unf_t_ref-str-ftr	Vrm, Ftr, Str	CURRENT	State, Mode
ue-unf_tnv_ref-str-ftr	Vrm, Ftr	CURRENT	State
ue-unf_tv_ref-str-ftr	Vrm, Str	CURRENT	
ue-unf-1_t_ref-str-ftr	Vrm, Ftr, Str	CURRENT	State, Mode, S
ue-unf-1_tnv_ref-str-ftr	Vrm, Ftr	CURRENT	State, S
ue-unf-rw-arg_t	Term	CURRENT	State
ue-unf-rd-arg_t	Term	CURRENT	State
ue-unf-rw-arg_con	Con	CURRENT	State
ue-unf-rd-arg_con	Con	CURRENT	State
ue-unf-rd-arg-tv_con	Con	CURRENT	
ue-unf-rd-arg_ref-lst	Lst	CURRENT	State, Mode
ue-unf-rd-arg-tl_ref-lst		CURRENT	
ue-unf-rd-arg-tv_ref-lst	Lst	CURRENT	
ue-unf-rd-arg-1_ref-lst	Lst	CURRENT	State, Mode, S
ue-unf-rd-arg-1-tl_ref-lst		CURRENT	S
ue-unf-rd-arg_ref-str-ftr	Ftr, Str	CURRENT	State, Mode
ue-unf-rd-arg-tnv_ref-str-ftr	Ftr	CURRENT	State
ue-unf-rd-arg-tv_ref-str-ftr	Ftr, Str	CURRENT	
ue-unf-rd-arg-1_ref-str-ftr	Ftr, Str	CURRENT	State, Mode, S
ue-unf-rd-arg-1-tnv_ref-str-ftr	Ftr	CURRENT	State, S
ue-deref-1	Vrm	CURRENT	Term
ue-deref-rd-arg-1		CURRENT	Term
ue-deref-rw-arg-ini-vrm	aVrm	CURRENT	
ue-deref-rd-arg-ini-vrm	aVrm	CURRENT	
ue-ini-vrm	aVrm	CURRENT	
ue-fail		CURRENT	State

Tabla 6.2: *Comandos de UNIFICACIÓN*

el estado requerido. Es importante remarcar que las acciones a ejecutar en cada UE son independientes entre si ya que afectan únicamente a datos locales a cada camino del árbol de búsqueda.

A nivel de modelo arquitectónico, cuando el ME envía un comando pasa a estado SUS-

PENDED. Es importante establecer el mecanismo de sincronización de la finalización del comando, que permite pasar el ME otra vez a estado RUNNING. En este sentido, los comandos se clasifican en dos tipos:

- **Comandos WRITE.** Engloban a todos los comandos de los cuales el ME no necesita ningún resultado.
- **Comandos READ.** Engloban al resto de comandos en los cuales el ME necesita algún resultado.

La ventaja que poseen los comandos WRITE radica en que el ME no necesita esperarse a su finalización para proseguir con la siguiente instrucción. Una vez enviado el comando, el ME pasa directamente a estado RUNNING.

Sin embargo, en los comandos READ, el ME se sincroniza con la finalización de su ejecución en todos los UEs implicados. Posteriormente, procesa el resultado que provoca esta espera. Existen 4 tipos distintos de resultados: (i) STATE: indica el estado de un UE después de finalizar la ejecución del comando (el ME necesita actualizar el número total de UEs en cada estado); (ii) MODE: indica el modo de unificación de términos estructurados (el ME puede optimizar la unificación de términos estructurados cuando el modo es común a todos los UEs); (iii) S: indica la dirección del primer elemento de un término estructurado (el ME trata directamente las unificaciones cuando el número de caminos actuales es uno); y (iv) TERM: referencia a un término Prolog (el ME lo necesita en el comando UE-SWITCH y en algunos comandos que se envían cuando existe un único camino actual).

6.5 Recolección de basura

La recolección de basura es una operación importante en un sistema en que la mayor parte de los datos que maneja son creados dinámicamente. A continuación se resume las gestión llevada a cabo en diversas zonas de memoria de la MAM en el contexto de esta operación.

Memorias HEAP y HEAP_UE. El rango de direcciones potencialmente visibles en cada una de estas zonas de memoria se actualiza en cada fracaso de una unificación. El puntero a la cima de cada memoria se obtiene de la estructura de datos cp (en una operación de Retroceso) o en la estructura de datos bg (en una operación de Avance). No se realiza recolección de basura de aquellas partes de la memoria HEAP pertenecientes al rango de direcciones visibles que nunca pueden ser accedidas. En la memoria HEAP_UE también pueden existir zonas inaccesibles ya que al realizar la operación de Avance siempre se continúa con el máximo valor

que tenga el puntero a esta zona en todas las soluciones obtenidas.

Memorias STACK y CTRL_UE. Los estructuras de datos cp o ip almacenadas en la memoria STACK se eliminan cuando se empieza a ejecutar la última alternativa de un punto de selección o de un punto de indexación, respectivamente. Las estructuras de datos env se eliminan antes de ejecutar el último objetivo de una cláusula (optimización de recursividad de cola). El rango de direcciones visibles viene determinado por la estructura de datos (cp, ip o env) más joven que aún no se ha eliminado. Las estructuras de datos cp_ue almacenadas en las memorias CTRL_UE se eliminan durante la ejecución del comando UE-REMOVE-CP.

Memorias TRAIL y TRAIL_UE. El espacio ocupado por estas memorias se elimina durante la operación de Retroceso en el momento en que se deshacen los vínculos globales (memoria TRAIL) o locales a cada UE (memorias TRAIL_UE).

Memoria SBG. En la operación de Avance se eliminan todas aquellas estructuras de datos bg que no tienen soluciones y sin posibilidad de encontrar ninguna otra solución (el registro TBG se actualiza a partir del valor existente en B.CBG). Esta misma gestión también se realiza en la operación de Retroceso, cuando se ejecuta tras un fracaso en la Unificación.

6.6 Predicados predefinidos

Tanto el Modelo de Ejecución como el Modelo Arquitectónico de Multipath se han descrito teniendo en cuenta únicamente el comportamiento declarativo de los predicados de que consta un programa. Sin embargo, en el lenguaje Prolog se establecen un conjunto de predicados predefinidos (built-in) con el fin de que pueda ser utilizado, no tan sólo como un método automático de resolución de problemas considerados como demostración de teoremas, sino que también pueda utilizarse en la programación de cualquier tipo de algoritmos de resolución de problemas.

En líneas generales, la realización de predicados predefinidos sigue la misma idea que en un predicado declarativo. El ME realiza las acciones indicadas por la semántica del predicado siempre que los datos sean globales a todos los caminos. En caso de obtener alguna variable múltiple cuando se recorre más de un camino, se envía el comando adecuado a los UEs en estado CURRENT.

El operador de corte (!) no presenta ningún problema específico en el sistema Multipath ya que el orden de ejecución de las cláusulas de un procedimiento siempre coincide con el orden secuencial. A nivel arquitectónico, es necesario introducir un registro denominado B0,

cuya función es apuntar al primer punto de selección que no debe ser eliminado por un operador de corte. Este registro también existe en la WAM.

Los predicados assert/1 y retract/1 no son tratados por la complejidad que acarrea su realización y por no ser parte directamente relacionada con el objetivo que se pretende obtener con el sistema Multipath.

Otros predicados con efectos laterales (por ejemplo, read/1 o write/1) necesitan serializar el recorrido del árbol de búsqueda. Los procedimientos que contienen predicados de este tipo deben ser ejecutados explorando únicamente un camino del árbol de búsqueda. Para ello, se inserta un punto de indexación al iniciar el procedimiento que tiene tantas alternativas como caminos locales existen en ese momento. En cada una de las alternativas se recorre únicamente un camino.

6.7 Resumen y contribuciones

En este capítulo se ha descrito la fase de interpretación real del Modelo Arquitectónico de Multipath. Esta descripción se ha realizado para cada uno de los elementos que forman el estado de la ejecución y para cada operación que lo transforma, tal como define el Modelo de Ejecución de Multipath. El modelo arquitectónico define una forma de representar y almacenar los elementos del estado de la ejecución, así como la semántica de las instrucciones que permiten la realización de estas operaciones.

La contribución global que se aporta en esta fase del modelo arquitectónico es la definición completa de la Máquina Abstracta de Multipath (MAM). Los aspectos innovadores más importantes que presenta hacen referencia a la posibilidad de realizar una exploración parcial en anchura del árbol de búsqueda y de ejecutar de forma concurrente comandos que operan sobre datos locales a los caminos recorridos.

7

REALIZACIÓN SECUENCIAL DE MULTIPATH

El último nivel de descripción de Multipath consiste en la realización concreta del sistema. En este nivel se identifican los elementos hardware y software en los que se mapea el Modelo Arquitectónico de Multipath.

En primer lugar, es preciso indicar que la primera fase del modelo, correspondiente a la interpretación abstracta, debería llevarse a cabo totalmente por software. Sin embargo, hay que tener en cuenta que el objetivo principal de este trabajo es evaluar el rendimiento en tiempo de ejecución de la búsqueda parcial en anchura aplicada en algoritmos indeterministas. El desarrollo del software necesario para realizar la fase de interpretación abstracta constituye un objetivo secundario que se incluye dentro de los trabajos futuros por realizar. En este sentido, se ha optado por utilizar un compilador de Prolog a WAM [37] e introducir de forma manual las modificaciones necesarias para conseguir el código MAM. Las instrucciones MAM así obtenidas reflejan el resultado de aplicar las operaciones que se han definido en el modelo arquitectónico para esta fase de interpretación abstracta. Teniendo en cuenta estas premisas, la descripción que sigue a continuación afecta únicamente a la fase de interpretación real de un programa.

Una de las características más relevantes de la fase de interpretación real del modelo arquitectónico estriba en la flexibilidad de poder aplicarlo en diferentes sistemas ya existentes con un hardware de propósito general orientado tanto a la ejecución secuencial como paralela de las aplicaciones. Se ha optado por evaluar el comportamiento de Multipath en ambos sistemas. En este capítulo se describen los aspectos propios de la realización de la Máquina Abstracta de Multipath en un entorno secuencial y se remite al capítulo posterior la descripción de la realización paralela en un multiprocesador con memoria compartida.

7.1 Interpretación real en SMAM

La realización secuencial de la fase de interpretación real del Modelo Arquitectónico de Multipath se denomina SMAM. En este entorno secuencial existe únicamente una **Unidad de Proceso (PU)** que ejecuta las acciones asociadas en el modelo arquitectónico al Motor Principal (ME) y a los Motores de Unificación (UEs).

Inicialmente, la PU realiza las acciones asociadas al ME: búsqueda, decodificación y ejecución de las instrucciones MAM. Tal como define el modelo arquitectónico, durante la ejecución de una instrucción, el ME puede enviar un comando a los UEs con la finalidad de que realicen la operación pertinente, de forma local, en los caminos que se encuentren en el estado apropiado. En este entorno secuencial, enviar un comando significa que la PU pasa a comportarse como un UE, y ejecuta la acción asociada al comando para todos los UEs involucrados. Al finalizar el comando, la PU vuelve a comportarse como el ME, y pasa a ejecutar las acciones existentes tras el comando.

La ejecución de las acciones del ME y de los UEs se realiza mediante interpretación software. El código asociado a las instrucciones y a los comandos MAM se ha desarrollado en lenguaje C. La entrada del intérprete corresponde al código objeto de las instrucciones MAM y al parámetro NUM_UE, que especifica el número máximo de UEs que pueden ser utilizados, es decir, el número máximo de caminos visibles en un momento de la ejecución. El código objeto se obtiene a través de una etapa de ensamblaje y de montaje de los ficheros fuente que contienen el código MAM textual de los programas a ejecutar. La figura 7.1 muestra las etapas de que consta esta fase de interpretación real.

El valor de un registro asignado a un UE particular se obtiene a través de indexar un vector que contiene los valores de dicho registro para todos los UEs. La indexación se realiza a través de un identificador de UE cuyo valor está comprendido entre el rango [0, NUM_UE-1].

Con el fin de evitar la consulta del registro de estado en que se encuentra cada UE para

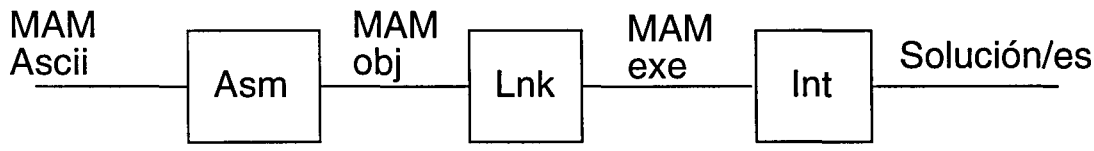


Figura 7.1: *Interpretación real llevada a cabo mediante una etapa de ensamblaje, montaje y ejecución.*

comprobar si debe ejecutar el comando o no, la PU dispone de unas estructuras de datos adicionales que indican los identificadores de UEs que se encuentran en cada posible estado. Con este objetivo, se definen los registros:

- VCUE: Dirección base de la zona de memoria donde se encuentran los identificadores de los UEs en estado CURRENT.
- VAUE: Identifica a los UEs en estado AVAILABLE.

y los siguientes campos:

- VFUE: Identifica a los UEs en estado FAILED sobre un cierto punto de selección, y se almacenan en la estructura de datos *cp* de la memoria STACK.
- VSUE: Identifica a los UEs en estado SOLUTION de un cierto objetivo en anchura, y se almacenan en la estructura de datos *bg* de la memoria SBG.
- VWUE: Identifica a los UEs en estado WAITING sobre una determinado grupo de caminos de un punto de indexación, y se almacenan en la estructura de datos *ip* de la memoria STACK.

7.2 Benchmarks

El conjunto de benchmarks que han sido analizados corresponden a programas que exhiben indeterminismo. Por tanto, existen acciones dentro del algoritmo que pueden resolverse de distintas formas y, además, la comprobación de si las soluciones locales obtenidas en estas acciones conducen a una solución global del programa no se realiza hasta un momento posterior de la ejecución.

La descripción particular de cada uno de estos benchmarks es la siguiente:

- BITSPAL: Encuentra todas las listas palindrómicas formadas por 20 elementos binarios

[110] (1024 soluciones).

- CUBE: Soluciona el puzzle Instant Insanity [115] (64 soluciones).
- Q10: Encuentra todas las maneras de colocar 10 reinas en un tablero de ajedrez de 10x10 sin que se maten entre ellas (724 soluciones).
- Q17_1: Variación del anterior benchmark en que sólo se pide la primera solución, siendo el número de reinas igual a 17 y el tablero de 17x17.
- HAM: Encuentra todos los caminos hamiltonianos de un grafo (60 soluciones).
- TRI: Corresponde a la versión estructurada del programa triangle de [115] (133 soluciones).
- TURTLES: Programa del mismo nombre correspondiente a un puzzle especificado en [115] (1 solución).
- ZEBRAS: Otro problema lógico cuya descripción también está en [115] (4 soluciones).

En todos los benchmarks se explora el árbol de búsqueda completo para encontrar todas las soluciones, excepto en el programa *q17_1* en el que únicamente se explora aquella parte del árbol de búsqueda necesaria para encontrar la primera solución.

7.3 Rendimiento de SMAM frente a WAM

El intérprete software de SMAM se ha ejecutado en una estación de trabajo Digital Alpha Station 5/166 cuya CPU es un Alpha 21164.

La evaluación del sistema Multipath se basa en comparar su rendimiento con respecto a la ejecución de los mismos benchmarks mediante la WAM. Para ello, se ha desarrollado otro intérprete software de la WAM (junto con su ensamblador y montador correspondiente), escrito con el mismo estilo de programación que el intérprete de la SMAM, al cual también se le han aplicado el mismo tipo de optimizaciones a nivel de compilación. Con ello se pretende comparar la ejecución de ambos intérpretes en las mismas condiciones.

7.3.1 Comportamiento previsible

Las ventajas que ofrece Multipath en una ejecución secuencial provienen, por el hecho de recorrer a la vez varios caminos del árbol de búsqueda de un programa, de la reducción del número total de instrucciones de control y de instrucciones de datos que operan con variables

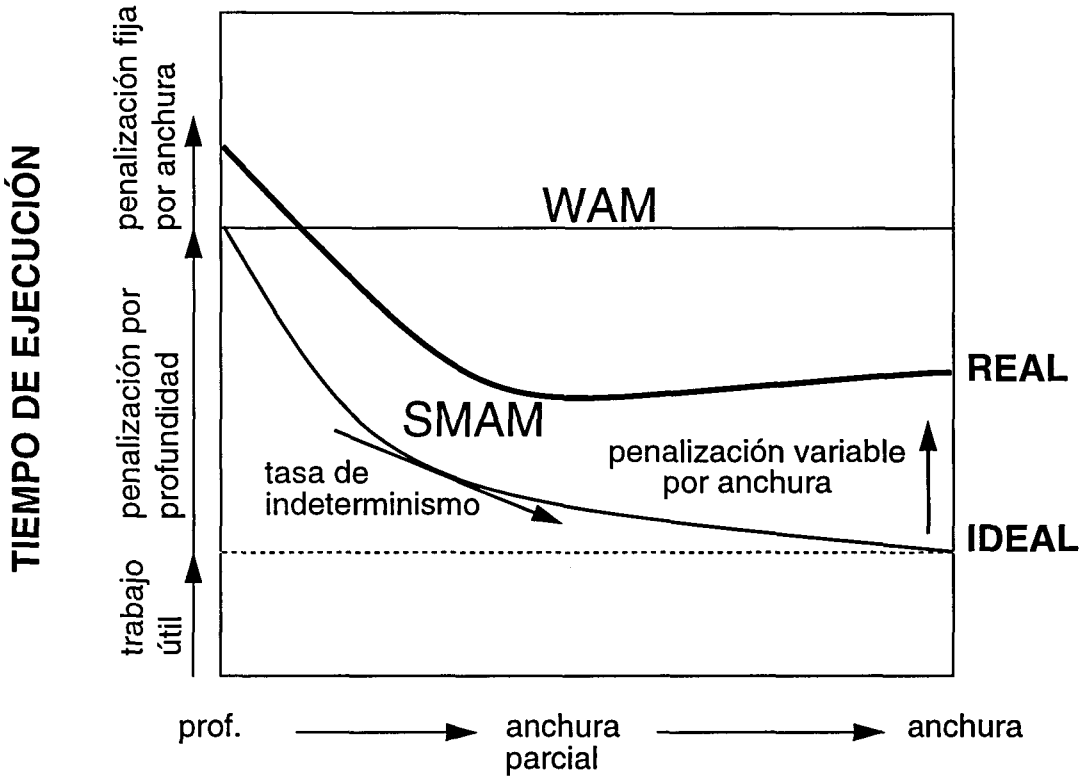


Figura 7.2: *Previsión del comportamiento IDEAL y REAL de la SMAM respecto la WAM.*

simples. Las características propias de cada programa repercuten en el rendimiento de Multipath: conforme el programa se comporte de forma más indeterminista, Multipath será capaz de recorrer simultáneamente un mayor número de caminos y, por tanto, sus beneficios serán más evidentes.

Con el fin de caracterizar a los benchmarks, definimos la **tasa de indeterminismo** como el cociente entre el número medio de UEs activas (en estado CURRENT) durante la ejecución del programa y el número máximo de UEs (parámetro NUM_UE). Esta razón depende únicamente del benchmark y permanece de forma casi constante conforme varía el parámetro NUM_UE. Cuanto mayor sea la tasa de indeterminismo de un programa mayor será la ganancia de Multipath para un mismo valor de NUM_UE.

El comportamiento que es lógico prever en cuanto a tiempo de ejecución de la SMAM respecto la WAM en la exploración de todo el árbol de búsqueda de un programa se muestra en la figura 7.2.

En primer lugar, el tiempo de ejecución de la WAM se desglosa en dos componentes:

- **Penalización por profundidad**, consistente en el tiempo gastado debido a la repetición de las instrucciones de control y de datos ocasionada por la exploración en profundidad.
- **Trabajo útil**, formado por el resto de tiempo y que consiste en la ejecución mínima (una única vez) de las diferentes instrucciones de control y de datos necesarias para recorrer todo el árbol de búsqueda.

El comportamiento ideal de SMAM conforme se aumente NUM_UE, es decir, se tienda hacia una exploración total en anchura, reducirá la penalización por profundidad. En el caso límite de una exploración total en anchura, el tiempo de ejecución pasará a ser únicamente el tiempo de trabajo útil. La pendiente de esta reducción está influenciada por la tasa de indeterminismo. Cuanto mayor sea esta tasa, la velocidad de disminución del tiempo de ejecución será mayor.

Por otra parte, es lógico prever que la gestión de la exploración en anchura realizada en Multipath introduzca un nuevo tipo de penalización. Los inconvenientes que plantea Multipath se pueden desglosar en tres factores principales:

- El hecho de gestionar nuevos registros y estructuras de datos a los existentes en la WAM.
- La aplicación de la técnica de copia de entornos durante la inicialización de un UE.
- El incremento en el tiempo medio de acceso a memoria conforme se aumenta el grado de exploración en anchura. Esto es debido al aumento del working set con que trabaja el programa y la consiguiente degradación del rendimiento de la jerarquía de memoria.

La penalización ocasionada por el primer factor se define como **penalización fija por anchura**, mientras que los dos últimos factores ocasionan la denominada **penalización variable por anchura**. Todo ello significa que el comportamiento real de SMAM diferirá respecto al comportamiento ideal de la siguiente forma:

- El tiempo de ejecución de SMAM cuando NUM_UE = 1 (equivale a realizar una exploración en profundidad) es ligeramente superior al tiempo de WAM ya que no pueden aplicarse las ventajas de exploración en anchura y se introduce la penalización por la gestión de nuevos elementos arquitectónicos (penalización fija por

anchura).

- Conforme se aumenta NUM_UE cabe esperar una reducción del tiempo de ejecución marcada por la razón de indeterminismo del programa aunque con una velocidad menor a la del comportamiento ideal debido a la penalización variable por anchura.
- Otro punto importante en la curva del comportamiento real corresponde a aquel momento en que todas las penalizaciones de la SMAM (penalización fija y penalizaciones variables por copia de entornos y por incremento del tiempo medio de acceso a memoria) aumentan significativamente hasta un punto en que la penalización global supera las ventajas de Multipath y, por consiguiente, el tiempo de ejecución de SMAM se incrementa.

7.3.2 Datos experimentales

El tiempo de ejecución concreto para todos los benchmarks se muestra en la figura 7.3. En cada gráfica, el tiempo de ejecución de SMAM se ha obtenido variando el número máximo de UEs (parámetro NUM_UE del modelo arquitectónico) desde 1 hasta 1000. El tiempo de ejecución WAM se muestra como una constante para poder apreciar la diferencia de rendimiento.

En todos los benchmarks, excepto para *q17_1*, se calculan todas las soluciones del programa. En estos casos, el árbol de búsqueda a recorrer en WAM y en SMAM es el mismo, por lo que puede aplicarse la predicción del comportamiento de Multipath descrita anteriormente. En este sentido, se aprecia como el tiempo de ejecución de cada benchmark se ajusta al comportamiento real previsible. Antes de pasar a analizar con más detalle los resultados, nótese como en todos los benchmarks excepto uno (*ham*), SMAM ofrece una ganancia respecto a WAM que en algún caso (*bitspal*) puede ser de más de un orden de magnitud.

Los factores que influencian en mayor medida el rendimiento de Multipath son:

- La tasa de indeterminismo.
- El número de copias efectuadas.
- El tiempo medio de acceso a memoria por referencia.

Un análisis de estos tres factores permite clasificar los distintos benchmarks. En este sentido, la figura 7.4 nos muestra el valor de estos factores. En cada una de las gráficas se muestran únicamente aquellos benchmarks que facilitan su visión. A continuación se indica el procedimiento utilizado para obtener la información necesaria en cada gráfica.

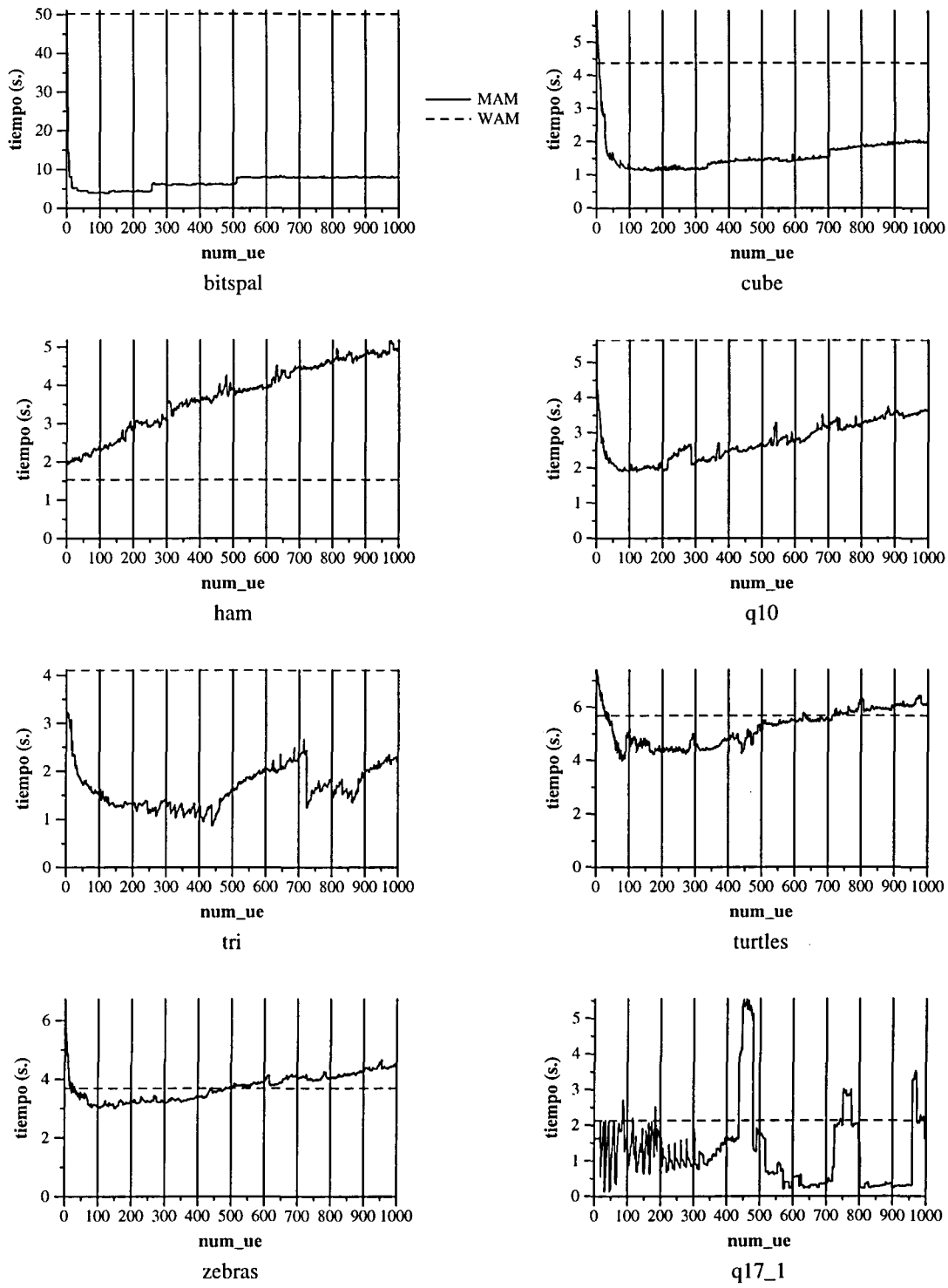


Figura 7.3: *Tiempo de ejecución de SMAM y WAM para el conjunto de benchmarks analizados.*

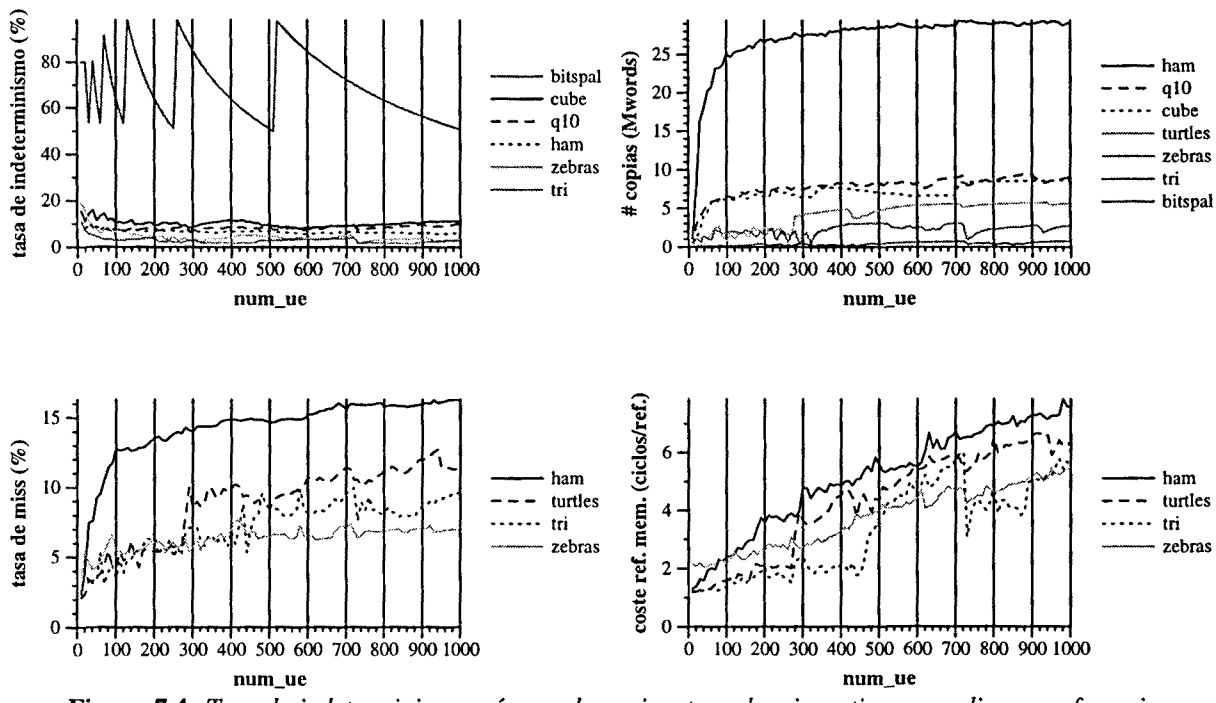


Figura 7.4: *Tasa de indeterminismo, número de copias, tasa de miss y tiempo medio por referencia a memoria en los benchmarks analizados.*

El cálculo de la tasa de indeterminismo para un cierto valor del parámetro NUM_UE requiere conocer el número medio de UEs activas durante la ejecución del programa. Este número se obtiene como el cociente entre el número de instrucciones MAM ejecutadas y el número de instrucciones MAM que se ejecutan cuando NUM_UE es igual a 1. La aplicación de este criterio supone que la distribución de las instrucciones a lo largo de todo el árbol de búsqueda es homogénea.

La gráfica correspondiente a las copias se mide en millones de palabras y se calcula a partir del número de copias de entornos realizadas multiplicada por el tamaño medio de un entorno.

La influencia del tiempo de acceso a memoria se muestra en dos gráficas. Por un lado, se muestra la tasa de miss en el primer nivel de la memoria caché del procesador Alpha 21164, obtenido mediante simulación utilizando la herramienta de instrumentalización *atom* [105]. Por otro lado, se muestra el coste de cada referencia a memoria medido en ciclos del procesador. Este dato corresponde a una estimación ya que se calcula considerando que cada instrucción ejecutada por el procesador que no genera acceso a memoria tiene un coste fijo (t). De esta

forma, el tiempo de ejecución de una instrucción que genera acceso a memoria se calcula dividiendo el tiempo total del programa menos el tiempo no gastado en memoria (obtenido a partir de t) entre el número total de accesos a memoria. Este dato es bastante exacto cuando el working set es elevado y, en cualquier caso, permite tener una estimación del incremento del tiempo de acceso a memoria conforme aumenta el tamaño del programa.

La observación de las dos gráficas relativas al coste de las referencias a memoria confirma una de las previsiones realizada respecto al modelo. A medida que se incrementa el grado de exploración en anchura de un programa existe un incremento lineal en el tiempo medio de acceso a memoria. Cuando el número de UEs es bajo cada referencia a memoria tiene un coste medio entre 1 y 2 ciclos de reloj. Cuando el número de UEs se incrementa hasta 1000, cada referencia a memoria llega a tardar entre 6 y 8 ciclos. Cada benchmark tiene una pendiente diferente al incrementar el número de UEs pero no discrimina lo suficiente para poder catalogarlos.

Debe tenerse en cuenta que el Modelo Arquitectónico de Multipath podría ser realizado en un sistema con soporte a la ejecución *multithreading* [56]. Con este tipo de soporte, se intenta reducir al máximo la penalización por cambio de contexto entre threads. En este sentido, las acciones asociadas a un UE corresponderían a un thread, y la PU podría explotar la concurrencia existente entre los threads para esconder la latencia a memoria principal en aquellos casos en que se produjese un fallo en el acceso a la memoria caché.

Los factores que discriminan en mayor medida el comportamiento de un programa son la tasa de indeterminismo y el número de copias efectuadas. Así, por ejemplo, el benchmark que mejor explota las ventajas de Multipath es *bitspal*. Es el programa que presenta una mayor tasa de indeterminismo y, a la vez, el número de copias que se efectúan es despreciable. En este caso, con un número relativamente pequeño de UEs se obtiene el tiempo correspondiente al denominado trabajo útil. Conforme el número de UEs se incrementa, el tiempo aumenta ligeramente por el hecho de tratar con un working set más grande. Debido a que es el programa con mayor tasa de indeterminismo, la penalización por profundidad es muy alta. Multipath la elimina, llegando a reducir el tiempo de ejecución respecto la WAM en más de un orden de magnitud.

El efecto de disminuir el grado de indeterminismo de un programa se puede observar en el programa *tri*. En este caso, el número de copias también es relativamente bajo pero la tasa de indeterminismo es la menor de todos los programas. Ello motiva que la pendiente de disminución del tiempo de ejecución conforme aumentamos NUM_UE es menor y, por consi-

guiente, el número de UEs óptimo tiende a incrementarse. En este benchmark, el tiempo de ejecución llega a ser hasta más de 4 veces menor respecto a la WAM.

El principal problema de Multipath corresponde a la penalización introducida por la copia de entornos. Esto se puede observar en los benchmarks *cube*, *q10*, *ham*, *zebras* y *turtles*. Con una tasa de indeterminismo relativamente pequeña (desde el 10% hasta el 5%), si el número de copias no es excesivamente alto (cosa que sucede en todos excepto en el benchmark *ham*) siempre existe un número óptimo de UEs en que el tiempo de ejecución es menor que el de WAM. Sin embargo, cuando el número de copias se incrementa enormemente (alrededor de 25 millones de palabras en *ham*) el rendimiento de Multipath es ligeramente inferior al de WAM.

La conclusión más evidente es que el tamaño de los entornos (número de variables múltiples de un programa) y el número total de copias de entornos son dos factores que influyen en gran medida en el rendimiento de Multipath. Por otra parte, las ventajas son mucho mayores cuanto más indeterminista se comporta un programa, como era de esperar desde un principio. En resumen, la ganancia de SMAM en los benchmarks analizados puede llegar a ser de hasta 13 veces en el mejor caso, de 0.7 en el peor de los casos y entre 2 y 5 veces en un término medio. Las técnicas descritas en el capítulo 5 respecto a gestión concurrente de entornos deben constituir el primer punto de mira de cara a aumentar en mayor medida aún la eficiencia del modelo propuesto en Multipath.

En el caso de encontrar una única solución del programa, el rendimiento entre la SMAM y la WAM se comporta de forma errática según varía NUM_UE ya que, aparte de los tres factores mencionados, depende también de la cantidad de exploración del árbol de búsqueda necesaria para encontrar la primera solución. Existirán situaciones en que MAM explore una menor parte del árbol y la reducción del tiempo de ejecución es realmente importante y otras situaciones en que explora una mayor parte del árbol de tal forma que no se consigue reducir el tiempo de la WAM.

7.4 Determinación automática del número óptimo de UEs

En la sección anterior se ha demostrado que el rendimiento de Multipath es muy significativo. Sin embargo, también es cierto que el parámetro NUM_UE puede provocar una diferencia de rendimiento notable y, por otra parte, su valor óptimo depende de las características concretas de cada programa.

Es deseable que el número óptimo de UEs pueda ser calculado automáticamente por el sistema. En este sentido, se propone la siguiente estrategia:

- Calcular de forma estática el tiempo medio en ejecutar un camino de un subconjunto del árbol de búsqueda para un rango de valores de NUM_UE, y elegir aquel valor que minimice dicho tiempo.

Para poder aplicar con éxito esta estrategia es necesario que el recorrido del árbol de búsqueda sea homogéneo durante toda la ejecución del programa. Para comprobar esta condición, se ha calculado el tiempo medio en recorrer un camino del árbol de búsqueda en base a explorar todo el árbol completo y, por otra parte, el tiempo medio por camino obtenido al explorar únicamente los primeros 16K caminos. La figura 7.5 muestra el tiempo obtenido para cada benchmark aplicando este criterio. Se puede comprobar que prácticamente no existen diferencias significativas en el cálculo del tiempo medio por camino. En consecuencia, en estos benchmarks el recorrido del árbol de búsqueda se considera que es homogéneo.

En base a esta conclusión, y antes de la fase de interpretación real, se realiza una fase adicional, denominada interpretación real reducida (IR*), con la finalidad de calcular el número de UEs óptimo para un benchmark. En esta fase se aplica la estrategia indicada anteriormente y, en concreto, recorre los primeros 16K caminos del árbol de búsqueda midiendo el tiempo gastado en recorrer, de estos 16K, los últimos 8K caminos. Los primeros 8K forman parte del transitorio inicial y no son considerados a la hora de calcular el tiempo medio por camino. Este recorrido se repite para un rango de NUM_UE igual a [20, 500]. Posteriormente, se ordenan todos los tiempos medios por camino y se obtiene el valor óptimo para el parámetro NUM_UE. La figura 7.6 simboliza las fases de ejecución de que consta la realización secuencial de Multipath.

Otra ventaja adicional que se consigue con esta nueva fase de interpretación real reducida es la de poder discernir entre aplicar el modelo Multipath o el modelo convencional de ejecución. Al igual que se determina el tiempo medio en ejecutar un camino en la SMAM, también se calcula para la WAM. A la hora de decidir el número óptimo de UEs puede obtenerse que es más recomendable ejecutar el modelo convencional. En este sentido, en el sistema Multipath nunca se incrementa el tiempo de ejecución respecto al modelo convencional. En cambio, se puede disminuir en gran medida según las características concretas del programa.

El cálculo automático del valor óptimo de NUM_UE parte de la condición de que el programa se comporte de forma homogénea durante toda su ejecución. En caso que esta premisa no se cumpla, se propone ejecutar dinámicamente un algoritmo que, en caso de una variación substancial del tiempo medio por camino, modifique el parámetro NUM_UE con el fin de adaptarse a los cambios que pueda presentar el árbol de búsqueda. Esta propuesta no ha sido

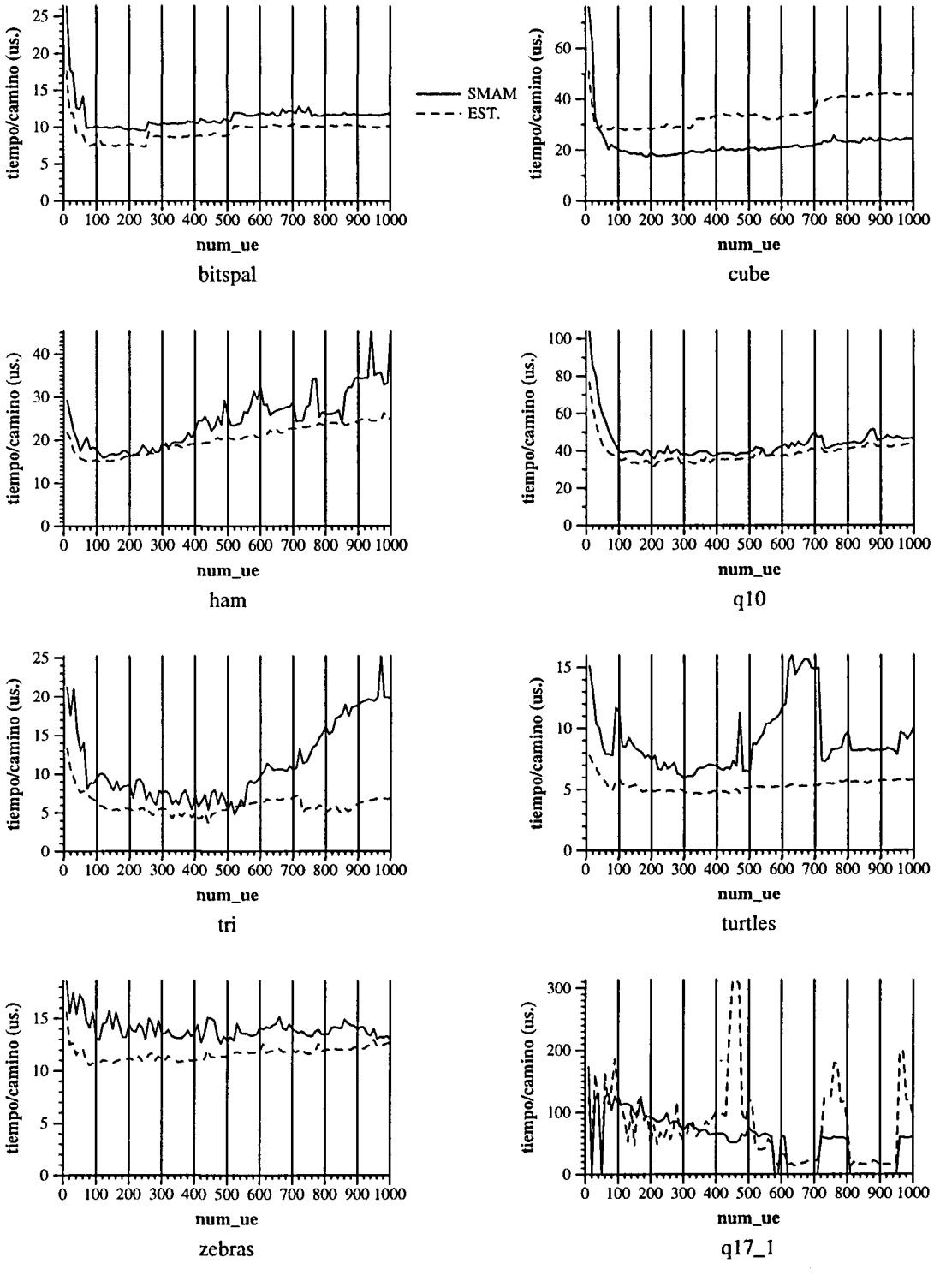


Figura 7.5: *Tiempo medio de ejecución de un camino, según todo el árbol (SMAM) o un subconjunto inicial de caminos (EST.).*

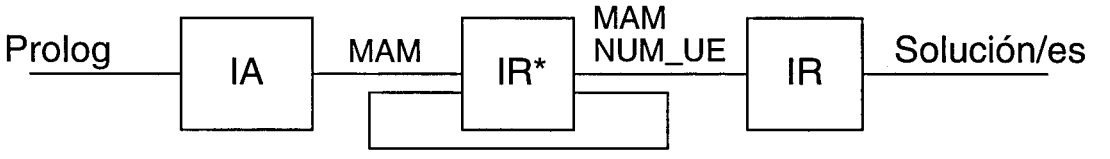


Figura 7.6: *Fases de ejecución de la realización secuencial de Multipath.*

realizada en la versión actual de Multipath.

7.5 Rendimiento de Multipath

En este apartado se resumen los datos más significativos de la ejecución secuencial de Multipath. En la figura 7.6 se observa el speed-up relativo entre Multipath y la ejecución convencional de Prolog identificada por la WAM. Hay que destacar que Multipath se comporta como la WAM en el peor de los casos con un speed-up igual a 1 y que, en el mejor de los casos, el speed-up llega a 13. En un punto intermedio se encuentran el resto de benchmarks con speed-ups entre 1.20 y 4.28. Respecto al benchmark que calcula sólo la primera solución (*q17_1*) el speed-up también es significativo llegando hasta aproximadamente una orden de magnitud.

La tabla 7.1 muestra de forma numérica los datos más representativos. En primer lugar, el tiempo de ejecución de Multipath y de la WAM, junto con el speed-up relativo entre los dos

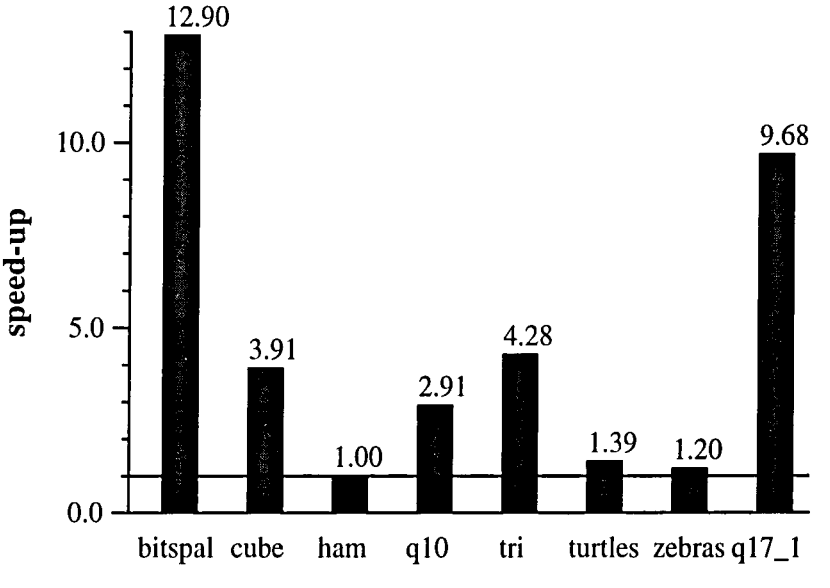


Figura 7.7: *Speed-up Multipath vs. WAM.*

	bitspal	cube	ham	q10	tri	turtles	zebras	q17_1
Tiempo Multipath (s)	3.89	2.27	1.53 (2.32)	1.94	0.96	4.10	3.08	0.22
Tiempo WAM (s)	50.20	4.38	1.53	5.64	4.11	5.68	3.69	2.13
Speed-up	12.90	1.93	1.00	2.91	4.28	1.39	1.20	9.68
Num_ue óptimo	114	128	WAM (92)	210	445	86	156	32
Instr. WAM (mil)	229641	13664	5074	16187	14537	15412	8748	6219
Instr. MAM-1 (mil)	205524	13898	4833	15007	9524	13767	8581	5745
Tiempo SMAM-1 (s)	51.07	5.95	1.91	4.25	3.21	7.40	6.73	1.61
Tasa indeter. (%)	56.08	10.68	(6.80)	7.03	1.60	4.06	5.01	6.25
Copias (Mwords)	0.01	6.27	(23.76)	6.94	0.13	3.05	1.85	0.16
Tasa miss (%)	6.35	5.62	(11.71)	9.59	5.32	4.43	5.53	1.60
Ciclos / ref. mem.	2.30	2.19	(2.26)	2.79	1.78	1.56	2.48	1.33
Var. simples	1	1703	(482)	32576	7111	10731	7	42670
Var. múltiples	20	53258	(29454)	61957	1620	114670	33	15117
Var. nulas	0	0	(0)	0	0	6436	45825	0
Instr. MAM (mil)	3214	1016	(772)	1016	1339	3945	1097	621
Com. MAM (mil)	278	285	(505)	272	444	2271	760	108

Tabla 7.1: *Resumen de los datos más significativos de la ejecución secuencial de Multipath.*

modelos. También se indica el número de UEs óptimo en la ejecución de Multipath. Nótese que en el benchmark *ham* se determina cómo óptima la ejecución convencional de la WAM. En este benchmark, los datos correspondientes a la ejecución forzando la utilización de SMAM se indican entre paréntesis.

A continuación se indica el número de instrucciones ejecutadas para cada programa en el modelo WAM y, seguidamente, las instrucciones y tiempo de ejecución de SMAM cuando NUM_UE es igual a 1. La comparación entre el número de instrucciones ejecutadas en ambos modelos permite observar la ganancia obtenida en Multipath por el efecto de reducir la cantidad de indeterminismo de un programa. Nótese que pueden existir casos en que no exista tal ganancia. Esto sucede si el número de instrucciones obviadas es menor en comparación con las instrucciones adicionales que deben introducirse en la MAM. Téngase en cuenta que el objetivo

principal en Multipath es la exploración eficiente del árbol de búsqueda y, como objetivo secundario, se encuentra la reducción del indeterminismo. De cualquier forma, existe una ganancia significativa en los benchmarks *q10*, *tri* y *q17_1*.

En el siguiente grupo de datos se indica la tasa de indeterminismo, el número de palabras copiadas, la tasa de miss en el acceso al primer nivel de memoria caché y el número medio de ciclos por referencia a memoria. Estos valores permiten caracterizar el comportamiento de Multipath, y han sido analizados en el apartado 7.3 dedicado al rendimiento del sistema.

Por último, se indica el número total de variables que se crean en cada programa, según su clasificación en variables simples, múltiples o nulas, y el número de instrucciones y comandos MAM ejecutados. Estos datos permiten conocer la proporción de variables simples respecto de variables múltiples y la de comandos por instrucción. Un programa se comporta mejor cuando el número de variables simples se incrementa, el número de variables múltiples disminuye y el número medio de comandos por instrucción se decrementa.

7.6 Comparación con Multilog

En este último apartado se compara el rendimiento de Multipath respecto a Multilog, el único sistema existente en la literatura que también aplica una exploración parcial en anchura. Téngase en cuenta que la definición inicial de Multipath y Multilog apareció publicada por primera vez en 1993 de forma simultánea, de tal manera que los dos trabajos se han desarrollado independiente y concurrentemente desde entonces.

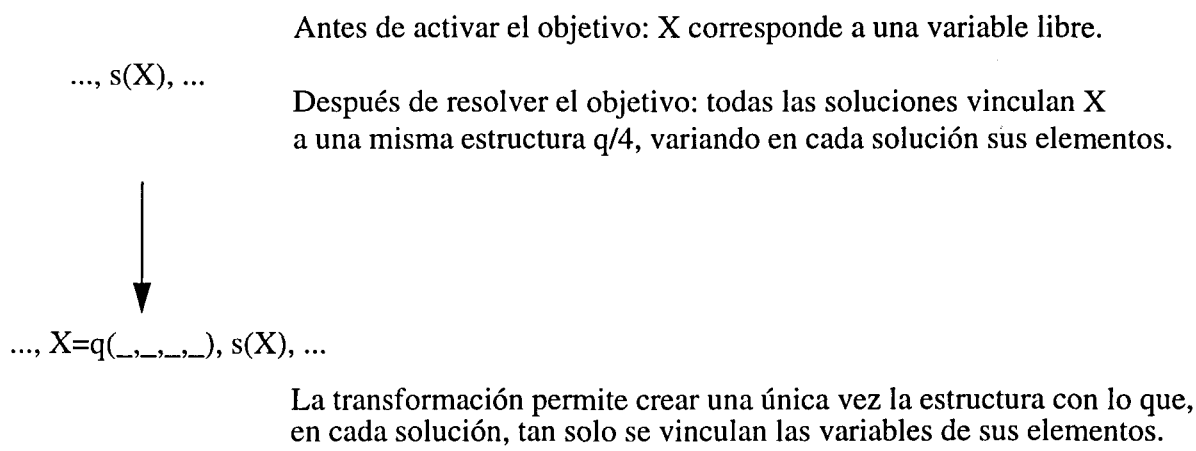


Figura 7.8: *Ejemplo de transformación de un programa por detección de patrones comunes en las soluciones.*

	bitspal	cube	q10	tri
Speed-up Multipath	12.90	3.91	2.91	4.28
Speed-up Multilog	12.40	3.10	1.40	1.40
Speed-up relativo	1.04	1.26	2.07	3.05

Tabla 7.2: *Comparación del sistema Multipath con Multilog*

Para poder comparar el rendimiento de Multipath con el de Multilog es necesario introducir una técnica presentada en Multilog. Esta técnica consiste en determinar un patrón (template) que caracterice a todas los vínculos de una variable en cada una de las soluciones obtenidas a través de un objetivo explorado en anchura [109]. Si se encuentra este patrón, se crea al principio de la activación del objetivo. De esta forma se evita tener que crear la misma estructura asociada al vínculo de una variable en todas las soluciones que se calculen (ver figura 7.7).

En Multilog no se propone ningún proceso automático para poder aplicar todas sus propuestas (incluida este tratamiento de los patrones).

En Multipath se ha definido el proceso de interpretación abstracta con el objetivo de determinar los objetivos susceptibles de ser explorados en anchura, el número de caminos óptimo de exploración en anchura, el tipo dinámico de las variables y la aplicabilidad de técnicas de reducción del indeterminismo. Sin embargo, no se ha descrito una gestión de los patrones en el sistema Multipath ya que no se ha definido el proceso automático de detección y transformación del código Prolog que requiere.

La contemplación de cada uno de los benchmarks nos permite observar que la gestión de los patrones únicamente sería aplicable en el programa *cube*. Se ha realizado manualmente la transformación del código, notándose una ganancia substancial en este programa en que puede ser aplicada. En concreto, la ejecución de este programa ofrece un speed-up de 2.02 respecto a la versión de Multipath descrita hasta ahora. Teniendo en cuenta esta modificación, y que sólo se conocen datos de la ejecución de Multilog para otros tres benchmarks [108], se presenta en la tabla 7.2 la comparación del rendimiento entre Multipath y Multilog. El speed-up corresponde a la ganancia de cada sistema respecto a la versión WAM utilizada en cada uno de ellos. Nótese que el speed-up relativo refleja, en todos los benchmarks, que existe una ganancia en el rendimiento del sistema Multipath, llegando a ser hasta 3 veces más rápido que Multilog.

7.7 Resumen y contribuciones

En este capítulo se ha presentado y analizado la realización secuencial de Multipath. El análisis se ha concentrado en la fase de interpretación real, que está basada en la Máquina Abstracta de Multipath, y se denomina SMAM. Se han determinado los factores que determinan el rendimiento de SMAM y se ha comparado la ejecución de Multipath con la ejecución convencional de Prolog y con el sistema Multilog.

En base a los resultados obtenidos, las principales conclusiones son las siguientes:

- Una ejecución secuencial que explora parcialmente en anchura un árbol de búsqueda es, por regla general, más eficiente que la tradicional exploración en profundidad.
- Los factores que influencian en gran medida el rendimiento de la ejecución de SMAM son la tasa de indeterminismo de un programa y el número de copias realizadas.
- Dado un programa, es posible determinar la conveniencia de aplicar una exploración fija en profundidad o una exploración parcial en anchura y, en este último caso, calcular el grado óptimo de exploración en anchura.
- Para los benchmarks analizados, el speed-up de Multipath respecto la ejecución convencional varía desde 1.00, en el peor de los casos, hasta 12.90, en el mejor de ellos. El speed-up de Multipath respecto Multilog varía desde 1.04 hasta 3.05

8

REALIZACIÓN PARALELA DE MULTIPATH

La segunda realización de Multipath se ha llevado a cabo en un entorno paralelo. Al igual que en la realización secuencial, en este capítulo se discuten únicamente los aspectos relacionados con la realización de la fase de interpretación real de un programa.

El objetivo en este caso es disponer de una herramienta con la que poder medir el rendimiento de una ejecución paralela del modelo arquitectónico. El mapeo de la Máquina Abstracta de Multipath en este entorno de ejecución paralela se denomina PMAM. La primera decisión que se ha llevado a cabo consiste en utilizar un sistema paralelo de propósito general ya existente, y programar las acciones del modelo arquitectónico totalmente en software mediante un lenguaje de programación soportado por dicho sistema (lenguaje C). El estudio del comportamiento de Multipath en este sistema se orienta hacia detectar las características que debe poseer un programa para obtener una ganancia significativa respecto la ejecución secuencial y, por otra parte, distinguir, si existen, los principales motivos de penalización que presente. En este caso, pueden proponerse aquellos elementos específicos a introducir en el hardware con vistas a obtener la mayor eficiencia posible.

8.1 Interpretación real en PMAM

La representación del control de flujo que propone el Modelo Arquitectónico de Multipath se refleja en la figura 8.1a. Por un lado, existe el secuenciamiento propio del ME, que es el resultado de ejecutar las operaciones que se han definido para este motor. Por otro lado, existe la concurrencia, que permite el paralelismo de caminos existente en Multipath, entre las operaciones ejecutadas por los UEs. Cada una de las operaciones en que se divide el trabajo de un UE corresponde a la ejecución de un comando. Por su parte, el trabajo del ME también se divide en operaciones, que están limitadas por el envío de dos comandos consecutivos. El tipo de dependencias que existen entre las operaciones del ME y de los UEs son las siguientes:

- **Operaciones UE**: El inicio de una operación en un UE concreto depende del envío del comando correspondiente y de que el UE se encuentre en el estado requerido. Todas las operaciones UE asociadas a un mismo comando son independientes entre si. Para un mismo UE, todas sus operaciones deben respetar el secuenciamiento indicado por el orden en que se han enviado los comandos.
- **Operaciones ME**: En el secuenciamiento global del ME existe un punto donde existen dependencias de control, que corresponde con la sincronización requerida

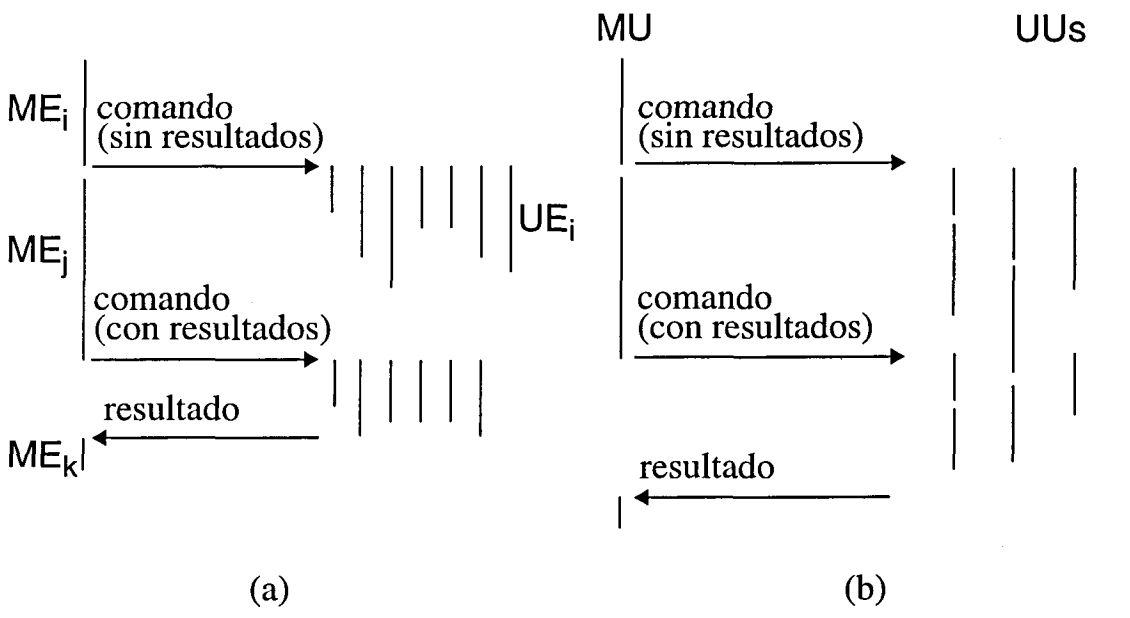


Figura 8.1: *Secuenciamiento en Multipath:*

(a) Control de flujo entre las operaciones definidas en MAM.

(b) Planificación de las operaciones a unidades de proceso en PMAM.

tras el envío de un comando. Hay dos posibilidades: (i) el comando no requiere sincronización, y (ii) el comando sí requiere sincronización. En el primer caso, el ME puede continuar ejecutando la siguiente operación de forma concurrente con todas las operaciones UE anteriores que estén pendientes. En el segundo caso, el ME debe esperar a que finalice la última operación UE asociada al comando.

Este tipo de proceso responde a una estrategia SPMD (Single Program Multiple Data). No obstante, y por motivos de disponibilidad, la versión paralela de la fase de interpretación real de Multipath, que simbolizamos como PMAM, se realiza en un multiprocesador con memoria compartida denominado *SGI Power Challenge XL,* que dispone de 8 procesadores MIPS R8000. Este multiprocesador responde a un modelo de proceso MIMD (Multiple Instructions Multiple Data). Una realización simple de Multipath en esta arquitectura MIMD dejaría al sistema operativo la decisión de distribuir la carga ocasionada por las operaciones potencialmente paralelas a tareas que serían ejecutadas entre todas las unidades de proceso disponibles. Sin embargo, la penalización por el cambio de contexto entre las tareas que ejecutasen estas operaciones es demasiado grande frente a la baja granularidad que poseen. Recuérdese que los comandos están asociados a unificaciones entre términos Prolog. Por ello, se propone una estrategia de planificación y de distribución de la carga controlada por el intérprete PMAM.

En este sentido, se definen dos tipos de unidades de proceso (ver figura 8.1b):

- **Unidad Principal (MU).** Es la responsable de ejecutar las operaciones del ME.
- **Unidad de Unificación (UU).** Cada UU es responsable de ejecutar las operaciones de un conjunto de UEs. Existe un número parametrizable de UUs en esta versión de Multipath, que es simbolizado por NUM_UU.

El número total de unidades de proceso presentes en el sistema es 1 + NUM_UU.

A continuación se describen las características más relevantes de esta realización. En primer lugar, se describe la planificación de operaciones UE a UUs, así como la información adicional que requiere la MU. Posteriormente, se describe la gestión de los comandos y, en último lugar, la distribución de la carga.

8.1.1 Planificación de UEs a UUs

Al igual que en la versión secuencial, es importante destacar la necesidad de agilizar el conocimiento del estado en que se encuentra un UE para determinar si debe ejecutar un comando o no. En lugar de tener un registro de estado en cada UE, se gestiona un vector de UEs para

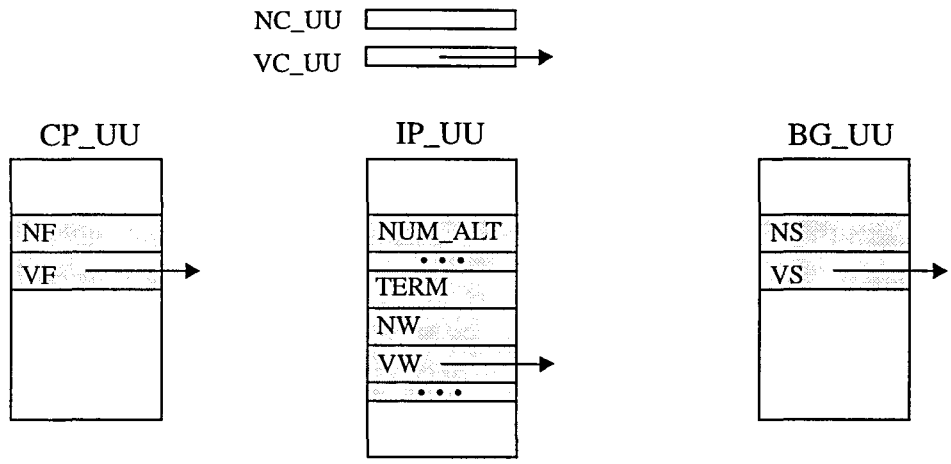


Figura 8.2: *Estructuras de datos asociadas a la planificación de los UEs asignados a cada UU.*

cada posible estado conteniendo la identificación de todos los UEs que se encuentran en dicho estado.

Estos vectores son los responsables de determinar el orden en que se ejecutarán, en cada UU, las operaciones UE correspondientes a un mismo comando. La figura 8.2 muestra las estructuras de datos asociadas a la asignación de UEs a UUs. En concreto, cada UU dispone de un registro denominado NC_UU, que contiene el número total de UEs asignadas a dicha UU en estado CURRENT; y de un registro VC_UU, que contiene la dirección base de todos los identificadores de UEs en estado CURRENT. Para los estados FAILED, WAITING y SOLUTION, cada UU dispone de unos vectores adicionales denominados CP_UU, IP_UU y BG_UU, respectivamente. Los elementos que almacenan están asociados con los puntos de selección (cp); puntos de indexación (ip) y objetivos en anchura (bg) que posea el ME en un momento de la ejecución, y contienen dos campos que reflejan el número de UEs y la dirección base del vector de identificadores de UEs. Recuérdese que un UE puede estar en estado FAILED de un cp determinado; en estado WAITING de un ip y un término concreto; o en estado SOLUTION de un bg determinado. Los comandos que afectan a UEs en estos estados tienen un parámetro, a partir del cual, una UU indexa en estas estructuras de datos para acceder al vector de UEs correspondiente.

Debido a que el sistema está realizado en un multiprocesador con memoria compartida, todas las estructuras de datos propuestas en MAM para los UEs son accesibles por cualquier UU. En este sentido, no hay ninguna restricción en cuanto a la planificación de UEs a una UU. Un UE puede ser asignado a cualquier UU.

Respecto a los UEs que se encuentran en estado AVAILABLE, la unidad de proceso encargada de controlar su gestión es la MU. En este sentido, la MU posee un vector denominado VAUE, que permite conocer en un momento determinado todas las UEs asociadas a este estado.

8.1.2 Gestión de los comandos

El envío de los comandos por parte de la MU y su recepción por parte de las UUs se gestiona por medio de una zona de memoria compartida denominada workpool. Asociado al workpool se definen una serie de bits de sincronización que notifican la existencia de nuevos comandos y la finalización de estos. La figura 8.3 muestra gráficamente la estructura del workpool y los bits de gestión del sincronismo. A continuación, se describe de forma textual con más detalle.

8.1.2.1 Workpool

El workpool tiene capacidad para almacenar MAX_COMANDOS, y cada entrada contiene una serie de campos que almacenan el código de operación (CO) y los argumentos de un comando. Existen tres campos para los argumentos (A1, A2 y A3), que corresponde con el número máximo de parámetros que puede tener un comando (ver tablas 6.1 y 6.2). El registro iWP_MU apunta siempre a la entrada del workpool en que debe publicarse el siguiente comando.

A nivel de modelo arquitectónico, se han definido 4 posibles tipos de resultados en los comandos. En PMAM, se tratan los dos tipos siguientes: (i) STATE, que corresponde a una

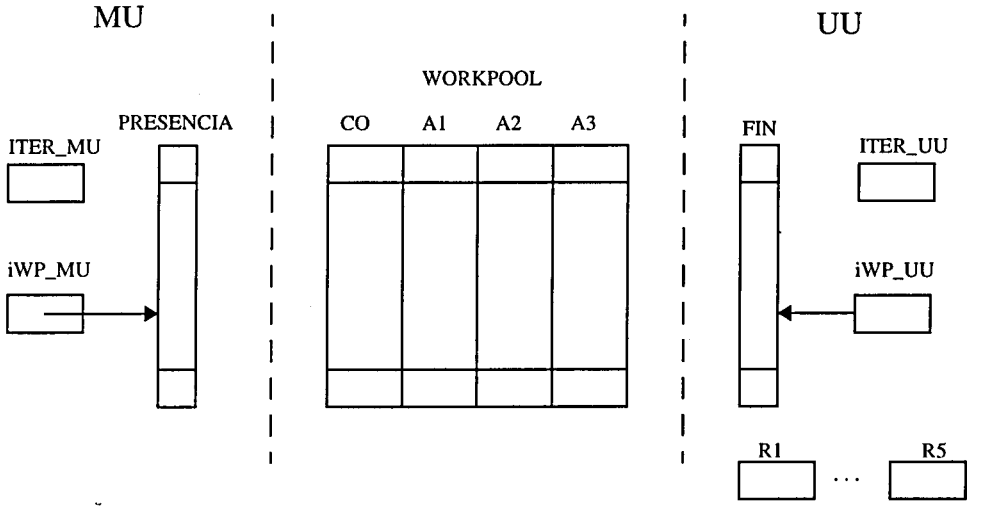


Figura 8.3: *Estructuras de datos asociadas a la gestión de los comandos.*

indicación del estado de cada UE tras finalizar la ejecución del comando; y (ii) MODE, que corresponde a una indicación del modo de unificación de términos estructurados. Este modo puede ser READ o WRITE en cada UE. Para ello, existen 5 registros (R1 a R5) que almacenan los resultados calculados en una UU referentes a estos dos tipos de resultados.

Para el primer tipo, en lugar de retornar un vector con la información asociado a cada UE, una UU únicamente devuelve el número total de UEs en estado CURRENT (en R1), y el número de UEs (en R2) y dirección base de los identificadores de UEs en estado AVAILABLE (en R3). El número de UEs en estado FAILED asociados a cada punto de selección es calculado por la MU a partir de la información existente en CP_UU.

Para el segundo tipo de resultados, una UU devuelve en el registro R4 una indicación acerca de si tiene UEs asignadas en modo READ; y en el registro R5 una indicación sobre si existen UEs que realizan la unificación en modo WRITE.

Los comandos que no retornan un resultado de este tipo tienen un tratamiento especial, y son descritos en el apartado siguiente.

8.1.2.2 Sincronización

Existe necesidad de sincronización en las UUs para saber si existe un nuevo comando por procesar y en el MU para conocer la finalización de un comando.

Se define un bit de presencia de comando por cada entrada del workpool, que es actualizado por la MU y consultado por las UUs. Este bit contiene el valor 1 (en pasadas pares por el workpool) y el valor 0 (en pasadas impares) para indicar la presencia de un nuevo comando. El registro ITER_MU indica si se está recorriendo el workpool en una iteración par o impar. De esta forma, se evita tener que inicializar el bit de presencia cada vez que la MU trata la finalización de un comando.

Por otra parte, se define un bit por entrada del workpool y por UU que simboliza la finalización de un comando en una determinada UU. Al igual que el bit de presencia, el valor 1 indica finalización en pasadas pares y el valor 0 corresponde a la finalización en pasadas impares. De esta forma, cada bit es únicamente actualizado por una UU y consultado por la MU. El siguiente comando que debe leer una UU viene indicado por el registro iWP_UU.

La publicación de comandos requiere comprobar en primer lugar que el workpool no esté lleno y, posteriormente, almacenar el código de operación y los argumentos de los comandos. Por último, se activa el bit de presencia según el tipo de pasada por el workpool, indicada por

ITER_MU.

En aquellos comandos que precisan resultados se realiza una espera activa consultando todos los bits de finalización. Una vez finalizado el comando, la MU actualiza el número de UEs en estado CURRENT, AVAILABLE y FAILED, y el registro MODE.

Los comandos que se realizan de forma especial son los siguientes:

- UE-SWITCH: Las UUs actualizan en la propia ejecución del comando las estructuras de datos de la MU correspondientes al punto de indexación involucrado. Esta actualización requiere exclusión mutua, que se consigue con llamadas al sistema operativo. La MU debe esperarse a la finalización completa del comando para proseguir.
- UE-BACKWARD: Existen dos comandos de este tipo (-CURRENT y -SOLUTION) que son enviados siempre que se inicia la exploración en anchura de nuevas alternativas del árbol de búsqueda. En el momento de enviar este tipo de comandos, la MU conoce el número global de UEs en estado AVAILABLE que serán requeridos por las distintas UUs. La MU publica un vector con los identificadores de los UEs necesarios. Cada UU recoge los identificadores que necesita, con exclusión mutua, e inicializa los nuevos UEs mediante la técnica de copia de entornos. Estos nuevos UEs finalizan el comando en estado SOLUTION, y los UEs a los que iba dirigido el comando pasan a estado CURRENT.
- UE-SINGLE: Con este nombre se engloba a todos aquellos comandos que se envían cuando sólo existe un único UE en estado CURRENT. El código de operación de estos comandos siempre incluye el substring "-1". Estos comandos los ejecuta directamente la MU, tras esperar la finalización de cualquier comando precedente.

8.1.3 Distribución de la carga

Al principio de la ejecución, existe un único UE activo (en estado CURRENT) que está asignado a una UU. El número de UEs activos a lo largo de la ejecución se modifica por varias causas.

En primer lugar, se incrementa el número de UEs activos cuando se realiza la operación de Avance. En este momento, se reúnen todas las soluciones obtenidas y los UEs que las gestionan pasan a ser activos. Por otra parte, en la operación de Retroceso a un punto de selección pasan a ser activos los UEs que ya existían en el momento de crear este punto de selección. Obsérvese que en el momento de encontrar una solución pueden crearse nuevos UEs, pero no son activos ya que pasan a estado SOLUTION, esperando la obtención de nuevas

soluciones. Por último, siempre que se realiza una Unificación o una Indexación, existe la posibilidad de obtener un fracaso en algún UE y, por tanto, disminuye el número de UEs activos.

El algoritmo de distribución de la carga en Multipath se ejecuta en el momento de reunir todas las soluciones en la operación de Avance. Hasta este momento, cada UE en estado SOLUTION estaba asignado a la UU que había encontrado la solución. El algoritmo de distribución asigna de forma equitativa todos los UEs que pasan a ser activos entre todas las UUs. La distribución de UEs a UUs se establece aplicando un criterio que favorece la localidad. En concreto, todas las UEs que ya estaban asignadas a una UU continúan asignadas a la misma UU. Todas aquellas UEs que sobrepasan el número medio son recogidas por las UUs que no disponían de suficientes, mediante exclusión mutua.

Cuando se retrocede a un punto de selección no es necesario distribuir la carga, ya que se recupera el mismo número de UEs y la misma asignación que existía en el momento de su creación.

Se ha comprobado experimentalmente que la carga de todas las UUs es aproximadamente la misma aplicando este algoritmo de distribución. Por tanto, no es necesario distribuir la carga de forma más frecuente, a medida que los UEs van fracasando durante la ejecución de unificaciones. Además, una aplicación más frecuente de este algoritmo decrementaría la localidad en el acceso a los datos.

8.2 Rendimiento de PMAM frente a SMAM

Se han ejecutado los mismos benchmarks utilizados en la realización secuencial de Multipath. El tiempo de ejecución obtenido en PMAM se muestra en la figura 8.4. Para cada benchmark, se indica el tiempo requerido para ejecutarlo variando el grado de exploración parcial en anchura (desde 1 hasta 1000 UEs) y el número de unidades de unificación (desde 1 hasta 7 UUs). También se visualiza el tiempo de ejecución de la realización secuencial en este sistema (SMAM).

El comportamiento de un programa se caracteriza en función de varios parámetros que marcan la posible ganancia y/o pérdida de la realización paralela respecto la secuencial. En primer lugar, la ganancia con respecto a la versión secuencial es función de la **tasa de indeterminismo** de un programa. Cuanto más indeterminista sea un programa mayor será la cantidad de proceso paralelo que podrá explotar el programa. Recuérdese que este parámetro representa el cociente entre el número medio de UEs activas respecto el número máximo de UEs, esto es NUM_UE. Tal como se ha descrito en el capítulo anterior, el programa *bitspal* exhibe la mayor

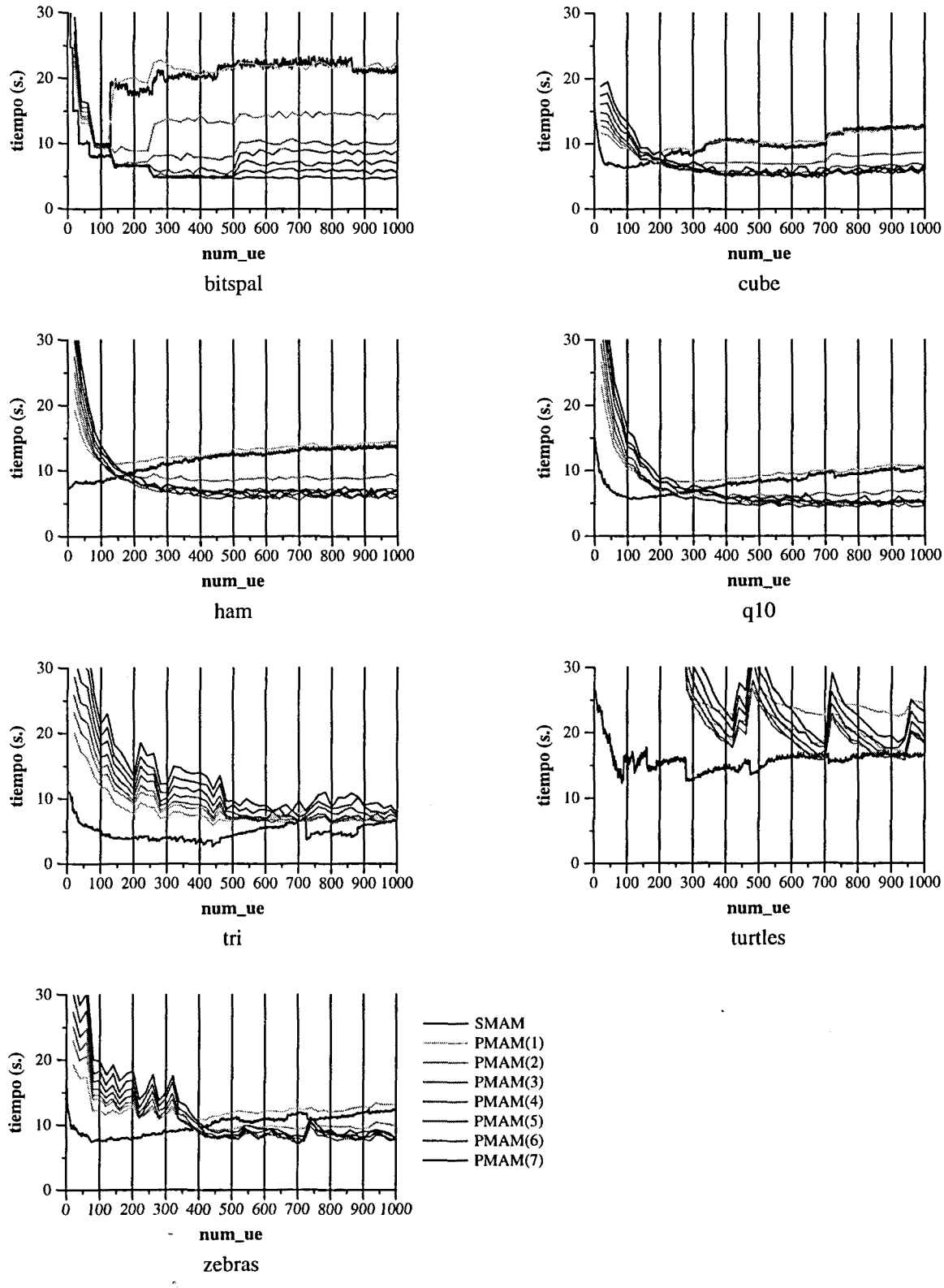


Figura 8.4: *Tiempo de ejecución de PMAM.*

tasa de indeterminismo (alrededor de un 60%) mientras que el resto oscila entre el 2% y el 10%.

Por otra parte, hay que tener en cuenta los factores que penalizan la ejecución paralela de Multipath. Existen tres motivos básicos que disminuyen la eficiencia de esta realización:

- **Penalización por conflictos en la red de interconexión con memoria.** La realización de Multipath en un sistema multiprocesador con memoria compartida incrementa el peso de este factor. Téngase en cuenta que el modelo arquitectónico necesita una gran cantidad de estructuras de datos residentes en memoria, hecho característico de los sistemas orientados a Prolog [114]. Además, este factor se ve aumentado por la técnica de inicialización de UEs basada en la copia de entornos.
- **Penalización por la publicación de comandos.** Este factor introduce un considerable aumento del tiempo de ejecución debido a la gestión del workpool para enviar y recibir un comando. Téngase en cuenta que en el envío de un comando debe comprobarse si el workpool no está lleno, guardar sus parámetros y su código de operación e indicar la presencia de un nuevo comando en el bit de sincronización. Estas acciones, así como sus análogas para recibir un comando, notificar su finalización y recoger los posibles resultados se realizan exclusivamente por software.
- **Penalización por la sincronización.** La sincronización constituye un factor de penalización intrínseco en el propio modelo arquitectónico. El ME debe esperarse a la finalización en un subconjunto de los comandos para poder disponer de los resultados que proporcionan.

Estos factores de penalización se ponen de manifiesto en mayor medida en aquellos programas con un bajo nivel de paralelismo. Por ejemplo, en los programas *tri*, *turtles* y *zebras*, con una tasa de indeterminismo relativamente pequeña, se envían un elevado número de comandos y, por tanto, la penalización por la publicación de estos comandos es más significativa que la cantidad de trabajo en paralelo a realizar. En estos benchmarks se observa un incremento del tiempo de ejecución conforme aumentamos el número de UUs para un mismo nivel de exploración en anchura.

En los benchmarks *cube*, *ham* y *q10* existe una ganancia con respecto a la versión secuencial pero queda limitada en mayor medida por el tiempo gastado en sincronización.

El programa en que se obtiene un mayor rendimiento es *bitspal*. Posee el mayor nivel de paralelismo y además los inconvenientes anteriormente indicados se reflejan en menor medida. Durante la ejecución se envían un menor número de comandos; a su vez, existe un porcentaje

mayor de comandos que no requieren sincronización; y, por último, es el programa que presenta un número de copias despreciable.

Otro aspecto importante que se refleja de las gráficas corresponde al número óptimo de UEs en cada benchmark. En la ejecución paralela este parámetro se incrementa respecto al obtenido en la ejecución secuencial.

8.3 Mejoras en la sincronización

La sincronización que se requiere efectuar durante la ejecución es un factor, introducido en el propio modelo arquitectónico, que penaliza el rendimiento del sistema. Con la intención de mejorar el comportamiento de la versión paralela de Multipath se han propuesto dos optimizaciones que afectan a la sincronización que se realiza en los comandos que generan resultados. Estas dos optimizaciones consisten en:

- La sincronización en un comando de unificación, que necesita como respuesta saber si tiene éxito o fracasa en todos los UEs activos, **espera a que se obtenga un número mínimo de UEs cuyo estado final sea CURRENT**. Este número mínimo debe garantizar que el flujo de control tras la ejecución del comando va a ser el mismo que una vez finalizado completamente el comando.
- En un comando de unificación no se realiza sincronización, sino que la MU continúa con una **ejecución especulativa** del código que le sigue, considerando que la unificación tiene éxito en algún UE.

La primera propuesta tiene en cuenta que, una vez se han obtenido 2 UEs en estado CURRENT, el control de flujo siempre va a proseguir por las mismas instrucciones. En concreto, el primer UE garantiza que no se produce fracaso en el comando y el segundo UE garantiza que nunca serán enviados comandos del tipo UE-SINGLE. La aplicación de esta técnica precisa la gestión de 3 variables en la MU que especifican si el número de UEs en estado CURRENT, FAILED o AVAILABLE contienen el valor correctamente actualizado o no. Siempre que la MU necesita consultar en un momento posterior esta información, espera la finalización del último comando publicado que la puede modificar. En este tipo de sincronización, la MU considera que el modo de unificación en comandos no finalizados completamente corresponde al caso general READ/WRITE, es decir, no puede optimizar los comandos posteriores de unificación de elementos de términos estructurados ya que no sabe si todos los UEs se encuentran en modo READ o en modo WRITE. Con referencia al workpool, se necesita introducir un bit por comando y por UU que especifica si existen como mínimo dos UEs que han finalizado con

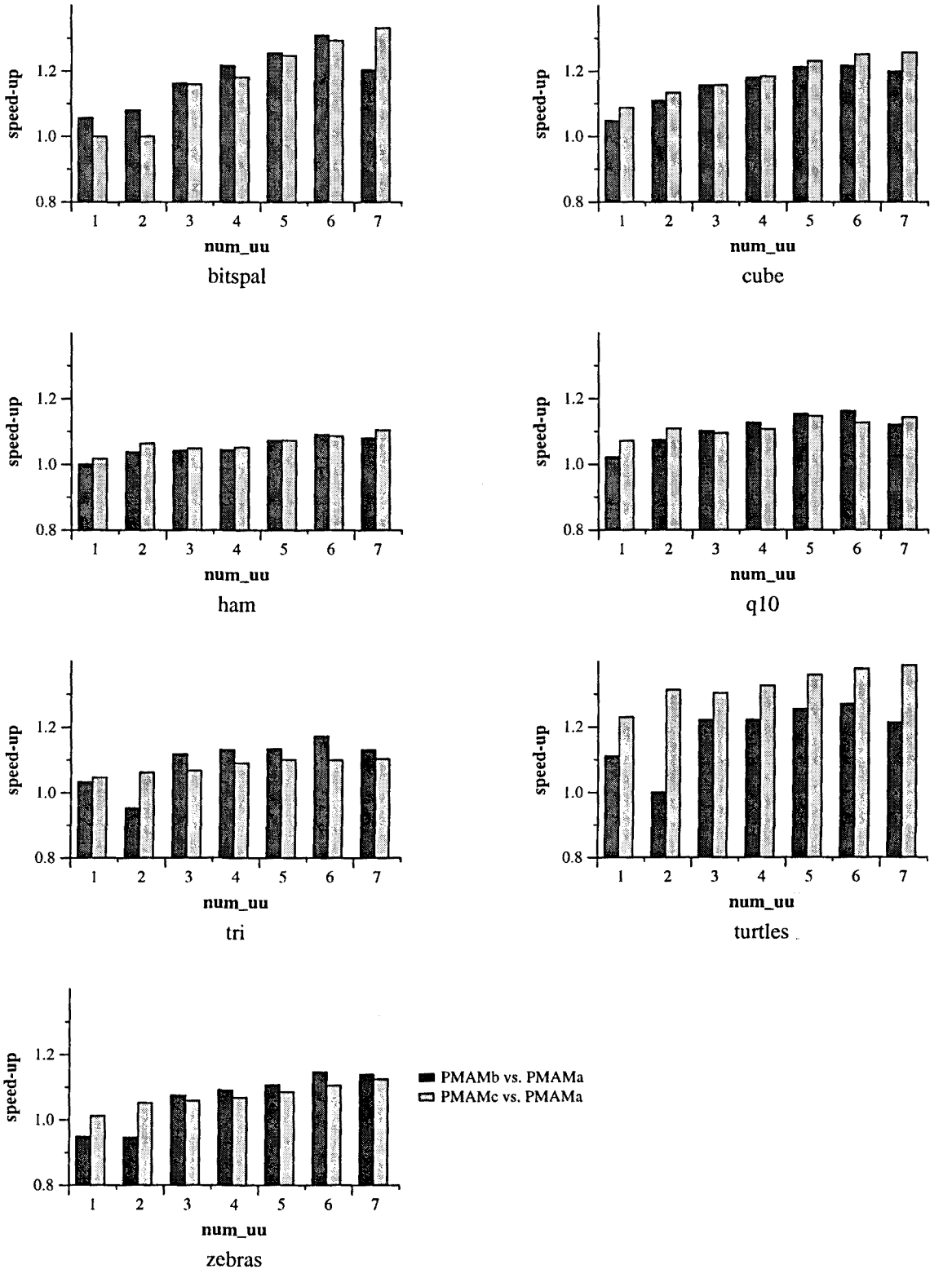


Figura 8.5: *Speed-up de PMAMb y PMAMc frente a PMAMa.*

éxito la ejecución de un comando en una UU.

La segunda propuesta se basa en la mayor probabilidad de que un comando de unificación finalice con éxito. En las sincronizaciones anteriores existen tres continuaciones diferentes tras un comando de unificación (fracaso, éxito con 1 UE o éxito con más de un UE). La ejecución especulativa considera únicamente dos continuaciones (fracaso o éxito). De esta forma se aumenta la probabilidad de elegir la alternativa correcta y también se simplifica la gestión que requiere. No obstante, se pierde la posibilidad de efectuar las optimizaciones que podían realizarse cuando se exploraba un único camino. En este sentido, un objetivo con atributo SEMIDET siempre será explorado en anchura y el tipo dinámico de una variable que depende del número de vínculos de otra variable siempre será MULTIPLE.

Por otra parte, la ejecución especulativa prosigue hasta aquel punto en que restaurar el estado de la ejecución en una predicción incorrecta es demasiado complejo. En concreto, se efectúa una espera completa en aquellos puntos en que se eliminan estructuras de datos de la MAM o se crean estructuras de datos nuevas que no son eliminadas en la operación de backtracking. Con esta condición, restaurar el estado de la ejecución en una predicción incorrecta significa que únicamente hay que deshacer, en la propia operación de backtracking, todos los vínculos que hayan podido ser calculados en una ejecución innecesaria. Esta alternativa para mejorar la penalización por sincronización también incorpora una consulta a cada instrucción MAM de la finalización de comandos pendientes. Esta comprobación permite deshacer los trabajos realizados en una predicción incorrecta lo antes posible.

La figura 8.5 muestra la ganancia, para cada benchmark y para un número de UUs entre 1 y 7, de las dos anteriores propuestas con respecto la sincronización descrita inicialmente en PMAM (en la gráfica es referenciada como PMAMa). La espera de un número mínimo de UEs se simboliza como PMAMb y la versión con ejecución especulativa se simboliza como PMAMc. El valor de speed-up corresponde a la ganancia media calculada a partir de los tiempos de ejecución obtenidos al variar NUM_UE en el rango [1, 1000]. Se puede observar una ganancia real en las dos versiones propuestas, que puede llegar hasta más de un 20%. Por otro lado, no existe una diferencia realmente significativa entre las dos alternativas. Esto es debido a que la cantidad de trabajo ejecutado de forma especulativa hasta tener que hacer una sincronización completa no es lo suficientemente grande.

De todas formas, existe un mayor número de casos en que la ejecución especulativa tiene un mejor comportamiento que la espera parcial, en especial, cuando se aumenta el número de UUs. Ello nos ha llevado a escogerla como la forma de realizar la sincronización en Multipath.

8.4 Rendimiento global de Multipath

En esta sección se analiza el rendimiento de Multipath respecto la ejecución convencional simbolizada por la WAM. Hay que tener en cuenta que la ganancia de Multipath en una ejecución paralela se ve afectada por la ganancia obtenida por pasar del modelo convencional a la ejecución secuencial y por la ganancia al pasar de una ejecución secuencial a una ejecución paralela de Multipath.

La figura 8.6 muestra el efecto individual de cada uno de estos dos factores para los benchmarks analizados. En esta gráfica se considera un número concreto de UEs (700) y un número también concreto de UUs (5). Se observa que el rendimiento en secuencial sigue aproximadamente el mismo comportamiento visto en el capítulo anterior. Téngase en cuenta que en esta gráfica no se contempla el número óptimo de UEs sino un valor fijo para todos los benchmarks. El rendimiento en paralelo oscila con speed-ups entre 1 y 4, según el benchmark.

Obsérvese que las características dinámicas de cada benchmark en cuanto a tasa de indeterminismo, número de copias efectuadas y proporción de comandos enviados por instrucción influyen en el modelo de ejecución que se le adapta mejor. Por ello, se contempla la posibilidad de utilizar la misma técnica descrita en el capítulo anterior para la ejecución secuencial, de cara a poder determinar de forma automática el modelo que mejor se adapta a cada benchmark, es decir, la WAM, una ejecución secuencial de Multipath o una ejecución paralela de Multipath. En base al comportamiento homogéneo del árbol de búsqueda en cada programa, es posible

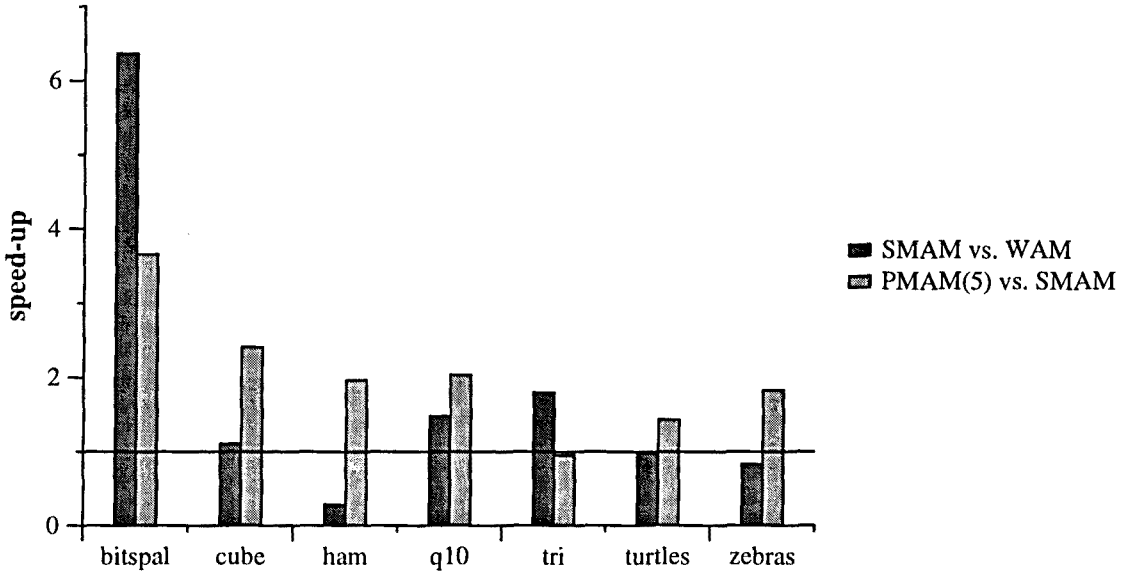


Figura 8.6: *Speed-up de WAM a SMAM y de SMAM a PMAM.*

realizar un preproceso en el que a partir del tiempo en ejecutar una misma parte inicial del árbol de búsqueda en cada modelo es capaz de determinar el que tiene una ejecución más rápida. A diferencia de la realización secuencial, este preproceso no se ha programado en la realización paralela y, en consecuencia, los datos de Multipath que siguen a continuación se han calculado después de ejecutar los benchmarks para todas las posibles configuraciones.

La figura 8.7 muestra el speed-up global de la realización paralela de Multipath respecto la WAM. Se observa como existen programas (*ham*) en que el modelo WAM ofrece el mejor rendimiento; programas (*tri*) en que el mejor resultado se obtiene mediante una ejecución secuencial de la MAM; y el resto de casos en que la ejecución paralela de la MAM ofrece el mejor resultado. Dentro de este último grupo, los programas *turtles* y *zebras* ofrecen una ganancia del 25% respecto la WAM; los programas *cube* y *q10* tienen un crecimiento lineal hasta 5 UUs del speed-up, llegando a valores alrededor de 4; y en el programa *bitspal* se obtienen speed-ups realmente significativos desde 18 hasta 45, variando el número de UUs desde 1 hasta 7, respectivamente.

Por último, se resumen en la tabla 8.1 los datos más característicos de esta versión de Multipath para cada uno de los benchmarks analizados. En primer lugar, se indica el tiempo de ejecución WAM; el tiempo de ejecución de SMAM (para un valor fijo de UEs igual a 700

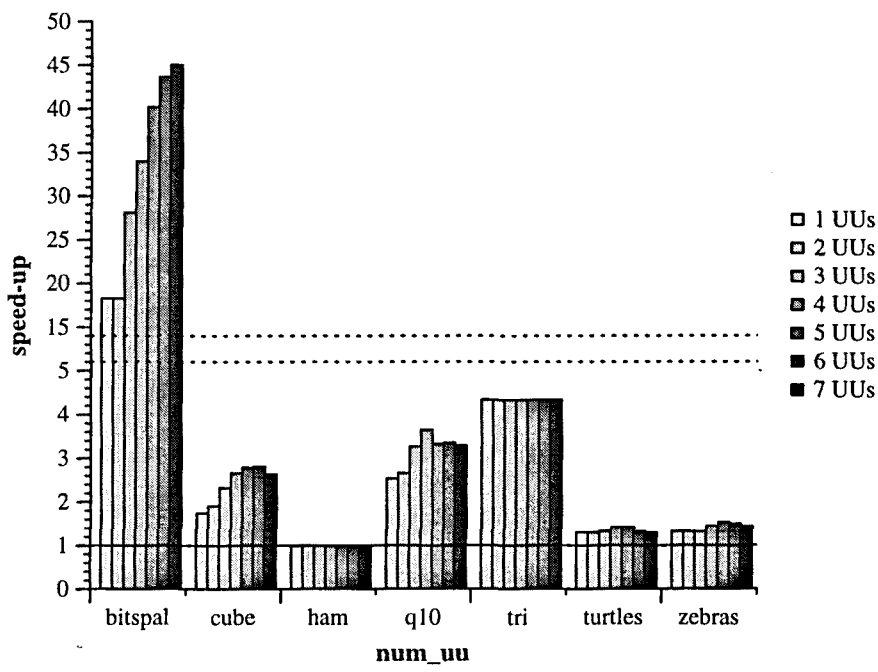


Figura 8.7: *Speed-up global de Multipath frente a WAM.*

	bitspal	cube	ham	q10	tri	turtles	zebras
Tiempo WAM (s)	142.6	11.0	**3.8**	14.2	11.7	16.0	9.8
Tiempo SMAM-700 (s)	22.4	9.9	13.1	9.6	6.5	16.2	11.7
Num_ue óptimo SMAM	88	86	2	112	438	84	77
Tiempo SMAM-opt (s)	7.8	6.3	7.3	5.6	**2.7**	12.3	7.4
Tiempo PMAM5-700 (s)	6.12	4.10	6.66	4.70	6.81	11.29	6.38
Num_ue óptimo PMAM5	500	660	920	720	1000	700	700
Tiempo PMAM5-opt (s)	**3.54**	**3.94**	5.63	**4.25**	5.78	**11.29**	**6.38**
Speed-up MULTIPATH	**40.28**	**2.79**	**1.00**	**3.34**	**4.33**	**1.42**	**1.54**
Instr. MAM-700	406769	218494	109163	230235	545431	690474	276328
Instr. MAM-700 por ejec. espec.	2068	1318	1913	31	182480	91760	25840
Comandos MAM-700.	60555	59651	61677	62186	252395	411751	189582
Sincronizaciones PMAM5-700	8252	24979	30870	24405	8521	147356	30102
Núm. medio UEs PMAM5-700	507.82	64.00	45.07	65.19	26.24	22.99	34.26

Tabla 8.1: *Resumen de los datos más significativos de la ejecución paralela de Multipath.*

y para el valor óptimo), y el tiempo de ejecución de PMAM con 5 UUs (también para los dos casos anteriores en cuanto a número de UEs). El speed-up de Multipath respecto la ejecución convencional se muestra a continuación. El cálculo de este valor tiene en cuenta la posibilidad de detectar el modelo que mejor se adapta a cada benchmark. Posteriormente, se indican algunos datos estadísticos que influyen en el rendimiento de Multipath, para un valor fijo de NUM_UE igual a 700. En concreto, se muestra el número total de instrucciones ejecutadas; las instrucciones ejecutadas de forma especulativa correspondientes a predicciones incorrectas; el número de comandos publicados; el número total de sincronizaciones en la ejecución del programa y el número medio de UEs activas, dato a partir del cual se calcula la tasa de indeterminismo del programa.

8.5 Mejoras hardware

Se ha observado que Multipath posee un rendimiento excelente en aquellos programas con un alto grado de indeterminismo. Para lograr un mayor rendimiento en aquellos programas que no exhiben tanto indeterminismo es preciso introducir en el hardware del sistema elementos espe-

cíficos que disminuyan las penalizaciones que se producen en una interpretación estrictamente software.

En relación con la penalización observada por la gestión del workpool de comandos, se hace totalmente necesaria la definición de registros hardware e instrucciones específicas en el lenguaje máquina de manipulación del workpool.

Por otra parte, sería recomendable evitar las esperas activas de sincronización mediante la gestión de interrupciones hardware que indicasen la finalización de los comandos. Esto facilitaría que la MU pueda colaborar ejecutando comandos en aquellas UUs que estuvieran más cargadas.

En relación con los conflictos en el acceso a memoria, debe tenerse en cuenta que el modelo arquitectónico propone zonas de memoria locales a cada UE. Un multiprocesador con una parte local del espacio de direcciones en cada unidad de proceso reduciría los conflictos por accesos a las memorias CTRL_UE y TRAIL_UE. La memoria HEAP_UE podría asignarse al espacio local siempre que exista hardware de soporte a las copias durante la inicialización de estas memorias. De la misma forma que se ha descrito en la realización basada en un sistema

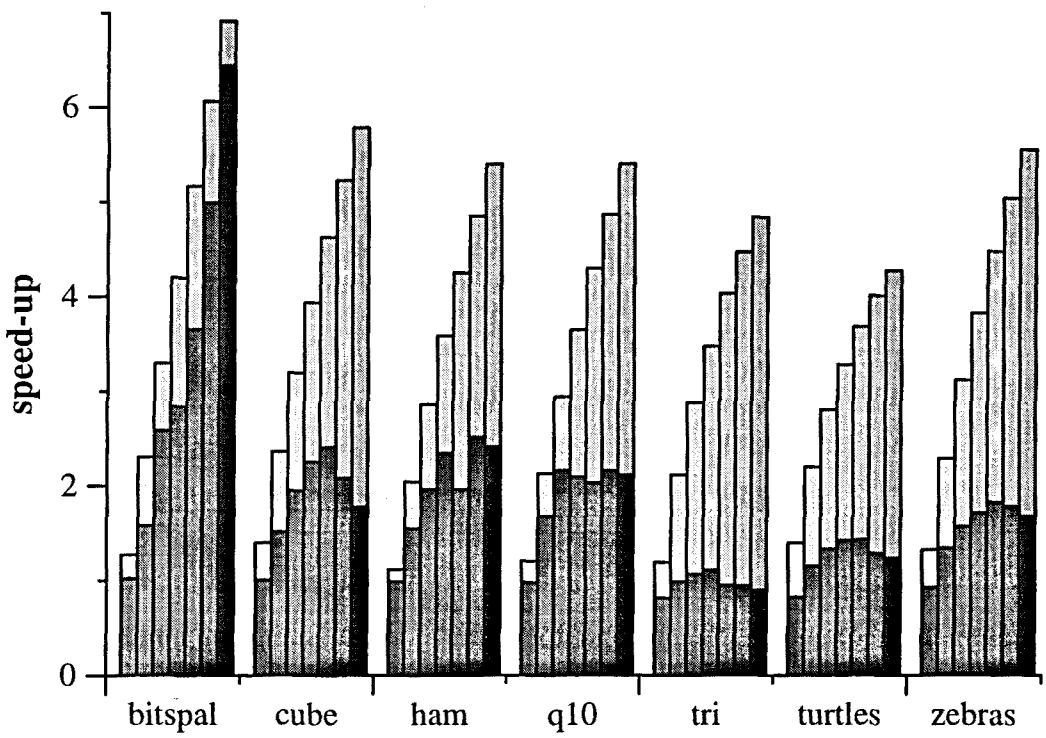


Figura 8.8: *Speed-up máximo alcanzable por Multipath en un arquitectura con hardware específico frente al speed-up real obtenido en PMAM. El speed-up se calcula respecto SMAM.*

monoprocesador, la existencia de una arquitectura orientada a ejecución multithreading también ayudaría a esconder la latencia en el acceso a memoria ya que todos las operaciones UE relativas a un mismo comando podrían ser fácilmente relacionadas con un thread del sistema.

Se ha simulado el comportamiento de Multipath eliminando las penalizaciones que se incurren en la versión actual de Multipath con el objetivo de conocer cuál es el límite en el rendimiento que se podría obtener con una arquitectura más adecuada al modelo [42]. En concreto, se ha supuesto que: (i) no existen conflictos en el acceso a memoria, (ii) la MU colabora en la ejecución de comandos cuando es preciso una sincronización, y (iii) no existe penalización en la publicación y lectura de los comandos en el workpool. La figura 8.8 muestra el speed-up límite que se puede conseguir frente al speed-up real que se ha conseguido en esta realización paralela de Multipath. Como puede observarse existe un gran potencialidad de aumentar el rendimiento de Multipath si se consideran arquitecturas más adecuadas que faciliten el modelo de proceso SPMD.

8.6 Resumen y contribuciones

En este capítulo se ha descrito y analizado la realización paralela de Multipath. Los aspectos específicos de mapear la Máquina Abstracta de Multipath en un entorno paralelo (PMAM) hacen referencia a la asignación de los motores de unificación a las unidades de proceso, la gestión de los comandos MAM y la distribución de la carga del sistema.

En base a los resultados obtenidos, las principales conclusiones son las siguientes:

- La ejecución paralela de MAM es recomendable en programas que exhiben un alto grado de indeterminismo.
- Los factores que degradan en mayor medida el rendimiento de PMAM son las esperas por sincronización, la penalización por la publicación y recepción de los comandos y los conflictos en la red de interconexión. Respecto a la sincronización, la propuesta de ejecución especulativa tiene una ganancia de un 20% sobre la espera de la finalización completa de comandos de unificación.
- Para los benchmarks en que es aconsejable la ejecución paralela, el speed-up global de Multipath respecto la WAM varía de 3.34 a 40.28 con 6 unidades de proceso.
- Se ha comprobado que existe una alta potencialidad para aumentar el rendimiento en una arquitectura paralela con hardware específico para el modelo de proceso SPMD.

9

CONCLUSIONES Y FUTUROS TRABAJOS

Este capítulo finaliza el trabajo realizado en esta tesis con una enumeración de las principales conclusiones a que se ha llegado y una descripción de las líneas futuras de trabajo relacionadas con el mismo tema.

9.1 Conclusiones

El diseño del sistema Multipath se ha realizado teniendo como principal punto de mira incrementar la eficiencia en la ejecución de aquellos programas que poseen un cierto grado de indeterminismo. Un programa presenta indeterminismo en el momento en que existen acciones, u objetivos, a ejecutar dentro del programa que tienen más de una solución.

La principal contribución de esta tesis es la definición de un nuevo modelo de ejecución que practica una exploración parcial en anchura del árbol de búsqueda de un programa indeterminista. La idea básica es ejecutar la continuación de un objetivo indeterminista simultáneamente para un cierto número de soluciones de dicho objetivo.

Los factores que intervienen positivamente respecto el recorrido tradicional, identificado por una exploración en profundidad, son

- La disminución del número de instrucciones necesarias para controlar el flujo de la ejecución y de instrucciones que operan sobre datos compartidos en todos los caminos recorridos simultáneamente.
- La posibilidad de explotar concurrencia entre las instrucciones que operan sobre datos locales a los caminos recorridos (paralelismo de caminos).

La segunda gran contribución viene marcada por la definición del modelo arquitectónico. Este modelo constituye una visión a más bajo nivel del modelo de ejecución, en la cual se adopta una forma de representación de los elementos que forman el estado de la ejecución y una forma de realización de las operaciones que lo transforman a lo largo de dicha ejecución. Este modelo deja abierta la posibilidad de realizar el sistema Multipath en diferentes plataformas hardware.

Anteriormente se han comentado las ventajas que ofrece a nivel lógico la exploración parcial en anchura. Es comprensible que su realización presente también una serie de inconvenientes. Los factores que intervienen negativamente en el rendimiento de Multipath son:

- Permitir la visibilidad en un momento dado de la ejecución del entorno de vínculos de más de un camino del árbol de búsqueda. Esta penalización se pone de manifiesto en la inicialización de los entornos mediante copias y en la mayor complejidad en la semántica de las instrucciones al tener que gestionar nuevas estructuras de datos y zonas de memoria.
- El aumento del espacio de memoria necesario en la ejecución, y la consiguiente pérdida de localidad en el acceso a los datos.

El rendimiento de la exploración parcial en anchura depende de las características propias de cada programa, que repercutirán en dar un mayor o menor peso a las ventajas y a los inconvenientes. Multipath establece de forma automática el grado óptimo de exploración parcial en anchura para cada programa.

El criterio utilizado se basa en un análisis global de determinismo y de tipos de datos con el fin de determinar los objetivos en que es recomendable el cálculo de más de una solución y de establecer a priori el tipo dinámico de las variables. En el primer caso, se ahorra la creación de estructuras de datos adicionales en la activación de aquellos objetivos que no manifiestan las ventajas de una exploración en anchura. En el segundo caso, el establecimiento del tipo de

las variables en el momento de su creación evita la penalización por el cambio dinámico de tipo a lo largo de su vida.

La determinación de la exploración parcial en anchura más adecuada se completa con una ejecución reducida de una primera parte del árbol de búsqueda para diferentes grados de exploración en anchura. El grado de exploración óptima viene determinado por aquél que ofrece un menor tiempo medio en recorrer un camino, e incluye la posibilidad de escoger, como un caso particular más, una exploración fija en profundidad realizada a partir de la WAM. En caso de un comportamiento heterogéneo del árbol de búsqueda, se propone modificar dinámicamente el grado de exploración en anchura hasta encontrar el que se adapta mejor a las características concretas del programa en un cierto momento de la ejecución.

A partir de los resultados obtenidos en una ejecución secuencial y en otra ejecución paralela de Multipath, se indican a continuación las principales conclusiones.

9.1.1 Rendimiento de una ejecución secuencial de Multipath

En una ejecución secuencial, la exploración parcial en anchura es, por regla general, más eficiente que la exploración en profundidad. Únicamente es aconsejable la exploración en profundidad en aquellos programas en que el número de copias de entornos es muy elevada. Multipath detecta estos casos mediante la ejecución reducida del principio del árbol de búsqueda.

El rendimiento de Multipath respecto la ejecución convencional identificada por la Máquina Abstracta de Warren es muy significativo. En el peor de los casos, es capaz de ejecutar un programa siguiendo el modelo convencional y, por tanto, no incrementa nunca el tiempo de ejecución de los programas. En el mejor caso de todos los programas analizados, se consigue un speed-up de más de un orden de magnitud.

Frente al sistema Multilog, que también realiza una exploración parcial en anchura, Multipath consigue reducir el tiempo de ejecución hasta una tercera parte.

Nótese que cuando se quieren encontrar todas las soluciones de un programa, su árbol de búsqueda es el mismo, independientemente del tipo de exploración. En este sentido, las ventajas de la exploración parcial en anchura provienen del hecho de reducir el número de instrucciones necesarias para recorrerlo. No obstante, cuando se quiere encontrar únicamente una solución, se recorre una parte diferente del árbol de búsqueda según el tipo de exploración. En estos casos, el rendimiento de la exploración parcial en anchura depende también de la proporción del árbol de búsqueda recorrida respecto la exploración en profundidad. Si las soluciones están uniformemente distribuidas en el árbol de búsqueda, Multipath ofrece un mayor

rendimiento. En el típico programa de las reinas, Multipath ofrece un speed-up de un orden de magnitud a la hora de encontrar la primera solución.

9.1.2 Rendimiento de una ejecución paralela de Multipath

En una ejecución paralela, la ventaja adicional de la exploración en anchura radica en la posibilidad de explotar el paralelismo de caminos, inherente en el modelo de ejecución. Este tipo de paralelismo es de baja granularidad ya que, básicamente, corresponde a la ejecución concurrente de comandos de unificación entre términos Prolog.

La ejecución paralela de Multipath se ha efectuado en un sistema multiprocesador de propósito general con memoria compartida. El rendimiento específico de la ejecución paralela respecto la secuencial es destacable en aquellos programas con una tasa elevada de indeterminismo. En los otros casos, la penalización por la publicación y recepción de los comandos paralelos, así como las esperas necesarias para la sincronización en espera de sus resultados, afectan en gran medida el rendimiento.

Se han propuesto técnicas que minimizan la penalización debida a la sincronización, basadas en una ejecución especulativa, que mejoran el rendimiento de Multipath en un 20%.

La simulación de Multipath en una arquitectura paralela orientada al proceso SPMD permite observar su alta potencialidad para obtener un rendimiento más elevado.

9.1.3 Efecto de la reducción del indeterminismo

Como un objetivo secundario, el modelo de ejecución de Multipath también presenta técnicas de disminución del grado de indeterminismo de un programa. Siempre que sea posible, es conveniente reducir el árbol de búsqueda lo antes posible. Nótese que no siempre es posible convertir un programa indeterminista en otro determinista. En estas situaciones es cuando la exploración parcial en anchura practicada en Multipath se ofrece una como una alternativa totalmente recomendable.

Multipath presenta tres técnicas de reducción del árbol de búsqueda, que se simbolizan como transformaciones a realizar en el código de entrada. Estas tres técnicas se caracterizan por el momento en que se reduce el árbol de búsqueda: de forma estática, en el momento de activar un objetivo y en el momento en que se han satisfecho objetivos del cuerpo de una cláusula. La reducción del indeterminismo en Multipath es una aplicación directa de los resultados obtenidos en el análisis global de determinismo y de tipos de datos, que ya es necesario realizar para optimizar la exploración parcial en anchura.

En el conjunto de benchmarks analizados, el porcentaje de reducción del número de instrucciones a ejecutar oscila entre el 5% y el 35%.

9.2 Líneas futuras de trabajo

El diseño de Multipath constituye un trabajo muy general en el que se aborda el *gap*, o diferencia semántica, entre un lenguaje declarativo y el lenguaje máquina de un computador. En este sentido, existen gran cantidad de aspectos que requieren una investigación más profunda. A continuación se describen las principales líneas de investigación que, personalmente, considero más atractivas.

9.2.1 Incorporación de nuevos elementos declarativos en el lenguaje

La definición del lenguaje al que está orientado un sistema constituye la primera gran decisión. Un lenguaje imperativo está más cerca del lenguaje máquina y, por tanto, es el más adecuado en aquellas aplicaciones de cálculo intensivo que requieren un código determinista lo más eficiente posible. Sin embargo, si se prima la eficacia en la fase de desarrollo de software es imprescindible incorporar en el lenguaje elementos declarativos, que facilitan la especificación de qué es lo que se desea obtener en vez de cómo obtenerlo. En el campo de la inteligencia artificial, el establecimiento de potentes lenguajes declarativos es crucial de cara a facilitar el aprendizaje del conocimiento y la resolución automática de problemas.

Prolog constituye un intento de elevar la capacidad declarativa del lenguaje: unificación, indeterminismo y variables de asignación única son su principales elementos declarativos. En Multipath se ha elegido Prolog ya que permite la posibilidad de especificar indeterminismo en un algoritmo. Resulta más cómodo poder especificar que un procedimiento puede tener más de una solución que no tener que tratar explícitamente en un bucle el cálculo de cada solución del procedimiento y, posteriormente, comprobar si es una solución global del programa. Por otra parte, al programar completamente el flujo de control en un lenguaje imperativo se pierde la visión de la funcionalidad del código y, por tanto, de la posibilidad de realizar transformaciones automáticas del código.

Sin embargo, existen otros aspectos declarativos que deben ser introducidos en el lenguaje. Para que un lenguaje sea utilizado ampliamente, situación que no se produce actualmente con el lenguaje Prolog, debe ser lo suficientemente flexible para especificar cómodamente cualquier tipo de algoritmo y existir potentes modelos de ejecución que sean capaces de traducir eficazmente la diferencia semántica respecto al lenguaje máquina de un computador de propósito

general.

Como investigador en el área de arquitectura y tecnología de computadores, creo más interesante partir de elementos declarativos particulares. Lo importante es poder automatizar el uso del modelo de ejecución más apropiado en cada caso y saber cuantificar el coste/rendimiento de introducir nuevos elementos arquitectónicos en el lenguaje máquina del computador.

9.2.1.1 Paradigma CLP

El paradigma de la programación lógica con restricciones (CLP) incorpora un nuevo elemento declarativo: permite especificar las propiedades que deben cumplir las variables, en lugar de tener que especificar el flujo de control necesario para calcular su valor.

Este paradigma constituye el principal punto de partida en trabajos posteriores relacionados con Multipath. Una de las mayores ventajas de la contribución principal de Multipath, la exploración parcial en anchura, radica en que no está ligada forzosamente a un lenguaje particular. Se puede aplicar siempre que exista indeterminismo en un programa, y el paradigma de CLP también admite esta posibilidad [127].

9.2.2 Generalización del modelo de ejecución

El modelo de ejecución es el encargado de especificar cómo debe ejecutarse un programa, es decir, determina el flujo de control que queda invisible al programador cuando utiliza un lenguaje con elementos declarativos. Como característica primordial de un modelo de ejecución, éste debe estar abierto a la posibilidad de adoptar diferentes estrategias y seleccionar la más adecuada para cada programa. Otra de las principales cualidades de un modelo de ejecución es la poder definir más de una fase en la interpretación de un programa. La realimentación de información del comportamiento de un programa es vital para poder optimizar ejecuciones posteriores. No siempre es posible encontrar un modelo matemático que de forma estática y determinista indique el flujo de control óptimo.

Multipath aplica esta idea con varias fases en la interpretación de un programa Prolog: abstracta, reducida y real. Cada fase tiene como objetivo proporcionar información que permite ejecuciones posteriores más eficientes. Sin embargo, el modelo de ejecución de Multipath se queda ceñido a la gestión del indeterminismo y la posible aplicación de una búsqueda fija en profundidad o una búsqueda parcial en anchura y, en este último caso, la explotación del paralelismo de caminos.

Dentro de este campo de trabajo, englobado como generalización del modelo de ejecución,

se proponen dos líneas que pueden atacarse independientemente o de forma conjunta.

9.2.2.1 Combinación de paralelismo de caminos con paralelismo O

Una de las fuentes de paralelismo más atrayentes en Prolog es el paralelismo O. En este punto, la principal línea de trabajo futura es la incorporación en el modelo de ejecución de la posibilidad de combinar el paralelismo de caminos con el paralelismo O.

Ambos tipos de paralelismo tienen una gestión común en muchos puntos. El problema de permitir la visibilidad de más de un entorno de vínculos también se presenta en el paralelismo O, y ya se encuentra solucionado en Multipath. Por otra parte, el paralelismo O posee una granularidad más gruesa que el paralelismo de caminos y sería recomendable su aplicación en aquellas unidades de proceso que no estén fuertemente acopladas. Es más adecuado explotar el paralelismo de caminos cuando las unidades de unificación están fuertemente acopladas con la unidad de proceso principal.

9.2.2.2 Ejecución de objetivos fuera de orden

Multipath presenta la misma regla de selección de objetivos que la ejecución convencional de Prolog. Los objetivos se activan en el mismo orden textual de escritura. Sin embargo, la ejecución de objetivos fuera de orden presenta también ventajas.

La posibilidad de ejecutar en primer lugar los objetivos deterministas frente a los indeterministas permite reducir el árbol de búsqueda de un programa. La gestión de este tipo de flujo de control ya se ha aplicado en el sistema Andorra-I [97] con resultados positivos. Este tipo de ejecución es un buen candidato a integrar y mejorar en Multipath.

Respecto a la posibilidad de aceptar el paradigma CLP como lenguaje del sistema, es necesario definir un modelo de ejecución que integre la ejecución retrasada de objetivos. En estos sistemas [81], la activación de los objetivos no se realiza en el ordèn textual del programa sino en el momento en que se cumplen las condiciones para su evaluación, es decir, el proceso es conducido por los datos.

9.2.3 Optimización del modelo arquitectónico

El modelo arquitectónico de Multipath permite la visualización simultánea de múltiples caminos del árbol de búsqueda con un método simple, pero costoso, como es la inicialización de entornos mediante copia. La principal ventaja reside en que el acceso al vínculo de cualquier variable requiere un único acceso a su entorno. La inicialización de motores es costosa ya que requiere

copiar todos los vínculos de variables múltiples del entorno padre, aun cuando existan variables que nunca serán referenciadas en el entorno hijo.

Aunque Multipath presenta un substancial incremento de rendimiento respecto la ejecución convencional, alrededor de la mitad del tiempo de la ejecución de la mayoría de los programas se gasta en copias de entornos.

9.2.3.1 Compartición de los entornos de vínculos

La principal línea de investigación en este aspecto radica en permitir la compartición de entornos entre los motores de la Máquina Abstracta de Multipath y gestionar dinámicamente el acceso al entorno de vínculos. En concreto, la inicialización de entornos no requeriría copia, pero el acceso a los vínculos de las variables que se referencien supondría una búsqueda en todos los entornos padre [43].

Este método implica dividir el rango de direcciones de un entorno en dos partes: la pendiente de inicializar y la parte creada posteriormente a la inicialización del motor. Para el primer espacio de direcciones es conveniente saber si necesita obtenerse el vínculo de un variable de los entornos padre o si ha sido referenciado anteriormente y, por tanto, contiene ya el vínculo correcto. Esta optimización de copia en demanda de vínculos de un entorno padre permite acelerar accesos posteriores a la misma variable.

9.2.4 Finalización y mejora de la realización secuencial

La realización secuencial de Multipath se basa en la emulación de las instrucciones del modelo arquitectónico mediante un lenguaje imperativo soportado por el sistema. Por otra parte, el análisis del rendimiento del sistema se ha concentrado únicamente en la fase de ejecución real de un programa. Asimismo, la posibilidad de ejecución concurrente que proporcionan los comandos definidos en el modelo arquitectónico se traduce en la programación de un bucle que ejecuta secuencialmente la tarea de un comando para cada motor.

En referencia a todos estos aspectos, se plantean diferentes alternativas a seguir como líneas futuras de trabajo, que se citan a continuación.

9.2.4.1 Programación de los algoritmos de la interpretación abstracta

La interpretación real de un programa se basa en información calculada durante la interpretación abstracta. En este trabajo se ha dado mayor prioridad al análisis del rendimiento de la ejecución real del programa frente a la ejecución abstracta. Por ello, únicamente se han establecido los

algoritmos y las estructuras de datos de la fase de interpretación abstracta que justifican la posibilidad de calcular de forma automática los resultados que se desean obtener. El hecho de ahorrar el tiempo requerido en el desarrollo del software de esta fase ha permitido concentrar los esfuerzos en la fase de ejecución real. En cualquier caso, la información que calcula siempre puede ser proporcionada por el programador.

Una vez demostrada la eficacia del modelo durante la ejecución real, se hace recomendable completar todo el sistema con la programación de la ejecución abstracta e investigar en aquellos aspectos que permitan disminuir su tiempo de ejecución.

9.2.4.2 Generación de código nativo

Otra forma de aumentar el rendimiento del sistema es generar código nativo en el lenguaje máquina del procesador. Actualmente, el intérprete C de la Máquina Abstracta de Multipath realiza la búsqueda, decodificación y ejecución de las instrucciones MAM.

Una primera optimización consistiría en expandir cada instrucción MAM de un programa en el código en lenguaje C equivalente.

A otro nivel de optimización, la traducción directa de las instrucciones MAM al lenguaje máquina del procesador permitiría efectuar optimizaciones que, con el intérprete actual escrito en lenguaje C, no son posible realizar. Téngase en cuenta que las instrucciones MAM tienen un alto nivel semántico y requieren aprovechar al máximo los recursos arquitectónicos del procesador. Sin embargo, con esta opción, se perdería la portabilidad del sistema Multipath a diferentes plataformas hardware.

9.2.4.3 Multipath en un procesador multithreading

Este es la línea de investigación que considero más atractiva en un futuro más inmediato. El código de los comandos de Multipath es independiente para cada motor de unificación y constituye una fuente potencial para establecer el flujo de control de los threads en una arquitectura multithreading. El principal objetivo en estas arquitecturas es esconder la latencia del acceso a la memoria por tener diversos threads preparados ya para el proceso de los datos, y que pueden pasar a ser ejecutados por la CPU en cualquier momento.

El estudio de los elementos hardware necesarios para minimizar la penalización introducida en la creación y eliminación de threads, así como en el cambio de contexto entre threads, es un trabajo a nivel de arquitectura de un computador, en el que Multipath puede ser considerado como una aplicación particular más a ejecutar en el computador. En este caso, se hace

viable la introducción de elementos hardware en la arquitectura de un computador, ya que éste no pierde la generalidad a que está aplicado.

9.2.5 Aumento de la eficacia de la ejecución paralela

La ejecución paralela de Multipath ha puesto de manifiesto la necesidad de añadir elementos hardware en la arquitectura de un computador paralelo con el objetivo de aumentar la eficacia en la ejecución de aplicaciones como Multipath. Estas aplicaciones se caracterizan por un modelo de proceso *Single Program Multiple Data* (SPMD). En el caso concreto de Multipath, el programa corresponde al código a ejecutar en cada comando y los datos pertenecen a los entornos de vínculos para los cuales se ejecuta un comando.

9.2.5.1 Soporte hardware para un modelo de proceso SPMD

Los aspectos específicos que introducen una mayor penalización son la gestión del workpool, las esperas debidas a la sincronización y los conflictos en la red de interconexión con memoria. En el capítulo 8 se han propuesto los elementos hardware que facilitarían este modelo de proceso.

Estos elementos forman la base para empezar una nueva línea de investigación relacionada con el sistema Multipath, pero lo suficientemente general como para ser aplicable en todos aquellos programas que presentan el mismo modelo de proceso.

Apéndice A

BENCHMARKS

Este apéndice contiene el código Prolog de los programas benchmark utilizados en el análisis del rendimiento del sistema Multipath.

BITSPAL

```
?- gen_list(L), bits(L), palindrome(L), fail.

gen_list([X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,
          Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9,Y10]).

palindrome(L):- rev(L,[],L).

rev([],R,R).
rev([H|T],Sofar,R):- rev(T,[H|Sofar],R).

bits([]).
bits([H|T]):- bits(T), bit(H).

bit(0). bit(1).
```

CUBE

```
?- sol(6,X), write(X), nl, fail.

sol(C,X) :- cubes(C,Q), sol(Q,[],X).

sol([],A,A).
sol([Q|Qs],A,F) :-
    set(Q,P),
    check(A,P),
    sol(Qs,[P|A],F).

check([],VAR).
check([q(A1,B1,C1,D1)|As],P) :-
    P = q(A2,B2,C2,D2),
    A1 \== A2, B1 \== B2, C1 \== C2, D1 \== D2,
    check(As,P).

set(q(P1,P2,P3),P) :- rotate(P1,P2,P).
set(q(P1,P2,P3),P) :- rotate(P2,P1,P).
set(q(P1,P2,P3),P) :- rotate(P1,P3,P).
set(q(P1,P2,P3),P) :- rotate(P3,P1,P).
set(q(P1,P2,P3),P) :- rotate(P2,P3,P).
set(q(P1,P2,P3),P) :- rotate(P3,P2,P).

rotate(p(C1,C2),p(C3,C4),q(C1,C2,C3,C4)).
rotate(p(C1,C2),p(C3,C4),q(C1,C2,C4,C3)).
rotate(p(C1,C2),p(C3,C4),q(C2,C1,C3,C4)).
rotate(p(C1,C2),p(C3,C4),q(C2,C1,C4,C3)).

cubes(4,[q(p(0,1),p(2,0),p(1,3)),
         q(p(3,3),p(2,0),p(1,2)),
         q(p(0,3),p(3,1),p(1,2)),
         q(p(0,0),p(3,0),p(1,2))]).
cubes(5,[q(p(2,1),p(1,4),p(3,1)),
         q(p(3,2),p(2,0),p(3,4)),
         q(p(1,4),p(3,1),p(0,4)),
         q(p(1,0),p(2,2),p(0,4)),
         q(p(4,2),p(4,3),p(0,3))]).
cubes(6,[q(p(0,5),p(1,5),p(3,1)),
         q(p(2,1),p(3,4),p(4,0)),
         q(p(3,0),p(4,5),p(2,4)),
         q(p(1,3),p(5,1),p(0,1)),
         q(p(0,2),p(0,2),p(5,2)),
         q(p(4,4),p(2,3),p(4,5))]).
cubes(7,[q(p(5,1),p(0,5),p(3,1)),
         q(p(2,3),p(1,4),p(4,0)),
         q(p(3,6),p(0,0),p(2,4)),
         q(p(6,4),p(6,1),p(0,1)),
```

```
        q(p(1,5),p(3,2),p(5,2)),
        q(p(5,0),p(2,3),p(4,5)),
        q(p(4,2),p(2,6),p(0,3))]).
```

Q10

```
?- queens2(10,Q), write(Q), nl, fail.

queens2(N,Qs) :-
        range(1,N,Ns),
        queens3(Ns,[],Qs).

range(N,N,[N]).
range(N,M,[N|T]) :-
        N < M,
        Np1 is N+1,
        range(Np1,M,T).

queens3([],Qs,Qs).
queens3(UnplacedQs,SafeQs,Qs) :-
        select(UnplacedQs,Remainder,Q),
        not_attack(SafeQs,Q),
        queens3(Remainder,[Q|SafeQs],Qs).

select([X|Xs],Xs,X).
select([Y|Ys],[Y|Zs],X) :-
        select(Ys,Zs,X).

not_attack(Xs,X) :-
        not_attack(Xs,X,1).

not_attack([],_,_).
not_attack([Y|Ys],X,N) :-
        YpN is Y+N,
        X =\= YpN,
        YmN is Y-N,
        X =\= YmN,
        N1 is N+1,
        not_attack(Ys,X,N1).
```

Q17_1

```
?- queens2(17,Q), write(Q), nl.
```

HAM

```
?- ham(X), write(X), nl, fail.

ham(X) :-
        cycle_ham([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t],X).
```

```
cycle_ham([X|Y],[X,T|L]):-
        chain_ham([X|Y],[],[T|L]),
        edge(T,X).

chain_ham([X],L,[X|L]).
chain_ham([X|Y],K,L):-
        delete(Z,Y,T),
        edge(X,Z),
        chain_ham([Z|T],[X|K],L).

delete(X,[X|Y],Y).
delete(X,[U|Y],[U|Z]):-
        delete(X,Y,Z).

edge(X,Y):-
        connect(X,L),
        el(Y,L).

el(X,[X|L]).
el(X,[Y|L]):-el(X,L).

connect(0,[1,2,3,4,5,6,7,8,9]).
connect(1,[0,2,3,4,5,6,7,8,9]).
connect(2,[0,1,3,4,5,6,7,8,9]).
connect(3,[0,1,2,4,5,6,7,8,9]).
connect(4,[0,1,2,3,5,6,7,8,9]).
connect(5,[0,1,2,3,4,6,7,8,9]).
connect(6,[0,1,2,3,4,5,7,8,9]).
connect(7,[0,1,2,3,4,5,6,8,9]).
connect(8,[0,1,2,3,4,5,6,7,9]).
connect(9,[0,1,2,3,4,5,6,7,8]).

connect(a,[b,j,k]).
connect(b,[a,c,p]).
connect(c,[b,d,l]).
connect(d,[c,e,q]).
connect(e,[d,f,m]).
connect(f,[e,g,r]).
connect(g,[f,h,n]).
connect(h,[i,g,s]).
connect(i,[j,h,o]).
connect(j,[a,i,t]).
connect(k,[o,l,a]).
connect(l,[k,m,c]).
connect(m,[l,n,e]).
connect(n,[m,o,g]).
connect(o,[n,k,i]).
connect(p,[b,q,t]).
connect(q,[p,r,d]).
```

```
connect(r,[q,s,f]).
connect(s,[r,t,h]).
connect(t,[p,s,j]).
```

TRI

```
main :- play(3,[1,1,1,1,1, 0,1,0,0,1, 1,1,1,0,1],X), write(X), nl, fail.

play(13,B,[]) :- !.
play(M,Board,[P|X]) :-
    move(P,Board,NewBoard),
    M1 is M+1,
    play(M1,NewBoard,X).

move(1,  [ 1, 1,X3, 0|R],
         [ 0, 0,X3, 1|R]).
move(2,  [X1, 1,X3, 1,X5,X6, 0|R],
         [X1, 0,X3, 0,X5,X6, 1|R]).
move(3,  [X1,X2,X3, 1,X5,X6, 1,X8,X9,X10,  0|R],
         [X1,X2,X3, 0,X5,X6, 0,X8,X9,X10,  1|R]).
move(4,  [X1,X2, 1,X4, 1,X6,X7, 0|R],
         [X1,X2, 0,X4, 0,X6,X7, 1|R]).
move(5,  [X1,X2,X3,X4, 1,X6,X7, 1,X9,X10,X11,  0|R],
         [X1,X2,X3,X4, 0,X6,X7, 0,X9,X10,X11,  1|R]).
move(6,  [X1,X2,X3,X4,X5, 1,X7,X8, 1,X10,X11,X12,  0|R],
         [X1,X2,X3,X4,X5, 0,X7,X8, 0,X10,X11,X12,  1|R]).
move(7,  [ 1,X2, 1,X4,X5, 0|R],
         [ 0,X2, 0,X4,X5, 1|R]).
move(8,  [X1,X2, 1,X4,X5, 1,X7,X8,X9,  0|R],
         [X1,X2, 0,X4,X5, 0,X7,X8,X9,  1|R]).
move(9,  [X1,X2,X3,X4,X5, 1,X7,X8,X9,  1,X11,X12,X13,X14,  0],
         [X1,X2,X3,X4,X5, 0,X7,X8,X9,  0,X11,X12,X13,X14,  1]).
move(10, [X1, 1,X3,X4, 1,X6,X7,X8, 0|R],
         [X1, 0,X3,X4, 0,X6,X7,X8, 1|R]).
move(11, [X1,X2,X3,X4, 1,X6,X7,X8, 1,X10,X11,X12,X13,  0|R],
         [X1,X2,X3,X4, 0,X6,X7,X8, 0,X10,X11,X12,X13,  1|R]).
move(12, [X1,X2,X3, 1,X5,X6,X7, 1,X9,X10,X11,X12,  0|R],
         [X1,X2,X3, 0,X5,X6,X7, 0,X9,X10,X11,X12,  1|R]).
move(13, [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,  1,  1,  0|R],
         [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,  0,  0,  1|R]).
move(14, [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,  1,  1,  0|R],
         [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,  0,  0,  1|R]).
move(15, [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,  1,  1,  0],
         [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,  0,  0,  1]).
move(16, [X1,X2,X3,X4,X5,X6, 1, 1, 0|R],
         [X1,X2,X3,X4,X5,X6, 0, 0, 1|R]).
move(17, [X1,X2,X3,X4,X5,X6,X7, 1, 1,  0|R],
         [X1,X2,X3,X4,X5,X6,X7, 0, 0,  1|R]).
move(18, [X1,X2,X3, 1, 1, 0|R],
```

```
        [X1,X2,X3, 0, 0, 1|R]).
move(19, [ 0, 1,X3, 1|R],
        [ 1, 0,X3, 0|R]).
move(20, [X1, 0,X3, 1,X5,X6, 1|R],
        [X1, 1,X3, 0,X5,X6, 0|R]).
move(21, [X1,X2,X3, 0,X5,X6, 1,X8,X9,X10,  1|R],
        [X1,X2,X3, 1,X5,X6, 0,X8,X9,X10,  0|R]).
move(22, [X1,X2, 0,X4, 1,X6,X7, 1|R],
        [X1,X2, 1,X4, 0,X6,X7, 0|R]).
move(23, [X1,X2,X3,X4, 0,X6,X7, 1,X9,X10,X11,  1|R],
        [X1,X2,X3,X4, 1,X6,X7, 0,X9,X10,X11,  0|R]).
move(24, [X1,X2,X3,X4,X5, 0,X7,X8, 1,X10,X11,X12,  1|R],
        [X1,X2,X3,X4,X5, 1,X7,X8, 0,X10,X11,X12,  0|R]).
move(25, [ 0,X2, 1,X4,X5, 1|R],
        [ 1,X2, 0,X4,X5, 0|R]).
move(26, [X1,X2, 0,X4,X5, 1,X7,X8,X9,  1|R],
        [X1,X2, 1,X4,X5, 0,X7,X8,X9,  0|R]).
move(27, [X1,X2,X3,X4,X5, 0,X7,X8,X9,  1,X11,X12,X13,X14,  1],
        [X1,X2,X3,X4,X5, 1,X7,X8,X9,  0,X11,X12,X13,X14,  0]).
move(28, [X1, 0,X3,X4, 1,X6,X7,X8, 1|R],
        [X1, 1,X3,X4, 0,X6,X7,X8, 0|R]).
move(29, [X1,X2,X3,X4, 0,X6,X7,X8, 1,X10,X11,X12,X13,  1|R],
        [X1,X2,X3,X4, 1,X6,X7,X8, 0,X10,X11,X12,X13,  0|R]).
move(30, [X1,X2,X3, 0,X5,X6,X7, 1,X9,X10,X11,X12,  1|R],
        [X1,X2,X3, 1,X5,X6,X7, 0,X9,X10,X11,X12,  0|R]).
move(31, [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,  0,  1,  1|R],
        [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,  1,  0,  0|R]).
move(32, [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,  0,  1,  1|R],
        [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,  1,  0,  0|R]).
move(33, [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,  0,  1,  1],
        [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,  1,  0,  0]).
move(34, [X1,X2,X3,X4,X5,X6, 0, 1, 1|R],
        [X1,X2,X3,X4,X5,X6, 1, 0, 0|R]).
move(35, [X1,X2,X3,X4,X5,X6,X7, 0, 1,  1|R],
        [X1,X2,X3,X4,X5,X6,X7, 1, 0,  0|R]).
move(36, [X1,X2,X3, 0, 1, 1|R],
        [X1,X2,X3, 1, 0, 0|R]).
```

TURTLES

```
?- turtles(Z), write(Z), nl, fail.

turtles([N01-T01, N02-T02, N03-T03, N04-T04,
        N05-T05, N06-T06, N07-T07, N08-T08,
        N09-T09, N10-T10, N11-T11, N12-T12]) :-

    Cards = [A11, A12, A13, A14,
            A21, A22, A23, A24,
            A31, A32, A33, A34],
```

```
        perm(T06,R06,B06,L06,N06), argl(N06,Cards,1),
        flip(T06,B02),
        perm(T02,R02,B02,L02,N02), argl(N02,Cards,2),
        flip(L06,R03),
        perm(T05,R05,B05,L05,N05), argl(N05,Cards,3),
        flip(L02,R01),
        flip(T05,B01),
        perm(T01,R01,B01,L01,N01), argl(N01,Cards,4),
        flip(R06,L07),
        perm(T07,R07,B07,L07,N07), argl(N07,Cards,5),
        flip(T07,B03),
        flip(R02,L03),
        perm(T03,R03,B03,L03,N03), argl(N03,Cards,6),
        flip(R07,L08),
        perm(T08,R08,B08,L08,N08), argl(N08,Cards,7),
        flip(T08,B04),
        flip(L04,R03),
        perm(T04,R04,B04,L04,N04), argl(N04,Cards,8),
        flip(B06,T10),
        perm(T10,R10,B10,L10,N10), argl(N10,Cards,9),
        flip(B05,T09),
        flip(L10,R09),
        perm(T09,R09,B09,L09,N09), argl(N09,Cards,10),
        flip(B07,T11),
        flip(R10,L11),
        perm(T11,R11,B11,L11,N11), argl(N11,Cards,11),
        flip(B08,T12),
        flip(R11,L12),
        perm(T12,R12,B12,L12,N12), argl(N12,Cards,12).

flip(tg,hg).  flip(tp,hp).  flip(to,ho).  flip(tb,hb).
flip(hg,tg).  flip(hp,tp).  flip(ho,to).  flip(hb,tb).

perm(tg,tp,hb,ho,1).
perm(tg,tp,hb,ho,2). perm(tp,hb,ho,tg,2). perm(hb,ho,tg,tp,2). perm(ho,tg,tp,hb,2).
perm(tb,tp,hb,ho,3). perm(tp,hb,ho,tb,3). perm(hb,ho,tb,tp,3). perm(ho,tb,tp,hb,3).
perm(tg,hb,hp,to,4). perm(hb,hp,to,tg,4). perm(hp,to,tg,hb,4). perm(to,tg,hb,hp,4).
perm(to,hb,hg,tp,5). perm(hb,hg,tp,to,5). perm(hg,tp,to,hb,5). perm(tp,to,hb,hg,5).
perm(hg,hp,tb,to,6). perm(hp,tb,to,hg,6). perm(tb,to,hg,hp,6). perm(to,hg,hp,tb,6).
perm(tb,tp,hg,ho,7). perm(tp,hg,ho,tb,7). perm(hg,ho,tb,tp,7). perm(ho,tb,tp,hg,7).
perm(tg,to,hg,hp,8). perm(to,hg,hp,tg,8). perm(hg,hp,tg,to,8). perm(hp,tg,to,hg,8).
perm(to,tp,hg,hb,9). perm(tp,hg,hb,to,9). perm(hg,hb,to,tp,9). perm(hb,to,tp,hg,9).
perm(ho,tp,tg,ho,10). perm(tp,tg,ho,ho,10). perm(tg,ho,ho,tp,10). perm(ho,ho,tp,tg,10).
perm(hb,hg,to,tp,11). perm(hg,to,tp,hb,11). perm(to,tp,hb,hg,11). perm(tp,hb,hg,to,11).
perm(to,hp,hg,tb,12). perm(hp,hg,tb,to,12). perm(hg,tb,to,hp,12). perm(tb,to,hp,hg,12).

argl(1,[X|Y],X).
argl(N,[A|L],X) :- N > 1, N1 is N-1, argl(N1,L,X).
```

ZEBRAS

```
?- solve(Houses), print(Houses), nl, fail.

solve(Houses) :-
  init(Houses),
  nextTo(house(ivory,A1,A2,A3,chesterfields), house(B1,B2,fox,B3,B4), Houses),
  nextTo(house(C1,japanese,C2,C3,parliaments), house(purple,D1,D2,D3,D4), Houses),
  nextTo(house(E1,norwegian,E2,E3,E4), house(blue,F1,F2,F3,F4), Houses),
  nextTo(house(G1,american,G2,G3,G4), house(H1,H2,H3,water,H4), Houses),
  nextTo(house(I1,I2,I3,I4,kools), house(J1,J2,horse,J3,J4), Houses),
  nextTo(house(K1,K2,K3,K4,seven_stars), house(L1,L2,zebra,L3,L4), Houses),
  member(house(red,english,M1,M2,M3), Houses),
  member(house(N1,spanish,dog,N2,N3), Houses),
  member(house(green,O1,O2,coffee,O3), Houses),
  member(house(P1,ukranian,P2,tea,P3), Houses),
  member(house(yellow,Q1,Q2,Q3,kools), Houses),
  member(house(R1,R2,parrot,whiskey,R3), Houses),
  member(house(S1,S2,S3,orange_juice,lucky_strikes), Houses),
  member(house(T1,T2,snails,T3,winstons), Houses).

init([house(X11,norwegian,X13,X14,X15),
      house(X21,X22,X23,X24,X25),
      house(X31,X32,X33,milk,X35),
      house(X41,X42,X43,X44,X45),
      house(X51,X52,X53,X54,X55),
      house(X61,X62,X63,X64,X65)]).

member(X,[X|A]).
member(X,[A1,X|A2]).
member(X,[A1,A2,X|A3]).
member(X,[A1,A2,A3,X|A4]).
member(X,[A1,A2,A3,A4,X|A5]).
member(X,[A1,A2,A3,A4,A5,X]).

nextTo(A,B,[A,B|X]).
nextTo(A,B,[B,A|X]).
nextTo(A,B,[X1,A,B|X2]).
nextTo(A,B,[X1,B,A|X2]).
nextTo(A,B,[X1,X2,A,B|X3]).
nextTo(A,B,[X1,X2,B,A|X3]).
nextTo(A,B,[X1,X2,X3,A,B|X4]).
nextTo(A,B,[X1,X2,X3,B,A|X4]).
nextTo(A,B,[X1,X2,X3,X4,A,B]).
nextTo(A,B,[X1,X2,X3,X4,B,A]).

print([]).
print([X|Y]) :- write(X), nl, print(Y).
```

Apéndice B

INTERPRETACIÓN ABSTRACTA DEL BENCHMARK Q10

En este apéndice se describe el resultado obtenido en cada una de las operaciones de la fase de interpretación abstracta del benchmark q10. Estas operaciones se denominan pre-proceso, análisis global, post-proceso y compilación.

En referencia al pre-proceso y al análisis global, se muestran únicamente los aspectos concernientes a un único procedimiento del programa. En este caso, se pretende reflejar un ejemplo con la definición completa de las funciones que permiten realizar el análisis de determinismo y de tipos de datos de un programa, así como también un ejemplo con la evaluación de estas funciones.

En referencia a las operaciones de post-proceso y de compilación, se muestra la salida obtenida para todo el programa.

Procedimiento queens3/3

El procedimiento queens3/3 del benchmark q10 es el siguiente:

```
queens3([],Qs,Qs).
queens3(UnplacedQs,SafeQs,Qs) :-
        select(UnplacedQs,Remainder,Q),
        not_attack(SafeQs,Q),
        queens3(Remainder,[Q|SafeQs],Qs).
```

Los objetivos asociados a este procedimiento (objetivos padre) son dos: el tercer objetivo de la segunda cláusula de este procedimiento y el segundo objetivo de la cláusula queens2/2 (ver código completo en el apéndice A):

```
..., queens3(Remainder,[Q|SafeQs],Qs), ...
..., queens3(Ns,[],Qs), ...
```

Pre-proceso

Esta operación inicializa el estado de la ejecución de la interpretación abstracta. En concreto, define un punto de entrada al procedimiento con los dos objetivos padre asociados. Este punto de entrada no posee función de indexación y contiene dos cláusulas alternativas.

```
        ..., queens3_e1(Ns,[],Qs), ...          % referenciado como gl0


% ENTRY_POINT(queens3_e1/3).                    % referenciado como ep1
queens3_e1(A1,A2,A3) :-
        queens3_c1(A1,A2,A3);                  %
        queens3_c2(A1,A2,A3).                  %

% CLAUSE(queens_c1/3)                          % referenciada como cl1
queens3_c1([],Qs,Qs).

% CLAUSE(queens3_c2/3)                         % referenciada como cl2
queens3_c2(UnplacedQs,SafeQs,Qs) :-
        select(UnplacedQs,Remainder,Q),        % referenciado como gl1
        not_attack(SafeQs,Q),                  % referenciado como gl2
        queens3(Remainder,[Q|SafeQs],Qs).      % referenciado como gl3
```

También define las funciones FUNCT a partir de un análisis local de cada cláusula. La notación con que se describen estas funciones FUNCT utiliza los operadores y constantes definidas en el capítulo 5.

```
% Funciones FUNCT asociadas al punto de entrada ep1
% Los objetivos padre son: pgl(ep1) = {gl0, gl3}
% Las cláusulas del punto de entrada son: cl(ep1) = [cl1, cl2]
```

NSOL_EP(ep1) = or(*SUCC_CL(cl)*) $\forall$ *cl* $\in$ *cl(ep1)*

IN_EP_ARG(ep1,i) = union(*IN_GL_ARG(gl, i)*) $\forall$ *gl* $\in$ *pgl(ep1); i=1,2,3*

OUT_EP_ARG(ep1,i) = union(*OUT_CL_ARG(cl, i)*) $\forall$ *cl* $\in$ *cl(ep1); i=1,2,3*

% Funciones FUNCT asociadas a la cláusula cl1
% Las variables de esta cláusula son: var(cl1) = [Qs:1]

SUCC_CL(cl1) =
and(unif(*IN_GL_ARG(gl,1)*, (LST0,SINGLE)),
unif(*IN_GL_ARG(gl, 2)*, unif(*IN_GL_ARG(gl, 3)*)
) $\forall$ *gl* $\in$ *pgl(ep1)*

OUT_CL_ARG(cl1,1) = (LST0,SINGLE)
OUT_CL_ARG(cl1,2) = *OUT_CL_VAR(cl1, 1)*
OUT_CL_ARG(cl1,3) = *OUT_CL_VAR(cl1, 1)*

OUT_CL_VAR(cl1,1) = unif(*IN_EP_ARG(ep1,2), IN_EP_ARG(ep1,3)*)

INEP_CL_ARG(cl1,1) = intersection(*IN_EP_ARG(ep1,1)*, (LST0,SINGLE))
INEP_CL_ARG(cl1,2) = *IN_EP_ARG(ep1,2)*
INEP_CL_ARG(cl1,3) = *IN_EP_ARG(ep1,3)*

% Funciones FUNCT asociadas a la cláusula cl2
% Las variables de esta cláusula son: var(cl1) = [UnplacedQs:1, SafeQs:2, Qs:3, Remainder:4, Q:5]
% Los objetivos de esta cláusula son: gl(cl2) = [gl1, gl2, gl3]

SUCC_CL(cl2) = and(and(*NSOL_EP(ep(gl1)), NSOL_EP(ep(gl2))*), *NSOL_EP(ep(gl3))*)

OUT_CL_ARG(cl2,1) = *OUT_CL_VAR(cl2,1)*
OUT_CL_ARG(cl2,2) = *OUT_CL_VAR(cl2,2)*
OUT_CL_ARG(cl2,3) = *OUT_CL_VAR(cl2,3)*

OUT_CL_VAR(cl2,1) = val(*OUT_EP_ARG(ep(gl1),1), gl3*)
OUT_CL_VAR(cl2,2) = cdr(*OUT_EP_ARG(ep(gl3),2)*)
OUT_CL_VAR(cl2,3) = *OUT_EP_ARG(ep(gl3),3)*
OUT_CL_VAR(cl2,4) = *OUT_EP_ARG(ep(gl3),1)*
OUT_CL_VAR(cl2,5) = car(*OUT_EP_ARG(ep(gl3),2)*)

% Funciones FUNCT asociadas al objetivo gl1
IN_GL_ARG(gl1,1) = *IN_GL_VAR(gl1,1)*
IN_GL_ARG(gl1,2) = *IN_GL_VAR(gl1,4)*
IN_GL_ARG(gl1,3) = *IN_GL_VAR(gl1,5)*

IN_GL_VAR(gl1,1) = *IN_EP_ARG(ep1,1)*
IN_GL_VAR(gl1,4) = (VAR,SINGLE)
IN_GL_VAR(gl1,5) = (VAR,SINGLE)

% Funciones FUNCT asociadas al objetivo gl2
IN_GL_ARG(gl2,1) = *IN_GL_VAR(gl2,2)*
IN_GL_ARG(gl2,2) = *IN_GL_VAR(gl2,5)*

IN_GL_VAR(gl2,2) = val(*IN_EP_ARG(ep1,2), gl1*)
IN_GL_VAR(gl2,5) = *OUT_EP_ARG(ep(gl2),2)*

% Funciones FUNCT asociadas al objetivo gl3
IN_GL_ARG(gl3,1) = *IN_GL_VAR(gl3,4)*
IN_GL_ARG(gl3,2) = (LST+(or(*IN_GL_VAR(gl3,5)*, elem(*IN_GL_VAR(gl3,5)*))), SINGLE)
IN_GL_ARG(gl3,3) = *IN_GL_VAR(gl3,3)*

IN_GL_VAR(gl3,2) = val(*OUT_EP_ARG(ep(gl2),1),gl2*)
IN_GL_VAR(gl3,3) = val(*IN_EP_ARG(ep1,3),gl2*)
IN_GL_VAR(gl3,4) = val(*OUT_EP_ARG(ep(gl1),2),gl2*)
IN_GL_VAR(gl3,5) = val(*OUT_EP_ARG(ep(gl2),2),gl2*)

Análisis global

El análisis global evalúa todas las funciones FUNCT y obtiene los puntos de entrada de un procedimiento. En el procedimiento queens/3 se han obtenido dos puntos de entrada.

```
        ..., queens3_e2(Ns,[],Qs), ...

% ENTRY_POINT(queens3_e1/3).
queens3_e1(A1,A2,A3) :-
        queens3_c1(A1,A2,A3);
        queens3_c2(A1,A2,A3).

% ENTRY_POINT(queens3_e2/3).
queens3_e2(A1,A2,A3) :-
        queens3_c2(A1,A2,A3).

% CLAUSE(queens3_c1).
queens3_c1([],Qs,Qs).

% CLAUSE(queens3/c2).
queens3_c2(UnplacedQs,SafeQs,Qs) :-
        select_e1(UnplacedQs,Remainder,Q),
        not_attack_e1(SafeQs,Q),
        queens3_e1(Remainder,[Q|SafeQs],Qs).
```

El resultado de las funciones FUNCT es el siguiente

% Funciones FUNCT asociadas al punto de entrada ep1
NSOL_EP(ep1) = (OTHER,(),NO)

IN_EP_ARG(ep1,1) = (LSTc(CON,UNKNOWN),UNKNOWN)
IN_EP_ARG(ep1,2) = (LST+c(CON,UNKNOWN),SINGLE)
IN_EP_ARG(ep1,3) = (VAR,SINGLE)

OUT_EP_ARG(ep1,1) = (LSTc(CON,UNKNOWN),UNKNOWN)
OUT_EP_ARG(ep1,2) = (LST+c(CON,UNKNOWN),SINGLE))
OUT_EP_ARG(ep1,3) = (LST+c(CON,UNKNOWN),SINGLE)

% Funciones FUNCT asociadas al punto de entrada ep2
NSOL_EP(ep2) = (OTHER,(),NO)

IN_EP_ARG(ep2,1) = (LST+c(CON,UNKNOWN),UNKNOWN)
IN_EP_ARG(ep2,2) = (LST0,SINGLE)
IN_EP_ARG(ep2,3) = (VAR,SINGLE)

OUT_EP_ARG(ep2,1) = (LST+c(CON,UNKNOWN),UNKNOWN)
OUT_EP_ARG(ep2,2) = (LST0,SINGLE)
OUT_EP_ARG(ep2,3) = (LST+c(CON,UNKNOWN),SINGLE)

% Funciones FUNCT asociadas a la cláusula cl1
SUCC_CL(cl1) = (YES,(ARG,1,LST0),NO)

OUT_CL_ARG(cl1,1) = (LST0,UNKNOWN)
OUT_CL_ARG(cl1,2) = (LSTc(CON,UNKNOWN),SINGLE))
OUT_CL_ARG(cl1,3) = (LST+c(CON,UNKNOWN),SINGLE))

INEP_CL_ARG(cl1,1) = (LST0,UNKNOWN)

% Funciones FUNCT asociadas a la cláusula cl2
SUCC_CL(cl2) = (UNKNOWN,((ARG,1,LST+),OTHER),NO)

OUT_CL_ARG(cl2,1) = (LSTc(CON,UNKNOWN),UNKNOWN))
OUT_CL_ARG(cl2,2) = (LSTc(CON,UNKNOWN),SINGLE))
OUT_CL_ARG(cl2,3) = (LST+c(CON,UNKNOWN),SINGLE))

% Funciones FUNCT asociadas al objetivo gl1
IN_GL_ARG(gl1,1) = (LSTc(CON,UNKNOWN),UNKNOWN))
IN_GL_ARG(gl1,2) = (VAR,SINGLE)
IN_GL_ARG(gl1,3) = (VAR,SINGLE)

% Funciones FUNCT asociadas al objetivo gl2
IN_GL_ARG(gl2,1) = (LSTc(CON,UNKNOWN),SINGLE))
IN_GL_ARG(gl2,2) = (CON,UNKNOWN)

% Funciones FUNCT asociadas al objetivo gl3
IN_GL_ARG(gl3,1) = (LSTc(CON,UNKNOWN),UNKNOWN))
IN_GL_ARG(gl3,2) = (LST+c(CON,UNKNOWN),SINGLE))
IN_GL_ARG(gl3,3) = (VAR,SINGLE)

Post-proceso

La operación de post-proceso determina la función de indexación, la posible inserción de instrucciones de corte, el atributo de indeterminismo y el tipo dinámico de las variables. A continuación se muestra el resultado obtenido para todo el programa q10.

```
% PROGRAM_ENTRY_POINT(main/0).
?-      main_c1.

% CLAUSE(main_c1/0).
:- type(main_c1/0, Q, SINGLE).
main_c1 :-
        queens2_e1(10,Q), writeln(Q), fail.

% ENTRY_POINT(queens2_e1/2).
queens2_e1(A1,A2) :-
        queens2_c1(A1,A2).
```

```
% CLAUSE(queens2_c1).
:- type(queens2_c1, Ns, SINGLE).
queens2_c1(N,Qs) :-
        range_e2(1,N,Ns),
        queens3_e2(Ns,[],Qs).

% ENTRY_POINT(range_e1/3).
range_e1(A1,A2,A3) :-
        range_c1(A1,A2,A3);
        range_c2(A1,A2,A3).

% ENTRY_POINT(range_e2/3).
range_e2(A1,A2,A3) :-
        range_c2(A1,A2,A3).

% CLAUSE(range_c1/3).
:- type(range_c1/3, N, SINGLE).
range_c1(N,N,[N]) :-
        !.

% CLAUSE(range_c2/3).
:- type(range_c2/3, T, SINGLE).
:- type(range_c2/3, Np1, SINGLE).
range_c2(N,M,[N|T]) :-
        N < M,
        Np1 is N+1,
        range_e1(Np1,M,T).

% ENTRY_POINT(queens3_e1/3).
queens3_e1(A1,A2,A3) :-
        ( A1 == [] ->       queens3_c1(A1,A2,A3) ;
        ( list(A1) ->       queens3_c2(A1,A2,A3) ;
                            error)).

% ENTRY_POINT(queens3_e2/3).
queens3_e2(A1,A2,A3) :-
        queens3_c2(A1,A2,A3).

% CLAUSE(queens3_c1).
queens3_c1([],Qs,Qs) :-
        !.

% CLAUSE(queens3_c2).
:- attr(queens3_c2/3, G1, NONDET).
:- type(queens3_c2/3, Remainder, S/M(traversal, A1)).
:- type(queens3_c2/3, Q, S/M(traversal, A1)).
queens3_c2(UnplacedQs,SafeQs,Qs) :-
        select_e1(UnplacedQs,Remainder,Q),
        not_attack_e1(SafeQs,Q),
        queens3_e1(Remainder,[Q|SafeQs],Qs).

% ENTRY_POINT(select_e1/3).
select_e1(A1,A2,A3) :-
        ( A1 == [] ->       fail ;
        ( list(A1) ->       (select_c1(A1,A2,A3) ; select_c2(A1,A2,A3)) ;
                            error)).
```

```
% CLAUSE(select_c1/3).
select_c1([X|Xs],Xs,X).

% CLAUSE(select_c2/3).
:- type(select_c1/3, Zs, S/M(traversal,cdr(A1)).
select_c2([Y|Ys],[Y|Zs],X) :-
        select_e1(Ys,Zs,X).

% ENTRY_POINT(not_attack_e1/2).
not_attack_e1(A1,A2) :-
        not_attack_c1(A1,A2).

% CLAUSE(not_attack_c1/2).
not_attack_c1(Xs,X) :-
        not_attack_e1(Xs,X,1).

% ENTRY_POINT(not_attack_e1/3).
not_attack_e1(A1,A2,A3) :-
        ( A1 == [] ->       not_attack_c1(A1,A2,A3) ;
        ( list(A1) ->       not_attack_c2(A1,A2,A3) ;
                            error)).

% CLAUSE(not_attack_c1/3).
not_attack/c1([],_,_).

% CLAUSE(not_attack_c2/3).
:- type(not_attack_c2/3, YpN, S/M(car(A1)).
:- type(not_attack_c2/3, YmN, S/M(car(A1)).
:- type(not_attack_c2/3, N1, SINGLE).
not_attack_c2([Y|Ys],X,N) :-
        YpN is Y+N,
        X =\= YpN,
        YmN is Y-N,
        X =\= YmN,
        N1 is N+1,
        not_attack_e1(Ys,X,N1).
```

Compilación

El código correspondiente a la Máquina Abstracta de Multipath (instrucciones MAM) del benchmark q10 es el siguiente.

```
program('q10')

label('?-_0')
label('main_c1_0')
        allocate
        put_con_i(&10, A1)
        put_create-vrs-y_i(Y1, A2)
        call('queens2_e1_2', 1)
        put_y_i(Y1, A1)
        escape('writeln_1')
        deallocate
        fail
```

```
label('queens2_e1_2')
label('queens2_c1_2')
        allocate
        unf_i_create-var-y(A2, Y2)
        put_ix_i(A1, A2)
        put_con_i(&1, A1)
        put_create-vrs-y_i(Y1, A3)
        call('range_e2_3', 2)
        put_y_i(Y1, A1)
        put_con_i('[]', A2)
        put_y_i(Y2, A3)
        deallocate
        exec('queens3_e2_3')

label('range_e1_3')
label('range_e1_3_a1')
    try_me_else('range_e1_3_a2')
label('range_c1_3')
        unf_i_ix(A1, A2)
        unf_ix-tv_ref-lst(A3)
           create-arg_ix(A1)
           create-arg_con('[]')
        neck-cut
        proceed

label('range_e1_3_a2')
    trust_me
label('range_e2_3')
label('range_c2_3')
        allocate
        unf_i_create-var-y(A1, Y4)
        unf_i_create-var-y(A2, Y3)
        unf_ix-tv_ref-lst(A3)
           create-arg_y(Y4)
           create-arg_y(Y2)
        put_y_i(Y4, A1)
        put_y_i(Y3, A2)
        escape('<_2')
        put_create-vrs-y_i(Y1, A1)
        put_y_i(Y4, A2)
        put_con_i(&1, A3)
        escape('add_imp_3')
        put_y_i(Y1, A1)
        put_y_i(Y3, A2)
        put_y_i(Y2, A3)
        deallocate
        exec('range_e1_3')
```

```
label('queens3_e1_3')
   switch_on_term(A1, '_error_0', 'queens3_c1_3', 'queens3_c2_3', '_error_0')

label('queens3_c1_3')
       unf_i_ix(A2, A3)
       proceed

label('queens3_c2_3')
label('queens3_e2_3')
       allocate
       unf_i_create-var-y(A2, Y4)
       unf_i_create-var-y(A3, Y2)
       set-s/m(true, 1, 0, A1)
       put_create-vrx-y_i(Y1, A2)
       put_create-vrx-y_i(Y3, A3)
       call-nondet('select_e1_3', 4)
       put_y_i(Y4, A1)
       put_y_i(Y3, A2)
       call('not_attack_e1_2', 4)
       put_y_i(Y1, A1)
       put_ref-lst_ix(A2)
          create-arg_y(Y3)
          create-arg_y(Y4)
       put_y_i(Y2, A3)
       deallocate
       exec('queens3_e1_3')

label('select_e1_3')
   switch_on_term(A1, 'error_0', fail, 'select_e1_3_a1', 'error_0')

label('select_e1_3_a1')
   try_me_else('select_e1_3_a2')
       unf_ix-tl_ref-lst(A1)
          unf-arg_ix(A3)
          unf-arg_ix(A2)
       proceed

label('select_e1_3_2')
   trust_me
       unf_ix-tl_ref-lst(A1)
          unf-arg_create-var-x(X4)
          unf-arg_create-var-x(X5)
       set-s/m(true, 1, 0, X5)
       unf_ix-tv_ref-lst(A2)
          create-arg_ix(X4)
          create-arg_create-vrx-x(X6)
       put_ix_i(X5, A1)
       put_ix_i(X6, A2)
       exec('select_e1_3')
```

```
label('not_attack_e1_2')
        put_con_i(&1, A3)
        exec('not_attack_e1_3')

label('not_attack_e1_3')
    switch_on_term(A1, 'error_0', 'not_attack_c1_3', 'not_attack_c2_3', 'error_0')

label('not_attack_c1_3')
        proceed

label('not_attack_c2_3')
        unf_ix-tl_ref-lst(A1)
           unf-arg_create-var-x(X9)
           unf-arg_create-var-x(X4)
        unf_i_create-var-x(X6, A2)
        unf_i_create-var-x(X7, A3)
        set-s/m(false, 1, 0, X9)
        put_create-vrx-x_i(X10, A1)
        put_ix_i(X9, A2)
        put_ix_i(X7, A3)
        escape('add_imp_3')
        put_ix_i(X6, A1)
        put_ix_i(X10, A2)
        escape('=\=_2')
        put_create-vrx-x_i(X8, A1)
        put_ix_i(X9, A2)
        put_ix_i(X7, A3)
        escape('sub_imp_3')
        put_ix_i(X6, A1)
        put_ix_i(X8, A2)
        escape('=\=_2')
        put_create-vrs-x_i(X5, A1)
        put_ix_i(X7, A2)
        put_con_i(&1, A3)
        escape('add_imp_3')
        put_ix_i(X4, A1)
        put_ix_i(X6, A2)
        put_ix_i(X5, A3)
        exec('not_attack_e1_3')

label('error_0')
        quit(0)

endprogram
```

REFERENCIAS

[1] H. Aït Kaci. *Warren's Abstract Machine - A Tutorial Reconstruction.* The MIT Press, 1991.

[2] K. M. Ali. OR-Parallel Execution of Prolog on a Multi-Sequential Machine. *International Journal of Parallel Programming*, 1987.

[3] K. M. Ali. OR-Parallel Execution of Prolog on BC-Machine. In *Proc. of the 5th Int'l Conference and Symposium on Logic Programming*, pp. 1531-1545, 1988.

[4] K. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *Proc. of the 1990 North American Conference on Logic Programming*, The MIT Press, 1990.

[5] K. M. Ali and R. Karlsson. Scheduling Or-Parallelism in Muse. In *Proc. of the 8th Int'l Conference on Logic Programming ICLP'91*, pp. 807-821, The MIT Press, 1991.

[6] J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Progams. *Communications of the ACM*, pp. 613-641, 1978.

[7] A. Beaumont, S. Muthuraman, P. Szeredi and D. H. D. Warren. Flexible Scheduling of Or-Parallelism in Aurora: The Bristol Scheduler. In *Proc. of the Parallel Architecture and Languages Europe PARLE'91*, pp. 403-420. Lecture Notes in Computer Science, vol. 506, Springer-Verlag, 1991.

[8] J. Bendl, P. Köves and P. Szeredi. The MPROLOG System. In *Proc. of the Logic Programming Workshop*, pp. 201-209, Debrecen, Hungary, 1980.

[9] P. Borgwardt. Parallel Prolog using Stack Segments on Shared Memory Multiprocessors. *In Proc. of the 1984 Int'l Symposium on Logic Programming ILPS'84*, pp. 2-11, The MIT Press, 1984.

[10] D. Boucher and M. Feeley. Abstract Compilation: A New Implementation Paradigm for

Static Analysis. In *Proc. of the 6th Int'l Conference on Compiler Construction CC'96*, pp. 192-207, Lecture Notes in Computer Science 1060, Springer Verlag, 1996.

[11] R. S. Boyer and J. S. Moore. The Sharing of Structure in Theorem Proving Programs. In *Machine Intelligence 7*, pp. 101-116, Edinburgh University Press, 1972.

[12] P. Brandt. Wavefront Scheduling. Internal Report, gigalips project, SICS, 1988.

[13] M. Bruynooghe. The Memory Management of Prolog Implementations. In *Logic Programming*, K. Clark and S. Tärnlund (ed.), pp. 83-98, Academic Press, 1982.

[14] R. Butler, T. Disz, E. Lusk, R. Olson, R. A. Overbeek and R. Stevens. Scheduling OR-Parallelism: An Argonne Perspective. In *Proc. of the 5th Int'l Conference and Symposium on Logic Programming*, pp. 1590-1605, The MIT Press, 1988.

[15] A. Calderwood and P. Szeredi. Scheduling Or-Parallelism in Aurora - The Manchester Scheduler. In *Proc. of the 6th Int'l Conference on Logic Programming ICLP'89*, The MIT Press, 1989.

[16] M. Carlsson, J. Widèn, J. Andersson, S. Andersson, K. Boortz, H. Nilsson and T. Sjöland. *SICStus Prolog Users's Manual*, SICS, Box 1263, 164 28 Kista, Sweden, 1991.

[17] J.-H. Chang and A. M. Despain. Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis. In *Proc. of the Symposium on Logic Programming SLP'85*, The MIT Press, 1985.

[18] J. Chassin de Kergommeaux and P. Codognet. Parallel Logic Programming Systems. ACM Computing Surveys, vol. 26, no. 3, pp. 295-336, 1994.

[19] C. Chen, A. Singhal and Y. N. Patt. *PUP: An Architecture to Exploit Parallel Unification in Prolog*. Report UCB/CSD 88/414, UC Berkeley, 1988.

[20] A. Ciepielewski, S. Haridi and B. Hausman. OR-Parallel Prolog on Shared Memory Multiprocessors. *Journal of Logic Programming*, vol. 7, pp. 125-147, 1989.

[21] K. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 1, pp. 1-49, 1986.

[22] P. Codognet and D. Díaz. Clp(B): Combining Simplicity and Efficiency in Boolean Constraint Solving. In *Proc. of the 6th Int'l Symp. on Programming Language Implementation and Logic Programming PLILP'94*, Springer-Verlag, 1994.

[23] J. Cohen. First Specialize, Then Generalize. *Communications of the ACM*, vol. 35, no. 3, pp. 34-39, 1992.

[24] A. Colmerauer. The Birth of Prolog. In *the Second ACM-SIGPLAN History of Programming Languages Conference*, ACM SIGPLAN Not., pp. 37-52, 1993.

[25] J. S. Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs.* Ph. D. Thesis, University of California, Irvine. Also available as Tech. Report 204, Dpt. of Computer and Information Science, U.C. Irvine, 1993.

[26] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of FixPoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pp. 238-252, 1977.

[27] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, vol. 13, num. 2-3, 1992.

[28] P. Cox and T. Pietrzykowsky. Deduction Plans: A Basis for Intelligent Bactracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3, pp. 52-65, 1981.

[29] J. Crammond. A Comparative Study of Unification Algorithms For Or-Parallel Execution of Logic Languages. In *Proc. of the Int'l Conference on Parallel Processing ICPP'85*, 1985.

[30] S. Debray. *Global Optimization of Logic Programs.* Ph. D. dissertation. Computer Science Dpt., SUNY Stony Brook, 1986.

[31] R. Dechter and J. Pearl. Generalize Best-First Search Strategies and the Optimality of A*. *Journal of the ACM*, vol. 32, no. 3, pp. 505-536, 1985.

[32] D. DeGroot. Restricted AND-Parallelism. In *Proc. of the Int'l Conference on Fifth Generation Computer Systems*, pp. 361-368, 1994.

[33] S. A. Delgado-Rannauro. OR-Parallel Logic Computational Models. In P. Kacsuk and M. J. Wise (eds.) *Implementations of Distributed Prolog*, pp. 3-26, John Wiley and Sons, 1992.

[34] D. Díaz and P. Codognet. A Minimal Extension of the WAM for the clp(FD). In *Proc. of the 10th Int'l Conf. on Logic Programming, ICLP'93*, pp. 774-790, 1993.

[35] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier. The

Constraint Logic Programming Language CHIP. In *Proc. of the Int'l Conference on Fifth Generation Computer Systems*, pp. 693-702, 1988.

[36] T. P. Dobry. *A High Performance Architecture for Prolog*. Ph. D. dissertation. Report UCB/CSD 87/352, Dpt. of Computer Science, UC Berkeley, 1987. Also published by Kluwer Academic Publishers, 1990.

[37] E. Elias. *DAC-Prolog: Un compilador de Prolog a WAM*. Tesina Dpt. Arquitectura de Computadors. Universitat Politècnica de Catalunya, 1996.

[38] G. Escalada-Imaz. A Parallel Interpretation Model of Logic Programs based on Multisystems of Equations. In *Parallel Computing and Transputers Applications*, pp. 745-754, IOS Press, 1992.

[39] Z. Farkas, P. Köves and P. Szeredi. MProlog: An Implementation Overview. In *Proc. of the ICLP'94 Workshop on Practical Implementations and Systems Experience*, Budapest, Hungary, 1993.

[40] K. Furukawa. Logic Programming as the Integrator of the Fifth Generation Computer Systems Project. *Communications of the ACM*, vol. 35, no. 3, pp. 82-92, 1992.

[41] A. González, J. Tubella and C. Aliagas. An Evaluation Tool for the EDS Parallel Logic Programming System. *Parallel Computing: From Theory to Sound Practice*. IOS Press, pp. 566-569, 1992.

[42] A. González and J. Tubella. The Multipath Parallel Execution Model for Prolog. In *Proc. of the 1st Int. Conf. on Parallel Symbolic Computation PASCO94*, pp. 164-173, World Scientific Pub., September 1994.

[43] G. Gupta and B. Jayaraman. On Criteria for OR-Parallel Execution Models of Logic Programs. In *Proc. of the 1990 North American Conf. on Logic Programming*, pp. 737-756, MIT Press, 1990.

[44] S. Habata, R. Nakazaki, A. Atarashi and M. Umemara. Co-operative High Performance Sequential Inference Machine: CHI. In *Proc. of the Int'l Conference on Computer Design ICCD'87*, pp. 601-604, IEEE Computer Society Press, 1987.

[45] R. Halstead. MultiLISP: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Language and Systems*. Vol. 7, No. 4, pp. 501-538, 1985.

[46] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Prac-

tice. *Journal of Logic Programming*, vol. 19-20 pp. 583-628, 1994.

[47] G. Haworth, S. Leunig, C. Hammer and M. Reeve. The European Declarative System, Database, and Languages. *IEEE Micro*, vol. 10, no. 6, December 1990.

[48] F. Henderson, Z. Somogyi and T. Conway. Determinism Analysis in the Mercury Compiler. In *Proc. of the Australian Computer Science Conference*, pp. 337-346, 1996.

[49] M. V. Hermenegildo. An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs. In *Proc. of the Third Int'l Conference on Logic Programming* ICLP'86, pp. 25-40, Lecture Notes in Computer Science, Springer-Verlag, 1986.

[50] M. V. Hermenegildo and K. J. Greene. The &-Prolog System: Exploiting Independent Ans-Parallelism. New Generation Computing, vol. 9, pp. 233-256, 1991.

[51] M. V. Hermenegildo, R. Warren and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, vol. 13, no. 4, pp. 349-367, 1992.

[52] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[53] B. K. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, J. M. Pendleton, T. P. Dobry, W. R. Bush and A. M. Despain. Fast Prolog with an Extended General Purpose Architecture. In *Proc. of the 17th. Int'l Symposium on Computer Architecture ISCA*, pp. 282-291, IEEE Computer Society Press, 1990.

[54] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, vol. 21, no.3, 1989.

[55] P. Hudak and P. Wadler. Report on the Functional Programming Language Haskell. *SIGPLAN Not.* 27(5), 1992.

[56] R. A. Ianucci, G. R. Gao, R. H. Halstead Jr. amd B. Smith. *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer Academic Publishers, 1994.

[57] ISO/IEC. *ISO Standard for Prolog ISO/IEC 13211-1:1995*. A 1993 draft is available by anonymous FTP from ai.uga.edu:/pub/prolog.standard/.

[58] J. Jaffar, S. Michaylov, P. Stuckey and R. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages* 14(3):339-395, 1992.

[59] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. In *Journal of Logic Programming*, vol. 19-20 pp. 503-581. Elsevier Science, 1994.

[60] R. Jeffries, A. A. Turner, P. G. Polson and M. E. Atwood. The Processes Involved in Designing Software. In *Cognitive Skills and Their Acquisition*, pp. 255-283. Lawrence Erlbaum Associates, 1981.

[61] P. Kacsuk and A. Bale. DAP Prolog: A Set-Oriented Approach to Prolog. *The Computer Journal*, vol. 30. pp. 393-403, 1987.

[62] Y. Kanada, K. Kojima and M. Sugaya. Vectorization Techniques for Prolog. In *Proc. of the Int. Conf. on Supercomputing*, pp. 539-549, 1988.

[63] M. Kantrowitz. *Prolog Resource Guide*. Available in news://comp.lang.prolog; ftp://ftp.cs.cmu.edu:/users/ai/pubs/faqs/prolog/prg_?.faq; http://www.cs.cmu.edu/Web/Groups/AI/html/faqs/lang/prolog/prg/top.html; mailto:mkant+prg@cs.cmu.edu.

[64] S. Kliger and E. Shapiro. From Decision Trees to Decision Graphs. In *Proc. of the North American Conference on Logic Programming NACLP'90*, pp. 97-116, The MIT Press, 1990.

[65] P. M. Kogge. *The Architecture of Symbolic Computers*. McGraw-Hill, 1991.

[66] R. E. Korf. Depth-First Iterative-Deepening: An Optimal Admisable Tree Search. *Artificial Intelligence*, vol. 27, no. 1, pp. 97-109, 1985.

[67] R. A. Kowalski. Predicate Logic as a Programming Language. In *Information Procesing 74*, pp. 569-574, North-Holland, 1994.

[68] V. Kumar and L. Kanal. A General Branch and Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures. *Artificial Intelligence*, vol. 21, no. 1, pp. 179-198, 1983.

[69] V. Kumar and Y.-J. Lin. A Data-Dependency-Based Intelligent Backtracking Scheme for Prolog. *Journal of Logic Programming*, vol. 5, pp. 165-181, 1988.

[70] K. Kurosawa, S. Yamaguchi, S. Abe and T. Bandoh. Instruction Architecture for a High Performance Integrated Prolog Processor IPP. In *Proc. of the 5th. Int'l Congference and Symposium on Logic Programming*, pp. 1506-1530, The MIT Press, 1988.

[71] G.-J. Li and B. W. Wah. How Good are Parallel and Orderd Depth-First Searches? In *Proc. of the Int'l Conference on Parallel Processing ICPP'86*, pp. 992-999, 1986.

[72] Y. Lin and V. Kumar. An Execution Model for Exploiting AND-Parallelism in Logic Progams. *New Generation Computing*, pp. 393-425, May 1988.

[73] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd. edition 1987.

[74] J. W. Lloyd. Combining Functional and Logic Programming Languages. In *Proc. of the 1994 Int'l Logic Programming Symposium ILPS'94*. Invited paper, 1994.

[75] J. W. Lloyd. Practical Advantages of Declarative Programming. In. *Proc. of the Joint Int'l Conf. on Declarative Programming, GULP-PRODE'94*, 1994.

[76] E. Lusk, D. H. D. Warren, S. Haridi et al. The Aurora OR-Parallel Prolog System. *New Generation Computing*, vol, 7 num. 2-3, pp. 243-271, 1990.

[77] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, vol. 4 no. 2, pp. 258-282, 1982.

[78] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart and M. I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, 1965.

[79] M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D. Henry de Villeneuve, A. Herold, P. Kay, B. Pérez, E. van Rossum and J. Schimpf. SEPIA - An Extendible Prolog System. In *Proc. of the 11th. World Computer Congress IFIP'89*. pp. 1127-1132, 1989.

[80] M. Meier. Compilation of Compound Terms in Prolog. In *Proc. of the North American Conference on Logic Programs NACLP'90*, pp. 63-79, The MIT Press, 1990.

[81] M. Meier. Better Later than Never. In *Implementations of Logic Programming Systems*, E. Tick and G. Succi (ed.), pp. 151-165, Kluwer Academic Publishers, 1994.

[82] D. Miller and G. Nadathur. Higher-Order Logic Programming. In *Proc. of the 3rd Int. Conf. of Logic Programming, Lecture Notes in Computer Science 225*, pp. 448-462. Springer-Verlag, 1986.

[83] J. Minker. Perspectives in Deductive Databases. *Journal of Logic Programming*, vol. 5, pp. 292-308, 1988.

[84] J. J. Moreno-Navarro and M. Rodríguez-Artalejo. BABEL: A Functional and Logic Programming Language Based on Constructor Discipline and Narrowing. In *Proc. of the Conference on Algebraic and Logic Programming ALP'89*. Lecture Notes Computer Science 343, pp. 223-232, 1989.

[85] J. J. Moreno-Navarro. Expressivity of Functional-logic Languages and their Implementation. *Tutorials of the Joint Int'l Conf. on Declarative Programming GULP-PRODE'94*, 1994.

[86] L. Naish. *Negation and Control in Prolog*. Ph. D. Dissertation, University of Melbourne. Also published as *Lectures Notes in Computer Science 238*, Springer-Verlag, 1986.

[87] L. Naish. Parallelizing NU-Prolog. In *Proc. of the 5th. Int'l Conference and Symposium on Logic Programming ICSLP'88*, pp. 1546-1564, 1988.

[88] L. Naish, P. Dart and J. Zobel. The NU-Prolog Debugging Environment. In *Proc. of the 6th. Int'l Conference on Logic Programming ICLP'89*, pp. 521-536. The MIT Press, 1989.

[89] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *Proc. of the Symposium on Logic Programming*, pp. 107-113, IEEE Computer Society Press, 1987.

[90] J. Noyé. An Overview of the Knowledge Crunching Machine. In *Emerging Trends in Database and Knowledge-base Machines*, IEEE Computer Society Press, 1993.

[91] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.

[92] L. M. Pereira and A. Porto. *An Interpreter of Logic Programs Using Selective Backtracking*. Report 3/80, Dept. de Informatica, Universidade Nova de Lisboa, 1980.

[93] A. Pettorosi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, vol 19-20, pp. 261-320. Elsevier Science, 1994.

[94] K. Ramamohanarao, J. Sheperd, I. Balbin, G. Port, L. Naish, J. Thom, J. Zobel and P. Dart. The NU-Prolog Deductive Database System. *IEEE Transactions on Data Engineering*, vol. 10, no. 4, pp. 10-19, 1987.

[95] J. A. Robinson. A Machine-oriented Logic based on the Resolution Principle. *Journal of the ACM*, vol. 12, pp. 23-41, 1965.

[96] J. A. Robinson. Logic and Logic Programming. *Communications of the ACM*, vol. 35, no. 3, pp. 40-65, 1992.

[97] V. Santos Costa, D. H. D. Warren and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits Both AND- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP'91*, pp. 83-93, SIGPLAN Notices vol. 26(7), 1991.

[98] V. Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I*. PhD Thesis. Department of Computer Science. University of Bristol, August

1993.

[99] E. Shapiro. *A Subset of Concurrent Prolog and its Interpreter.* Tecnical Report, The Weizmann Institute of Science, Rehovot, Israel, 1983.

[100] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, vol, 21, no. 3, 1989.

[101] F. N. Sibai, K. L. Watson and M. Lu. A Parallel Unification Machine. *IEEE Micro*, August 1990.

[102] Z. Somogyi, F. Henderson and T. Conway. The execution algorithm of Mercury: an eficient purely declarative logic programming language. To appear in the *Journal of Logic Programming.*

[103] G. Spring and D. Friedman. *Scheme and the Art of Programming.* McGraw-Hill/MIT Press, 1990.

[104] V. P. Srini, J. Tam, T. Nguyen, C. Chen, A. Wei, J. Testa, Y. Patt and A. M. Despain. VLSI Implementation of a Prolog Processor. In *Proc. of the Stanford VLSI Conference,* 1987.

[105] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, 1994.

[106] G. Steele. *Common LISP - The Language.* Digital Press, 1984.

[107] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[108] D. A. Smith. Multilog: Data OR-Parallel Logic Programming. In *Proc. of the 10th Int. Conf. on Logic Programming*, MIT Press, 1993.

[109] D. A. Smith and T. Hickey. Multi-SLD Resolution. In *Proc. of Logic Programming and Automated Reasoning LPAR'94*, Springer-Verlag, 1994.

[110] D. A. Smith. Modeling Backtracking, Disjunctive Constraints, and Control/Data Or-Parallelism. In *Proc. of the Post-ICLP'94 Workshop on Parallel and Data Parallel Execution of Logic Programs*, 1994.

[111] D. A. Smith. Why Multi-SLD Beats SLD (Even on a Uniprocessor). In *Proc. of the 6th Int'l Symp. on Programming Language Implementation and Logic Programming,*

PLILP'94. LNCS 844, Springer-Verlag, 1994.

[112] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In P*roc. of the North American Conference on Logic Programming*, The MIT Press, 1989.

[113] K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima and A. Mitsuishi. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In *Future Generation Computer Systems*, 1984.

[114] E. Tick. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, 1988.

[115] E. Tick. *Parallel Logic Programming*. The MIT Press, 1991.

[116] H. Touati and A. M. Despain. An Empirical Study of the Warren Abstract Machine. In *Proc. of the Symposium on Logic Programming SLP'87*, pp. 114-124, IEEE Computer Society Press, 1987.

[117] J. Tubella, A. González and C. Aliagas. Representación de la carga de programas lógicos en la evaluación de un sistema multiprocesador. *Jornadas sobre Programación Declarativa PRODE'91*, pp. 497-518, 1991.

[118] J. Tubella, A. González and C. Aliagas. Design and Evaluation of a Two-Level Hierarchical Multiprocessor for Logic Programming. In *Proc. of the 10th IASTED Int'l Conf. on Applied Informatics*, Acta Press, pp. 45-48, 1992.

[119] J. Tubella and A. González. Measuring Scheduling Policies in Pure Or-Parallel Programs. In *Proc. of the 2nd Conf. on Declarative Programming PRODE'93*, pp. 57-71, 1993.

[120] J. Tubella and A. González. MEM: A New Execution Model for Prolog. *Microprocessing and Microprogramming*, vol. 39, pp. 83-86, North-Holland, 1993.

[121] J. Tubella and A. González. Combining Depth-First and Breadth-First Search in Prolog Execution. In *Proc. of the 1994 Joint Conf. on Declarative Programming GULP-PRODE94*. vol. 2, pp. 452-453, 1994.

[122] J. Tubella and A. González. A Partial Breadth-First Execution Model for Prolog. In *Proc. of the 6th Int. Conf. on Tools with Artificial Intelligence TAI94*, pp. 129-137, IEEE Computer Society Press, November 1994.

[123] J. Tubella and A. González. Exploiting Path Parallelism in Logic Programming. In *Proc. of the 3rd Euromicro Workshop on Parallel and Distributed Processing PDP95*, pp. 164-

173, IEEE Computer Society Press, January 1995.

[124] J. Tubella and A. González. *The MULTIPATH Architecture for Prolog Programs.* Research Report UPC-DAC-1995-55. Dpt. of Computer Architecture (UPC), 1995.

[125] D. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In *Conf. on Functional Programming Languages and Computer Architecture, LNCS 201*, pp. 1-16, 1985.

[126] K. Ueda. Making Exhaustive Search Programs Deterministic. In *Proc. of the Third Int'l Conference on Logic Programming ICLP'86*, pp. 270-282, 1986.

[127] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[128] P. Van Roy and A. M. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer*, vol. 25, no. 1, pp. 54-68, 1992.

[129] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, vol. 19-20, pp. 385-441, Elsevier Science Inc., 1994.

[130] D. H. D. Warren. Prolog on the DECsystem-10. In *Expert Systems in the Micro-Electronic Age*, Edinburgh University Press, 1979.

[131] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical note 309, SRI International Artificial Intelligence Center, 1983.

[132] D. H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog. Abstract Design and Implementation Issues. In *Proc. of the Symposium on Logic Programming SLP'87*, pp. 46-53, IEEE Computer Press, 1987.

[133] D. S. Warren. Efficient Prolog Memory Management for Flexible Control Strategies. In *Proc. of the 1984 Int'l Symposium on Logic Programming ILPS'84*, pp.198-202, MIT Press, 1984.

[134] D. S. Warren. Memoing for Logic Programs. *Communications of the ACM*, vol. 35, no. 3, pp. 93-11, 1992.

[135] H. Westphal, P. Robert, J. Chassin and J. Syre. The PEPSys Model: Combining Backtracking, AND- and OR-parallelism. In *Proc. of the 1987 Int. Symposium on Logic Programming*, pp. 436-448, The MIT Press, 1987.

[136] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, Inc. 1976.

[137] R. Yang, T. Beaumont, I. Dutra, V. Santos Costa and D. H. D. Warren. Performance of the Compiler-based Andorra-I System. In *Proc. of the 10th Int'l Conference on Logic Programming*, pp. 150-166, The MIT Press, 1993.

[138] T. Yokota and K. Seo. Pegasus - An ASIC Implementation of High-Performance Prolog Processor. In *Proc. of the EURO ASIC'90*, pp. 156-159, IEEE Computer Society Press, 1990.