UNIVERSITAT POLITÈCNICA DE CATALUNYA

Departament d'Arquitectura de Computadors

RECURSOS ANCHOS: UNA TÉCNICA DE BAJO COSTE PARA EXPLOTAR PARALELISMO AGRESIVO EN CÓDIGOS NUMÉRICOS

Autor: David López Alvarez
Directores: Mateo Valero Cortés
Josep Llosa i Espuny

Capítulo 1

Conceptos básicos de paralelismo a nivel de instrucciones.

1.1 Introducción

Desde sus comienzos, uno de los objetivos fundamentales de la investigación en computadores ha sido la disminución del tiempo de ejecución de los programas, dedicándosele muchos esfuerzos a todos los niveles (tecnología, arquitectura, compilación, aplicaciones,...). Una parte importante de ese esfuerzo se ha dirigido a la búsqueda del paralelismo, es decir, al intento de ejecutar más de una instrucción simultáneamente. Hay diversos grados de paralelismo, dependiendo de a qué nivel se intente aplicar. El paralelismo a nivel de instrucciones (*Instruction-Level Parallelism, ILP*) es una familia de técnicas de diseño para procesadores y compiladores orientadas a explotar el paralelismo existente entre instrucciones de lenguaje máquina. Las operaciones implicadas son las normales de un procesador tipo RISC, y de un programa escrito pensando en un típico procesador secuencial (una de las características principales de estas técnicas es que son transparentes al programador).

Aunque se han explotado pequeñas cantidades de ILP en procesadores desde la década de los 60, fue en la década de los 80 cuando se empezó a usar ampliamente en procesadores comerciales y, desde mediados de los 90, la mayoría de los procesadores comerciales incorporan, en mayor o menor medida, técnicas avanzadas de ILP.

En este capítulo veremos una perspectiva histórica de los procesadores y las técnicas hardware y software empleadas para explotar ILP, y describimos los conceptos básicos utilizados a lo largo de este trabajo.

1.2 Perspectiva histórica

Si la industria automovilística hubiera hecho en los últimos 30 años lo que la informática, hoy podríamos comprar un Rolls Royce por 2$, que consumiera 1 litro de gasolina cada 10 millones de km.

Malcom Forbes. Forbes Magazine 1985.

Hasta finales de la década de los 50, la tecnología electrónica estaba dominada por las válvulas de vacío. La invención del transistor en 1947 y del circuito integrado a principios de la década de los 60 marcaron el comienzo de una revolución, dado que la cantidad de transistores que caben en un *chip* ha tenido un crecimiento exponencial. Este efecto ya fue predicho en 1965 por Gordon E. Moore [Moo65] en la ley de Moore, en la que afirmó que "el número de transistores de un circuito integrado se dobla, aproximadamente, cada 18 meses", ley que sigue siendo válida hoy día [Moo95, Sch97]. Así, los transistores integrados en un único *chip* pasaron de 2 a principios de los años 60 a los casi 20 millones de la actualidad.

Este crecimiento de las posibilidades de un chip permitió tener, ya en los 60, procesadores donde cabían más puertas lógicas que las estrictamente necesarias para construir un procesador de propósito general. Algunas máquinas comerciales de la época aprovecharon esta disponibilidad de espacio para implementar una forma limitada de ILP.

En 1963, Control Data Corporation empezó a trabajar en el CDC6600 [Tho64], que tenía diez unidades funcionales. Cada una de ellas podía comenzar a ejecutar una instrucción aunque otra instrucción anterior no hubiera acabado en otra unidad funcional, siempre y cuando ambas instrucciones fueran independientes. En esta máquina el hardware decidía qué operación se lanzaba en un ciclo determinado.

Tres años después, IBM lanzaba el IBM 360/91 [IBM67]. Esta máquina, basada parcialmente en el procesador experimental de IBM Stretch [Blo59, Buc62], ofrecía más ILP que el CDC6600, aunque su rendimiento fuera menor pues tenía menos unidades funcionales. Las ideas usadas para mantener las unidades funcionales ocupadas (el algoritmo de Tomasulo [Tom67]) siguen siendo básicas en los actuales sistemas superescalares.

La aparición de la filosofía RISC (*Reduced Instruction Set Code*) fue fundamental para la explotación del ILP. Los procesadores RISC disponen de pocas instrucciones, muy sencillas y muy generales, lo que permite una rápida interpretación y ejecución. Además, debido a que cada instrucción ocupa menos recursos, la planificación hardware es más sencilla.

La filosofía RISC ya estaba presente en el IBM Stretch y en los primeros computadores de Seymour Cray (que trabajó en el equipo diseñador del CDC6600 y CDC7600 antes de fundar su propia compañia, Cray Research e introducir el CRAY-I en 1976). A finales de los 70 estas ideas eran introducidas por John Cocke en el diseño del IBM 801 [Rad82], que no llegó a comercializarse. Las ideas se acabaron de definir en proyectos de dos universidades, el *Berkeley RISC project* y el *Stanford MIPS project* [PaDi80, PaSe81, HJB+82].

Los procesadores han ido incorporando conceptos de ILP en su arquitectura, con la aparición de los procesadores segmentados en la década de los 50, los vectoriales en los 70 y los superescalares y los *Very Long Instruction Word* (VLIW) en los 80. La mayoría de los microprocesadores diseñados durante los 90 implementan, en mayor o menor medida, conceptos de ILP. Algunos ejemplos son: los IBM RS-6000 [IBM90] y POWER2 [WhDh94], los IBM/Motorola PPC601 [Moo93], PPC603 [BAH+94], PPC604 [SDC94] y PPC620

[DNJ95], los MIPS R8000 [Hsu94] y R10000 [Hei94], los DEC Alpha 21064 [DEC92], Alpha 21164 [ERPR95] y Alpha 21264 [MR96] y los HP PA-7100 [AAD+93], PA-7200 [KCZ+94] y PA-8000 [Kum97]. Esta tendencia parece apuntar a que los microprocesadores del futuro seguirán haciendo uso de técnicas para explotar el ILP.

1.3 Evolución de la arquitectura desde la perspectiva del ILP

El ordenador es la evolución lógica
del hombre: inteligencia sin moral.

John J. Osborne

Una máquina de propósito general es capaz de ejecutar una serie de instrucciones (programa) almacenado en memoria. Las primeras aproximaciones a máquinas de propósito general ejecutaban las instrucciones de una en una secuencialmente, de manera que para que una instrucción empezase a ejecutarse era necesario que la instrucción anterior hubiera acabado su ejecución (figura 1.1). Este tipo de arquitectura se denomina secuencial. A continuación veremos varias arquitecturas que intentan solapar la ejecución de diversas instrucciones.

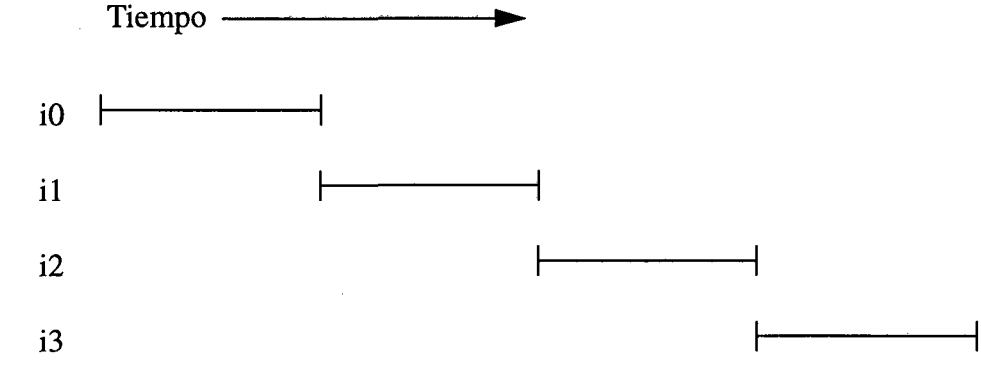

Figura 1.1 : Ejecución secuencial

Arquitecturas segmentadas y supersegmentadas

. Segmentación (*pipelining*) [Kog81] es una técnica que explota de manera efectiva el ILP, siendo utilizada por primera vez en los años 50 en el IBM Stretch [Blo59, Buc62]. En un procesador segmentado, los diversos pasos de una instrucción se ejecutan en unidades diferentes, llamadas etapas (*pipeline stages*), que están separadas por registros, en los que se guarda la información que debe pasar de una etapa a la siguiente. De esta manera, dos instrucciones pueden estar ejecutándose simultáneamente, cada una de ellas en una etapa diferente. Por ejemplo, la figura 1.2 muestra un procesador en que se ha dividido cada instrucción en tres etapas (*fetch* o búsqueda de la instrucción, decodificación y ejecución). El tiempo de ciclo del procesador se adapta a la etapa más larga en tiempo, de manera que a cada ciclo de reloj se activa una nueva instrucción y las que ya estaban activas cambian a la siguiente etapa. Así pues, en el ciclo 0 se lanza la instrucción i0, que entra en la etapa de *fetch*; en el ciclo 1, la instrucción i1 entra en la etapa de *fetch*, mientras la i0 entra en la etapa de decodificación; en el ciclo 2, la instrucción i2 entra en la etapa de *fetch*, mientras i1 cambia a decodificación e i0 cambia a ejecución. En este instante, el *pipeline* está lleno (todas las etapas están en funcionamiento) y se están ejecutando 3 instrucciones en paralelo.

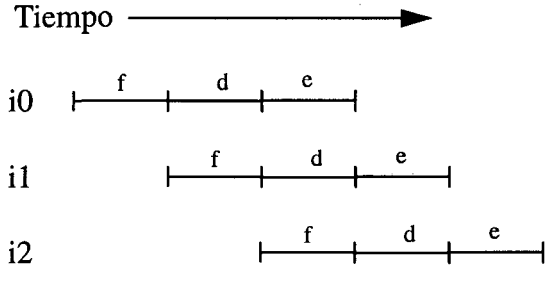

Figura 1.2 : Ejecución segmentada

Aunque el tiempo de ejecución de una instrucción sea mayor que en un procesador secuencial (debido a que todas las etapas duran el mismo tiempo, es decir, el de la etapa más larga), el tiempo de ejecución de un programa se reduce substancialmente.

No siempre se pueden lanzar instrucciones a la velocidad necesaria para mantener el *pipeline* lleno, debido a dependencias entre instrucciones. Dos instrucciones son dependientes cuando el resultado del programa depende del orden de ejecución de las mismas, por lo que deben ejecutarse en el mismo orden en que fueron escritas originalmente. Cabe destacar que las máquinas RISC fueron diseñadas pensando en la segmentación, y que la totalidad de los procesadores actuales la implementan.

Llevando la segmentación al límite, encontramos los procesadores supersegmentados (*superpipelined*) [Jou89], donde las etapas se dividen en subetapas y se decrementa el tiempo de ciclo, al tiempo que se aumenta el grado de paralelismo. Un ejemplo de máquina supersegmentada es el MIPS R4000 [MWV92].

Arquitecturas vectoriales

Una arquitectura vectorial dispone de instrucciones en alto nivel que trabajan sobre vectores. Una típica operación vectorial sería sumar dos vectores de 64 elementos en coma flotante, dejando el resultado en un tercer vector; todo ello se realizaría en una única instrucción. Las ventajas de un procesador vectorial se centran en su rapidez, debida a:

- el decremento de instrucciones a ejecutar: una única instrucción substituye a varias, de manera que nos ahorra el tiempo de *fetch* y decodificación de las instrucciones substituidas.

- al ejecutar varias iteraciones escalares mediante una iteración vectorial reducimos el número de instrucciones ejecutadas debidas al control del bucle (incremento de índices, saltos, etc...).

- si una operación es vectorial, se puede lanzar sin tener que controlar los riesgos dinámicamente, ya que esta información la provee el compilador o directamente el programador.

- al acceder a memoria con patrones conocidos (se sabe a priori cual será el siguiente elemento de memoria al que se accederá), el tiempo de acceso se reduce.

Los costes de un procesador vectorial van asociados a sus ventajas: se requiere un hardware costoso para implementar las unidades funcionales vectoriales, así como el banco de registros y, sobre todo, el subsistema de memoria, que debe ser capaz de desarrollar suficiente velocidad para impedir que las unidades funcionales queden paradas a media operación. A nivel del compilador, éste debe detectar los riesgos de dependencias.

Los primeros procesadores vectoriales fueron el CDC STAR-100 [HiTa72] y el TI ASC [Wat72], aunque el primer vectorial de éxito comercial fue el CRAY-1 [Rus78], de Cray Research. Casas como CRAY, NEC o Fujitsu han anunciado nuevos procesadores vectoriales a corto plazo.

Arquitecturas superescalares

El término procesador superescalar aparece por primera vez en el artículo de Agerwala y Cocke [AgCo87], siendo el primer procesador que lo implementa el Astronautics ZS-1 [SDV+87]. Un procesador superescalar es capaz de lanzar varias instrucciones por ciclo (actualmente alrededor de 4), en un único bloque de instrucciones (*instruction stream*). La figura 1.3 muestra como sería la ejecución de instrucciones en un procesador superescalar capaz de lanzar dos instrucciones por ciclo.

Todas las instrucciones del bloque deben ser independientes y satisfacer ciertas condiciones de emisión (*issue*), como por ejemplo que no haya más de un número determinado de referencias a memoria. Si alguna de las instrucciones del bloque es dependiente de otra, o bien no cumple las condiciones de *issue*, sólo las instrucciones precedentes se lanzan a

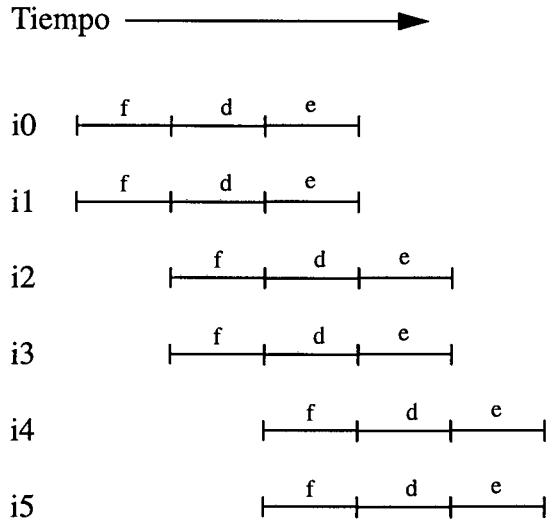

Figura 1.3 : Ejecución superescalar

ejecutar. La Tabla 1.1 muestra el número de instrucciones que se pueden emitir y las restricciones de diversos procesadores superescalares actuales.

El mayor problema de estos procesadores es determinar las dependencias entre operaciones. Un excelente trabajo sobre el diseño y la complejidad de procesadores superescalares es el libro de Johnson [Joh91].

Procesador	Emitidas	L/S	Int ALU	FP ALU	Branch
IBM POWER2	6	2	2	2	2
Power PC620	4	1	1	1	1
HP 8000	4	2	2	2	1
DEC Alpha 21164	4	2	2	2	1
MIPS R10000	4	1	2	2	1
Sun UltraSPARC	4	1	1	1	1
Intel P6	3	1	2	1	1

Tabla 1.1: Características de procesadores superescalares comerciales

Arquitecturas VLIW

En un procesador *Very Long Instruction Word* (VLIW), el compilador tiene la responsabilidad de empaquetar múltiples operaciones que puedan lanzarse simultáneamente en una única instrucción (larga, de ahí el nombre de la arquitectura), de manera que se está decidiendo qué unidades funcionales trabajarán en cada instante. Es pues el compilador quien decide qué operaciones empezarán su ejecución en cada ciclo, sin intervención del hardware (esta planificación se denomina estática, *static scheduling*). En una arquitectura VLIW es importante distinguir entre una instrucción y una operación. Cada ciclo se lanza una única instrucción con múltiples operaciones (figura 1.4).

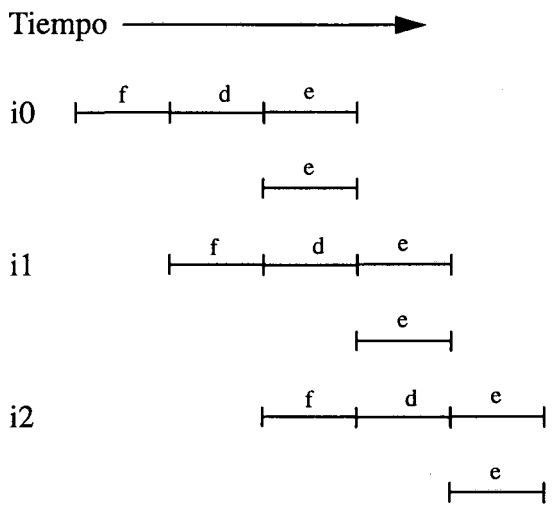

Figura 1.4 : Ejecución VLIW

La primera aproximación a VLIW fue presentada por Charlesworth [Cha81] sobre el diseño del Floating Point Systems AP-120B y estaba orientada a la microprogramación horizontal. El nombre VLIW fue acuñado en el artículo de Fisher [Fis83]. Ejemplos de este tipo de procesadores son el Multiflow [CNO+88, CHJ+90] y el Cydrome Cydra 5 [RYYT89, DHB89, BYA93].

1.4 Dependencias

Ley de Hofstadter: Siempre se tarda más de lo que crees, incluso si tienes en cuenta la ley de Hofstadter.

Douglas Hofstadter, 'Gödel, Escher, Bach', 1979.

Un efecto importante del paralelismo es que, al solaparse las instrucciones, se introducen riesgos debidos a dependencias, tanto de datos como de control, que impiden que se alcance el paralelismo máximo que el hardware puede ofrecer. Las dependencias de control se producen cuando la ejecución de una instrucción depende del resultado de otra (por ejemplo en el control de un bucle, o en instrucciones del tipo if-then-else). Las dependencias de datos se producen cuando los datos de dos instrucciones entran en conflicto y se pueden clasificar en función del orden de los accesos de lectura y escritura de sus operandos. Tomemos dos instrucciones, *i* y *j*, siendo *i* la primera que se ejecutaría en una arquitectura secuencial. Los riesgos son:

- RAW: lectura después de escritura (*read after write*), también llamada dependencia de flujo (*flow dependence*) o verdadera (*true dependence*). La instrucción *j* trata de leer un dato que genera la instrucción *i*. En este caso, la instrucción *j* debe esperar a que *i* genere el dato.

- WAR: escritura después de lectura (*write after read*), también llamada antidependencia (*antidependence*). La instrucción *j* escribe un dato sobre una variable que es leída por *i*. En este caso, la instrucción *j* no debe escribir antes de que la instrucción *i* lea sus datos.

- WAW: escritura después de escritura (*write after write*), también llamada dependencia de salida (*output dependence*). Las instrucciones *i* y *j* escriben en la misma variable. En este caso, la última en escribir debe ser la *j*.

- RAR: lectura después de lectura (*read after read*), también llamada dependencia de entrada (*input dependence*). Ambas instrucciones leen el mismo dato. No ofrece ningún conflicto y no es, por tanto, un riesgo. Sin embargo es una información que puede ser usada por el compilador para efectuar optimizaciones.

1.5 Hardware de soporte para ILP

Leer manuales sobre informática sin el hardware es tan frustrante como leer manuales sobre sexo sin el software.

Arthur C. Clarke

La planificación puede hacerse durante la compilación (planificación estática), durante la ejecución (planificación dinámica) o una combinación de ambas. En el caso de la planificación dinámica, se requiere un cierto soporte hardware, del que hablaremos en este punto.

Scoreboarding

La técnica de *Scoreboard* fue utilizada originalmente en el CDC 6600 [Tho64]. El hardware necesario es: un *buffer* asociado a cada unidad funcional y un marcador (*scoreboard*). El *buffer* se usa para almacenar instrucciones que están esperando operandos (y todos ellos forman la llamada ventana de instrucciones). El marcador es una estructura de control centralizada que controla los registros fuente y destino de cada instrucción encolada en los *buffers*, así como sus dependencias.

El funcionamiento del algoritmo de *scoreboarding* es el siguiente: una instrucción se lee de la memoria (*fetch*) y se lanza mientras hay unidades funcionales disponibles. Si los operandos fuente de la instrucción no están disponibles, la instrucción espera sus operandos almacenada en el buffer de la unidad funcional donde se ejecutará. El lanzamiento de instrucciones se para cuando se intenta lanzar una instrucción a una unidad funcional cuyo *buffer* está ocupado. El marcador detecta el momento en que todos los operandos de una instrucción están disponibles, provocando que la instrucción empiece su ejecución.

Las limitaciones de *scoreboarding* son las siguientes: el lanzamiento de instrucciones se para cuando una unidad funcional es requerida por más de una instrucción o cuando hay una dependencia de salida. Además, el marcador centralizado es complejo de implementar cuando el número de *buffers* es muy grande.

Algoritmo de Tomasulo

El algoritmo de asignación fuera de orden de Tomasulo [Tom67] fue implementado por primera vez en la unidad de coma flotante del IBM 360/91 [IBM67]. El hardware necesario para su implementación es: un bit de ocupado (*busy bit*) y una etiqueta (*tag field*) asociados a cada registro, un conjunto de estaciones de reserva asociado a cada unidad funcional y un bus común de datos. El *busy bit* indica si el valor del registro asociado es válido, o si por el contrario está esperando a cargarse con el resultado de una unidad funcional. El *tag field* indica de qué estación de reserva se recibirá el dato (qué unidad funcional lo está calculando) si el *busy bit* está activo. Las estaciones de reserva forman la ventana de instrucciones, y el bus de datos común conecta la salida de todas las unidades funcionales con todos los registros y estaciones de reserva.

Cuando una instrucción es decodificada, se lanza a la estación de reserva de la unidad funcional donde debería ejecutarse. Los *busy bits* y los *tags fields* de sus registros fuente son enviados a su vez a la estación de reserva donde se almacena la instrucción. Las operaciones que tienen todos los operandos disponibles pasan a estado disponible (*ready*). Cuando la unidad funcional está desocupada escoge una de las operaciones *ready* y la ejecuta. Una vez ejecutada, se envía por el bus de datos común el resultado y el identificador (*tag*), de manera que todas las estaciones de trabajo pueden leer esta información y actualizar sus campos.

El algoritmo de Tomasulo implementa, de hecho, *register renaming* y reordenación de código (lo que elimina las antidependencias y las dependencias de salida). El algoritmo puede ser muy costoso de implementar para una ventana de instrucciones grande, debido al coste de la comparación de los *tag fields*. Además, el *register renaming* produce interrupciones imprecisas debido a la ejecución fuera de orden. Este problema se puede solucionar con el método de *Register Update Unit* [SoVa87, Soh90], una extensión del algoritmo de Tomasulo que garantiza interrupciones precisas.

1.6 Compilación para ILP: planificación y asignación de registros

Life is what happens to you
while you're busy
making other plans.

John Lennon. Beautiful boy (1980).

El compilador realiza una planificación (*scheduling*) de las instrucciones a ejecutar al decidir el orden del código. Una de las premisas de ILP es extraer el máximo de paralelismo en tiempo de compilación, dado que el compilador puede realizar un análisis más exhaustivo del programa que en las pocas instrucciones disponibles en una ventana de instrucciones (nivel hardware). Además, extraer el paralelismo a nivel de compilador permite que el hardware sea más sencillo y, en general, tenga un tiempo de ciclo más corto.

Los algoritmos de planificación pueden clasificarse en función del grupo de operaciones sobre el que actúan. Si un algoritmo actúa sobre un bloque básico, se denomina de planificación local (*local scheduling* [ACD74]). Si usa varios bloques básicos, se denomina planificación global (*global scheduling*). Ejemplos de algoritmos de planificación global son: *trace scheduling* [Fis81, Ell86, LFK+93], *percolation scheduling* [Nic85, EbNa89], *superblock scheduling* [CMC+91, HMC+93], *hyperblock scheduling* [MLC+92] y *boosting* [SLH90, SHL92, Smi92].

Un caso particular de planificación global es el que intenta planificar diversos bloques básicos pertenecientes a distintas iteraciones de un bucle. Estos sistemas reciben el nombre de planificadores cíclicos (*cyclic schedulers*). En el caso de un bucle, los bloques básicos que se deben planificar juntos son copias de la misma pieza de código. Una manera de aplicar planificación sería desenrollar (*unroll*) el cuerpo del bucle un número determinado de veces (llamado el grado de desenrollamiento o *unrolling factor*) y aplicar planificación global sobre el nuevo cuerpo del bucle. Sin embargo, este método no extrae el máximo paralelismo, puesto que planifica un número limitado, aunque ahora mayor, de instrucciones. Una solución a este problema, aunque no aplicable por su complejidad, sería desenrollar el bucle completamente

antes de aplicar el planificador. Las técnicas de segmentación software (*software pipelining)* [Cha81] consiguen el beneficio de un desenrollado total, sin tener que llevarlo a cabo.

La idea principal de la segmentación software es ejecutar un patrón de instrucciones pertenecientes a distintas iteraciones (llamado código núcleo o *kernel code*) de manera que una iteración empieza antes de que finalice la anterior. Esta técnica, cuando se aplica en hardware recibe el nombre de segmentación (*pipelining*); así pues, cuando se aplica a programas, recibe el nombre de segmentación software (*software pipelining*).

El problema de la planificación de programas es NP-completo. Existen métodos de programación lineal [Sch86] que permiten calcular la planificación óptima, como *integer linear programming* [GNvD93] o SPILP [NiGa93, GAG94], pero el coste en tiempo es enorme. Así pues existen diferentes heurísticas para encontrar una planificación pseudo óptima en un tiempo razonable, como el *perfect pipelining* [AiNi88, AiNi91], *UnRolling, Pipelining and Rerolling (URPR)* [SDX86], *global URPR (GURPR*)* [SuWa91] o las técnicas de planificación módulo (*modulo scheduling*) [RaGl81, Lam88, JoAl90], como *Huff Slack Modulo Scheduling* [Huf93], *Hypernode Reduction Modulo Scheduling (HRMS)* [LVAG95, LVAG98] o *Swing Modulo Scheduling* [LGAV96]. En el siguiente capítulo describiremos HRMS, que es el planificador utilizado en este trabajo, y repasaremos los conceptos básicos de planificación software.

Un problema muy relacionado con el de la planificación es la asignación de registros, dado que puede haber mucha presión sobre los registros (es decir, que la relación entre datos que deban ser guardados en los registros, y el número de los mismo sea muy elevada), lo que forzaría que una planificación correcta no fuera viable por necesitar más registros de los disponibles físicamente en la máquina.

La asignación de registros puede hacerse antes o después de la planificación. Independientemente de cuál sea el paso que se ejecute primero, éste toma decisiones sin considerar su impacto sobre el paso que se ejecutará después. Una solución a este problema

sería ejecutar la planificación y la asignación de registros simultáneamente [Pin93]. Pero teniendo en cuenta que ambas técnicas (planificación óptima y asignación de registros óptima) son problemas NP-completos y que se implementan con heurísticas, encontrar dos heurísticas razonables que interactúen es muy difícil. Se han propuesto algoritmos que realizan ambas tareas de forma conjunta, pero hasta ahora sólo para planificación acíclica.

Algunos compiladores implementan primero la asignación de registros con métodos como *graph coloring* [Cha82, BCKT89], seguido de un paso de planificación sobre bloques básicos. Sin embargo, desde el punto de vista de técnicas como planificación software, el problema de la asignación de registros es que introduce antidependencias y dependencias de salida que limitan al planificador, de manera que es interesante hacer primero la planificación y asignar registros después. En este trabajo se sigue un sistema como el planteado en la figura 1.5, en la que se efectúa primero la planificación y después se intenta la asignación de registros. Si no hay suficientes registros, entonces se reduce la presión sobre los registros y se replanifica.

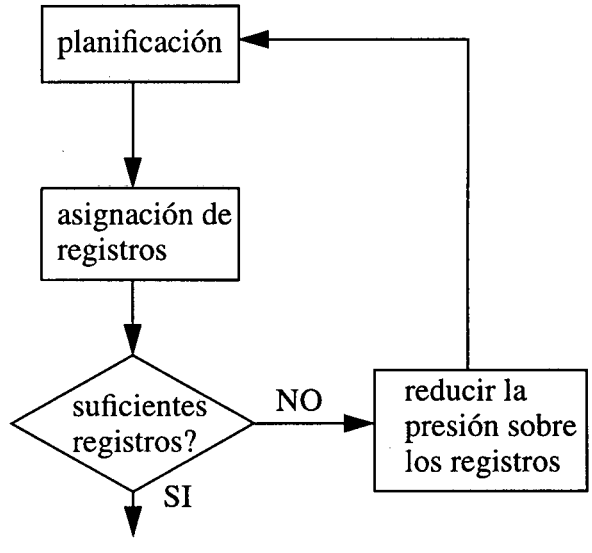

Figura 1.5 : Esquema planificación-asignación de registros

Para reducir la presión sobre los registros, existen diversas alternativas [RLTS92, LAV98]:

- dividir el bucle en dos o más bucles con menos operaciones que el original. En general, los bucles con pocas operaciones tienden a necesitar menos registros; sin embargo, puede darse el caso de que los nuevos bucles necesiten menos ciclos para ser planificados y tengan una presión sobre los registros superior a la del bucle original.

- volver a planificar el bucle incrementando el número de ciclos en los que se va a planificar. Esta técnica se basa en la idea de que la presión sobre los registros es, más o menos, proporcional al número de iteraciones que se están ejecutando concurrentemente. Al incrementar los ciclos usados por la planificación se tiene más libertad a la hora de realizarla, y por tanto se puede reducir la presión sobre los registros.

- acortar el tiempo de vida de un valor (es decir, el intervalo de tiempo en ciclos entre que se genera dicho valor y el momento en que se usa por última vez) enviándolo a memoria durante unos ciclos para liberar el registro (esta técnica recibe el nombre de *spilling* y el código añadido el de *spill code* [Cha82, BGG+89])

En el siguiente capítulo veremos estos temas con un poco más de profundidad. El lector puede encontrar un buen estudio sobre los problemas de registros en la planificación software en [Llo96, LVA96, LAV98].