**Chương 5**

**THỰC THI HỆ THỐNG NHÚNG:**

**ĐỒNG THIẾT KẾ PHẦN CỨNG/PHẦN MỀM**

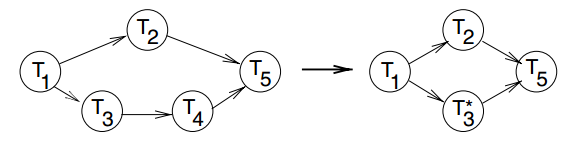
**Lựa chọn không gian thiết kế:** Trong hầu hết các trường hợp, có một vài các thiết kế đáp ứng các đặc tả. Lựa chọn không gian thiết kế là quá trình phân tích tập hợp các thiết kế có thể. Trong số những thiết kế đáp ứng các đặc tả, một thiết kế phải được lựa chọn.

Các luồng thiết kế đặc biệt có thể sử dụng các hoạt động này trong những trình tự khác nhau. Không có bộ tiêu chuẩn cho các hoạt động thiết kế. Chúng tôi sẽ đề cập ngắn gọn một số trình tự đang được sử dụng ở phần cuối của chương này (xem trang 190) để cung cấp một số ý tưởng về các luồng thiết kế thực tế có thể như thế nào.

**5.1 Quản lý đồng thời mức tác vụ**

Như đã đề cập ở trang 52, độ chi tiết của các đồ thị tác vụ là một trong các thuộc tính quan trọng nhất của chúng. Thậm chí đối với các đồ thị tác vụ phân cấp, nó có thể là hữu ích để thay đổi độ chi tiết của các nút. Chia tách các đặc tả thành các tác vụ hoặc quy trình không nhất thiết phải nhắm vào hiệu quả thực thi tối đa. Thay vào đó, trong giai đoạn đặc tả, một sự tách biệt rõ ràng về các mối quan tâm và một mô hình phần mềm sạch quan trọng hơn quan tâm đến việc thực thi quá nhiều. Do đó, sẽ không nhất thiết phải có một sự tương ứng một-một giữa các tác vụ trong đặc tả và chúng trong việc thực thi. Điều này có nghĩa là nên nhóm lại các tác vụ. Việc nhóm lại như vậy thực sự khả thi qua việc sáp nhập và chia tách các tác vụ.

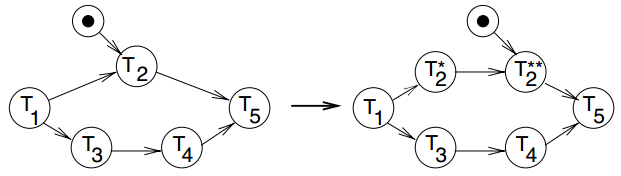
Sự sáp nhập các đồ thị tác vụ có thể được thực hiện bất cứ khi nào mà một số tác vụ Ti là tác vụ ngay trước của một số tác vụ Tj khác và Tj không có bất kỳ tác vụ nào khác ngay trước (xem hình. 5.3 với Ti = T3 and T­j = T4). Sự chuyển đổi này có thể đưa đến việc giảm chi phí trong chuyển ngữ cảnh nếu nút được thực thi trong phần mềm, và nó có thể đưa đến một tiềm năng lớn hơn trong việc tối ưu nói chung.



Hình 5.3. Sáp nhập các tác vụ

Mặt khác, việc chia tách các tác vụ có thể thuận lợi hơn vì những lý do sau:

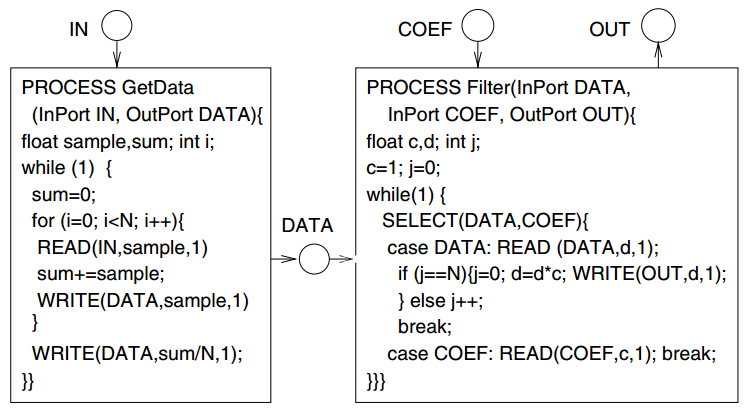
Các tác vụ có thể nắm giữ các tài nguyên (như lượng lớn bộ nhớ) trong khi chúng đang chờ cho một số đầu vào. Để sử dụng tối đa các tài nguyên, tốt nhất nên ràng buộc việc sử dụng các tài nguyên này trong khoảng thời gian mà các tài nguyên này thực sự được sử dụng. Trong hình 5.4, chúng ta giả định rằng tác vụ T2 yêu cầu một số đầu vào ở đâu đó trong mã của nó. Trong phiên bản ban đầu, việc thực hiện các tác vụ T2 chỉ có thể bắt đầu nếu đầu vào này có sẵn. Chúng ta có thể chia nút thành và như vậy đầu vào chỉ cần thiết cho thực hiện. Bây giờ, có thể bắt đầu sớm hơn, dẫn đến lịch trình tự do hơn. Việc tự do lịch trình được cải thiện này có thể cải thiện việc sử dụng tài nguyên và thậm chí có thể kích hoạt tính năng đáp ứng thời hạn nào đó. Nó cũng có thể có ảnh hưởng đến bộ nhớ cần thiết để lưu trữ dữ liệu, vì có thể giải phóng một số bố nhớ của nó và bộ nhớ này có thể được sử dụng bởi các tác vụ khác trong khi đang chờ đợi đầu vào.



Hình 5.4. Chia tách các tác vụ

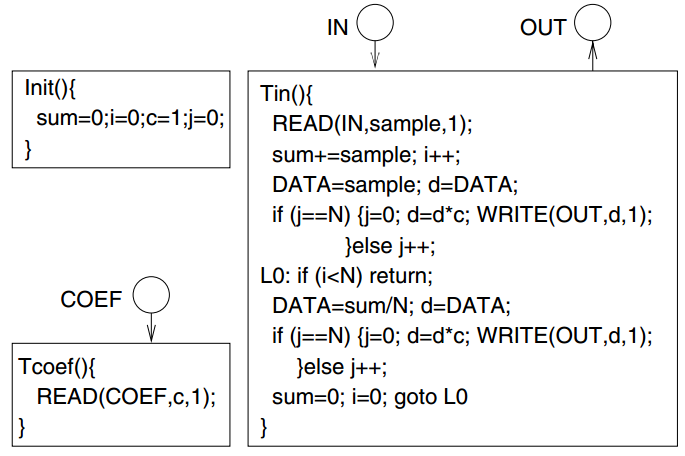
Người ta có thể cho rằng dù sao đi nữa thì các tác vụ nên giải phóng các tài nguyên như một lượng lớn bộ nhớ trước khi chờ đầu vào. Tuy nhiên, tính dễ đọc của các đặc tả ban đầu có thể liên quan đến việc thực thi các vấn đề trong giai đoạn thiết kế sớm.

Các biến đổi rất phức tạp của các đặc tả có thể được thực hiện với một kĩ thuật dựa trên mạng Petri của Cortadella et al. [Cortadella et al., 2000]. Kĩ thuật của họ bắt đầu với một đặc tả bao gồm một tập hợp các tác vụ mô tả trong một ngôn ngữ gọi là FlowC. FlowC mở rộng của C với tiêu đề PROCESS và các giao tiếp tác vụ bên trong nhất định dưới dạng các lời gọi hàm READ and WRITE. Hình 5.5 cho thấy một đặc tả đầu vào sử dụng FlowC.



Hình 5.5. Đặc tả hệ thống

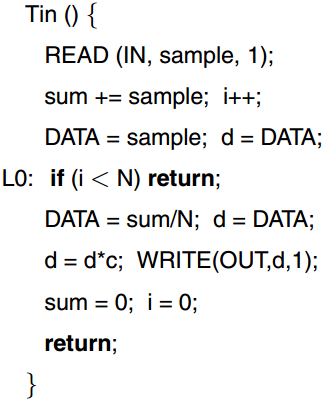
Ví dụ sử dụng các cổng đầu vào **IN** và **COEF**, cũng như các cổng đầu ra **OUT**. Giao tiếp liên tiến trình điểm tới điểm giữa các tiến trình được thực hiện thông qua một kênh đệm **DATA** đơn hướng. Tác vụ **GetData** đọc dữ liệu từ môi trường và gửi nó tới kênh **DATA**. Mỗi lần **N** mẫu được gửi, giá trị trung bình của chúng cũng được gửi thông qua cùng một kênh. Tác vụ **Filter** đọc **N** giá trị từ kênh (và bỏ qua chúng) và sau đó đọc giá trị trung bình, nhân giá trị trung bình với **c** (**c** có thể được đọc vào từ cổng **COEF**) và viết kết quả tới cổng **OUT**. Tham số thứ ba trong lời gọi **READ** và **WRITE** là số lượng các mục được đọc hoặc viết. Các lời gọi **READ** đang bị khóa, lời gọi **WRITE** đang bị khóa nếu số lượng các mục trong kênh xấp xỉ một ngưỡng đã định nghĩa trước. Câu lệnh SELECT có cùng ý nghĩa như như câu lệnh cùng tên trong ADA (xem trang 57): việc thực thi tác vụ này là bị đình chỉ cho đến khi đầu vào tới từ một trong các cổng. Ví dụ này đáp ứng tất cả các tiêu chí trong việc chia tách các tác vụ mà đã được đề cập trong nội dung của hình 5.4. Cả hai tác vụ sẽ chờ cho đầu vào trong khi đang chiếm giữ tài nguyên. Hiệu quả có thể được cải thiện với việc cấu trúc lại các tác vụ này. Tuy nhiên, việc chia tách đơn giản trong hình 5.4 là chưa đủ. Kĩ thuật được đề xuất bởi Cortadella et al. là một kĩ thuật toàn diện hơn. Sử dụng kĩ thuật của họ, trước tiên các chương trình FlowC được dịch sang (được mở rộng) các mạng Petri. Các mạng Petri cho mỗi tác vụ sau đó được sáp nhập thành một mạng Petri đơn. Sử dụng các kết quả từ lý thuyết mạng Petri, các tác vụ mới được sinh ra. Hình 5.6 cho thấy một cấu trúc tác vụ mới có thể.



Hình 5.6. Các tác vụ phần mềm được sinh

Trong cấu trúc tác vụ mới này, có một tác vụ thực hiện tất cả các khởi tạo: Thêm vào đó, có một tác vụ cho mỗi cổng đầu vào. Một thực thi hiệu quả sẽ tăng các lần ngắt mỗi lần đầu vào mới được nhận cho một cổng. Cần một ngắt duy nhất cho mỗi cổng. Các tác vụ sau đó có thể được bắt đầu trực tiếp với các ngắt đó, và sẽ không cần phải gọi hệ điều hành cho điều đó. Việc giao tiếp có thể được thực thi khi một biến toàn cục được chia sẻ duy nhất (đang giả sử một không gian địa chỉ được chia sẻ cho tất cả các tác vụ). Chi phí của hệ điều hành về tổng thể sẽ rất nhỏ, nếu yêu cầu tất cả.

Mã cho tác vụ Tin thể hiện trong hình 5.6 là một tác vụ mà được sinh ra bởi việc tối ưu liên tác vụ của cấu trúc tác vụ dựa trên mạng Petri. Nó cần được tiếp tục tối ưu hóa bằng việc tối ưu nội tác vụ, khi việc kiểm tra được thực hiện trong lần đầu tiên nếu câu lệnh luôn luôn sai (j là bằng i-1 trong trường hợp này, và i và j là được đặt lại về 0 bất cứ khi nào i bằng N). Với câu lệnh thứ hai, kiểm tra luôn luôn đúng, khi điểm điều khiển này chỉ được vươn tới nếu i là bằng với N và i là bằng với j bất cứ khi nào nhãn L0 được vươn tới. Số lượng các biến cũng có thể được giảm thiểu. Sau đây là một phiên bản của Tin đã được tối ưu:



Phiên bản đã tối ưu của Tin có thể được sinh ra bởi một trình biên dịch rất thông minh. Thật không may, hầu như không có trình phiên dịch nào ngày nay sẽ thực hiện việc tối ưu này. Tuy nhiên, ví dụ này cho thấy các loại biến đổi cần thiết để sinh ra các cấu trúc tác vụ “tốt”. Để biết thêm chi tiết về việc sinh tác vụ, tham khảo Cortadella et al. [Cortadella et al., 2000].

Các tối ưu hóa tương tự như thứ vừa trình bày được mô tả trong cuốn sách của Thoen [Thoen và Catthoor, 2000]. Một danh sách của các ẩn phẩm của IMEC về quản lý tác vụ đồng thời có sẵn trên trang web của IMEC [IMEC Desics group, 2003].

**5.2 Các phép tối ưu mức cao**

Có nhiều phép tối ưu mức cao có khả năng cải thiện hiệu quả của phần mềm nhúng.

**5.2.1 Chuyển đổi dấu chấm động sang dấu chấm tĩnh**

Chuyển đổi dấu chấm động sang dấu chấm tĩnh là một kĩ thuật thường được sử dụng. Chuyển đổi này được thúc đẩy bởi một thực tế là nhiều tiêu chuẩn xử lý tín hiệu (như MPEG-2 hoặc MPEC-4) được quy định dưới dạng các chương trình C sử dụng các kiểu dữ liệu dấu chấm động. Nó giúp cho các nhà thiết kế tìm một thực thi hiệu quả cho các tiêu chuẩn này.

Đối với nhiều ứng dụng xử lý tín hiệu, có thể thay thế số dấu chấm động với số dấu chấm tĩnh (xem trang 109). Các lợi ích có thể rất đáng kể. Ví dụ, việc giảm các chu kì đếm xuống 75% and tiêu tốn năng lượng xuống 76% được ghi nhận cho thuật toán nén video MPEG-2 [Hls, 2002]. Tuy nhiên, có một vài sự thiếu chính xác thường phát sinh. Chính xác hơn, có một sự cân bằng giữa chi phí của việc thực thi và chất lượng của thuật toán (ví dụ được đánh giá thông qua cái được gọi là tỉ số tín hiệu trên nhiễu (SNR)). Với các độ dài từ ngắn, chất lượng có thể bị ảnh hưởng nghiêm trọng. Do đó, kiểu dữ liệu dấu chấm động có thể được thay thế với kiểu dữ liệu dấu chấm tĩnh, nhưng sự mất mát chất lượng cũng cần phải được phân tích. Việc thay thế này ban đầu được thực hiện bằng tay. Tuy nhiên, nó là một quá trình này rất tẻ nhạt và dễ bị lỗi.

Do đó, các nhà nghiên cứu đã cố gắng để hỗ trợ việc thay thế này với các công cụ. Một trong các công cụ được biết đến nhiều nhất là FRIDGE (môi trường thiết kế lập trình dấu chấm động) [Willems et al., 1997], [Keding et al., 1998]. Các công cụ FRIDGE được thực hiện thương mại như bộ phận của gói công cụ Synopsys System Studio [Synopsys, 2005].

Trong FRIDGE, tiến trình thiết kế bắt đầu với một thuật toán được mô tả trong C, bao gồm các số dấu chấm động. Thuật toán này sau đó được chuyển đổi sang một thuật toán được môt tả trong **fixed-C**. Fixed-C mở rộng của C với hai kiểu dữ liệu dấu chấm động, sử dụng các đặc trưng định nghĩa kiểu của C++. Fixed-C là một tập hợp con của C++ và cung cấp hai kiểu dữ liệu fixed và Fixed. Các kiểu dữ liệu dấu chấm tĩnh có thể được khai báo rất giống các biến khác. Khai báo sau đây khai báo một biến vô hướng, một con trỏ, và một mảng kiểu dữ liệu dấu chấm tĩnh.

fixed a, \*b, c[8]

Cung cấp các tham số của các loại dữ liệu dấu chấm tĩnh có thể (nhưng không phải) được trì hoãn cho đến lúc gán:

a=fixed(5,4,s,wt,\*b)

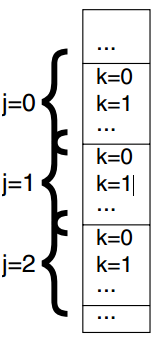
Phép gán này đặt tham số độ dài từ của biến a là 5 bit, độ dài từ phần thập phân là 4 bit, dấu để thể hiện (s), xử lý tràn với wrap-around (w), và mode làm tròn với truncation (t). Các tham số cho các biến mà được đọc trong một phép gán là được xác định bởi phép gán(s) với các biến đó. Kiểu dữ liệu **Fixed** là tương tự **fixed**, ngoại trừ việc kiểm tra tính nhất quán giữa các tham số được sử dụng trong khai báo và trong phép gán được thực hiện. Với mỗi phép gán với một biến, các tham số (bao gồm độ dài từ) có thể khác nhau. Thông tin tham số này có thể được bổ sung vào chương trình-C nguồn trước khi ứng dụng được mô phỏng. Việc mô phỏng cung cấp các phạm vi giá trị cho tất cả các phép gán. Dựa trên các thông tin đó, FRIDGE bổ sung các thông tin tham số tới tất cả các phép gán. FRIDGE cũng suy diễn các thông tin tham số từ ngữ cảnh. Cho ví dụ, giá trị tối đa của các bổ sung được coi là tổng của các đối số. Thông tin tham số được bổ sung có thể dựa trên sự mô phỏng hoặc trên các cân nhắc trường hợp xấu nhất. Được dựa trên các mô phỏng, FRIDGE không cần giả định các giá trị trường hợp xấu nhất sẽ là kết quả từ một phân tích chính thức. Kết quả chương trình C++ được mô phỏng lại để kiểm tra sự mất mát chất lượng. Phiên bản Synopsys của Fridge sử dụng các kiểu dữ liệu dấu chấm tĩnh SystemC để thể hiện thông tin kiểu dữ liệu được sinh ra. Theo đó, System C có thể được sử dụng trong mô phỏng kiểu dữ liệu dấu chấm tĩnh.

Một phân tích về sự cân bằng giữa nhiễu bổ sung được giới thiệu và độ dài từ cần thiết đã được để xuất bởi Shi và Brodersen [Shi và Brodersen, 2003] và cũng bởi Menard et al. [Menard và Sentieys, 2002].

**5.2.2 Các chuyển đổi vòng lặp đơn**

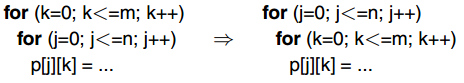
Đó là có một số các chuyển đổi vòng lặp mà có thể được áp dụng đối với các đặc tả. Sau đây là danh sách của các chuyển đổi vòng lặp tiêu chuẩn:

* **Hoán vị vòng lặp:** Xem xét một mảng hai chiều. Theo tiêu chuẩn **C** [Kernighan và Ritchie, 1988], các mảng hai chiều được bố trí trong bộ nhớ như trong hình 5.7. Các giá trị chỉ số liền kề của chỉ số thứ hai được ánh xạ tới một khối liền tiếp các vị trí trong bộ nhớ. Cách bố trí này được gọi là thứ tự ưu tiên hàng [Muchnick, 1997]. Chú ý rằng việc bố trí cho các mảng là khác nhau trong FORTRAN: các giá trị liền kề của chỉ số đầu tiên là được ánh xạ tới một khối các vị trí liên tiếp trong bộ nhớ (thứ tự ưu tiên cột). Các ấn phẩm mô tả các tối ưu hóa trong FORTRAN có thể gây nhầm lẫn.



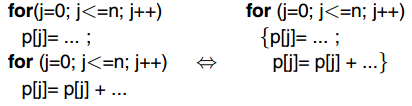
Hình 5.7. Bố trí bộ nhớ cho mảng hai chiều p[j][k] trong **C**

Bố trí ưu tiên hàng thường có lợi cho tổ chức các vòng lặp mà chỉ số cuối cùng tương ứng với vòng lặp trong cùng. Một hoán vị vòng lặp tương đương được thể hiện trong ví dụ sau đây:



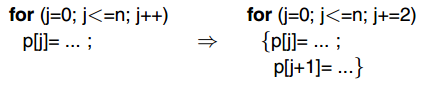
Các hoán vị như vậy có thể có một tác động tích cực vào việc tái sử dụng các phần tử của mảng trong bố nhớ đệm, khi bộ duyệt kế tiếp của thân vòng lặp truy cập một ví trí liền kề trong bộ nhớ. Các bộ nhớ đệm thường được tổ chức để mà các vị trí liền kề có thể truy cập nhanh hơn đáng kể so với các vị trí ở xa hơn vị trí đã truy cập trước đó.

* **Tổ hợp vòng lặp, phân rã vòng lặp:** Có nhiều trường hợp hai vòng lặp riêng rẽ có thể được tổ hợp lại, và cũng có những trường hợp một vòng lặp đơn được tách thành hai. Sau đây là một ví dụ:



Phiên bản bên trái có thể có lợi nếu bộ xử lý mục tiêu cung cấp một lệnh lặp với chi phí bằng không mà có thể chỉ được sử dụng trong các vòng lặp nhỏ. Phiên bản bên phải giúp cải thiện bộ nhớ đệm (do vị trí của các tham chiếu tới mảng **p** được cải thiện), và cũng tăng khả năng tính toán song song trong thân vòng lặp. Như rất nhiều chuyển đổi khác, nó là rất khó để biết chuyển đổi nào dẫn tới mã tốt nhất.

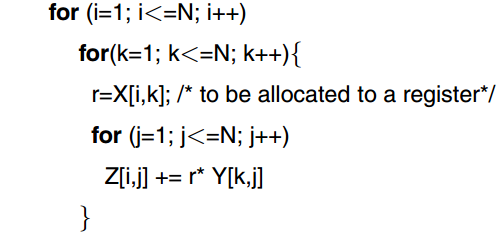
* **Dãn vòng lặp – Loop unrolling:** Loop unrolling là một chuyển đổi tiêu chuẩn tạo ra một vài thể hiện trong thân vòng lặp. Sau đây là ví dụ khi vòng lặp được dãn ra:



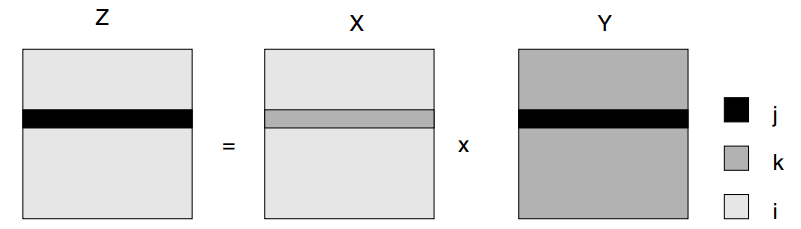
Số lượng các bản sao của vòng lặp được gọi là **yếu tố dãn – unrolling factor**. Unrolling factor có thể lớn hơn 2. Unrolling giảm chi phí vòng lặp (các nhánh trên mỗi xử lý của thân vòng lặp là ít nhất) và do đó thường làm tăng tốc độ. Một trường hợp đặc biệt, các vòng lặp có thể đã không được cuộn hoàn toàn, loại bỏ hoàn toàn chi phí điều khiển và các nhánh. Tuy nhiên unrolling làm tăng kích thước mã. Unrolling thường được giới hạn với các vòng lặp có một số hữu hạn các lần lặp.

**5.2.3 Xếp chồng/tạo khối vòng lặp - Loop tiling/blocking**

Có thể quan sát được rằng tốc độ bộ nhớ đang tăng ở tỉ lệ chậm hơn so với bộ vi xử lý. Từ khi các bộ nhớ nhỏ nhanh hơn các bộ nhớ lớn (xem trang 118), việc sử dụng các bộ nhớ phân cấp có thể có nhiều lợi ích. Các bộ nhớ nhỏ bao gồm bộ nhớ đệm - cache và bộ nhớ tạm – scratch-pad. Yếu tố tái sử dụng đáng kể các thông tin trong các bộ nhớ đó là cần thiết. Nếu không, bộ nhớ phân cấp không thể được khai thác hiệu quả. Hiệu quả của vệc tái sử dụng có thể được minh họa thông qua việc phân tích ví dụ sau. Chúng ta hãy xem xét việc nhân ma trận với các mảng kích thước **N × N** [Lam et al., 1991]:



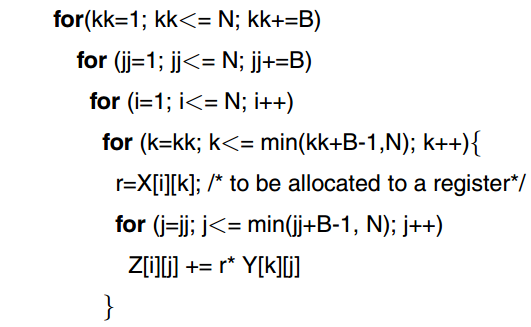
Chúng ta hãy xem xét các mô hình truy cập cho đoạn mã này. Phần tử chung X[i,k] được sử dụng bởi tất cả các lần lặp của vòng lặp trong cùng. Trình biên dịch thông thường sẽ có khả năng để phân bổ phần tử này tới một thanh ghi và tái sử dụng nó trong mỗi lần xử lý của vòng lặp trong cùng. Chúng ta giả định rằng các phần tử mảng được đặt ở trong một thứ tự ưu tiên hàng (vì nó là tiêu chuẩn của **C**). Điều này có nghĩa là các phần tử mảng đó với giá trị chỉ số hàng liền kề (bên phải nhất) được lưu trữ trong các vị trí bộ nhớ liền kề. Theo đó, các vị trí liền kề của **Z** và **Y** được lấy ra trong các lần lặp của vòng lặp trong cùng. Thuộc tính này là có lợi nếu hệ thống bộ nhớ sử dụng tìm nạp trước (bất cứ khi nào một từ được nạp vào bộ nhớ đệm, việc nạp tới từ kế tiếp cũng được bắt đầu).



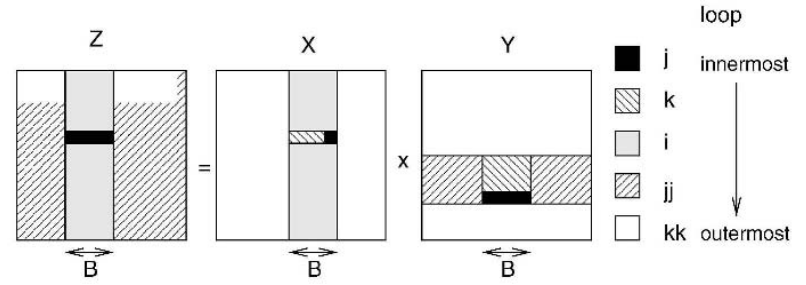
Hình 5.8. Mô hình truy cập cho nhân ma trận không khóa

Với một lần lặp của vòng lặp trong cùng, các vùng đen của mảng **Z** và **Y** được truy cập (và nạp vào bộ nhớ đệm). Cùng một thông tin có hay không có thì vẫn ở trong bộ nhớ đệm cho lần lặp tiếp ở các vòng lặp giữa hoặc ngoài cùng phụ thuộc vào kích thước của bộ nhớ đệm. Trong trường hợp tồi nhất (nếu **N** là lớn hoặc bộ nhớ đệm quá nhỏ), thông tin phải được nạp lại cho mỗi lần xử lý của vòng lặp trong cùng và các phần tử bộ nhớ đệm không được tái sử dụng. Tổng số các tham chiếu bộ nhớ có thể lên tới **2 N3** (với các tham chiếu tới **Z** và **Y**) + **N2** (cho các tham chiếu tới **X**).

Nghiên cứu trong tính toán khoa học dẫn tới việc thiết kế các thuật toán xếp chồng và tạo khối [Xue, 2000], nó cải thiện tính cục bộ của các tham chiếu. Sau đây là phiên bản đã xếp chồng của thuật toán trên:



Hình 5.9 chỉ ra mô hình truy cập tương ứng.



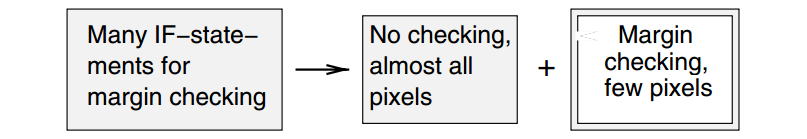
Hình 5.9. Mô hình truy cập cho nhân ma trận đã chồng/tạo khối

Vòng lặp trong cùng hiện tại bị hạn chế vì vậy nó truy cập tới ít phần tử mảng hơn (chúng chỉ ra bằng màu đen). Nếu một yếu tố khối thích hợp được lựa chọn, các phần tử vẫn ở trong bộ nhớ đệm khi lần lặp kế tiếp của vòng lặp trong cùng bắt đầu. Yếu tố khối **B** có thể được chọn để các phần tử của các vòng lặp trong cùng vừa với bộ nhớ đệm. Đặc biệt, nó có thể chọn được một ma trận con **B × B** của **Y** vừa với bộ nhớ đệm. Điều này tương ứng với việc tái sử dụng các yếu tố của **B** cho **Y**, khi các phần tử trong ma trận con được truy cập **B** lần cho mỗi lần gọi **i**. Một khối các phần tử hàng **B** phải vừa với bộ nhớ đệm. Chúng sau đó sẽ được tái sử dụng qua các lần lặp của **k**, kết quả các yếu tố của **B** cũng được tái sử dụng cho **Z**. Điều này sẽ giảm tổng số các tham chiếu bộ nhớ nhiều nhất là **2 N3/B** (cho các tham chiếu tới **Z** và **Y**) + **N2** (cho các tham chiếu tới **X**). Trong thực tế, các yếu tố tái sử dụng có thể ít hơn **B**. Việc tối ưu các yếu tố tái sử dụng đã trở thành một lĩnh vực nghiên cứu toàn diện. Các nghiên cứu ban đầu quan tâm tới việc cải tiến hiệu suất mà có thể được thực hiện bởi xếp chồng. Cải tiến hiệu suất cho nhân ma trận với một hệ số giữa 3 và 4.3 được báo cáo bởi Lam [Lam et al., 1991]. Các cải tiến được mong đợi sẽ gia tăng với khoảng cách ngày càng tăng giữa tốc độ bộ vi xử lý và bộ nhớ. Xếp chồng cũng có thể giảm năng lượng tiêu tốn của hệ thống bộ nhớ [Chung et al., 2001].

**5.2.4 Chia tách vòng lặp**

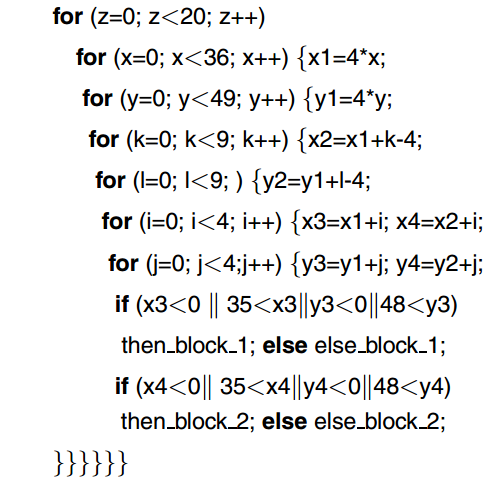
Tiếp theo, chúng ta sẽ thảo luận về việc chia tách vòng lặp như một cách tối ưu hóa khác có thể được áp dụng trước khi biên dịch chương trình. Có khả năng, việc tối ưu này cũng sẽ được thêm vào các trình biên dịch.

Nhiều thuật toán xử lý ảnh thực hiện một vài kiểu lọc. Việc lọc này bao gồm việc xem xét các thông tin về một điểm ảnh nào đó cũng như các hàng xóm của nó. Các tính toán tương ứng thường khá bình thường. Tuy nhiên, nếu các điểm ảnh được xem xét gần với biên của ảnh, không phải tất cả các điểm ảnh hàng xóm đều tồn tại và việc tính toán phải được sửa đổi. Trong một mô tả đơn giản của thuật toán lọc, những thay đổi có thể dẫn đến các kiểm tra được thực hiện trong các vòng lặp trong cùng của thuật toán. Một phiên bản hiệu quả hơn của thuật toán có thể được sinh ra với việc chia tách các vòng lặp sao cho một thân vòng lặp xử lý các trường hợp thông thường và thân vòng lặp thứ hai xử lý các trường hợp ngoại lệ. Hình 5.10 là một thể hiện đồ họa cho chuyển đổi này.

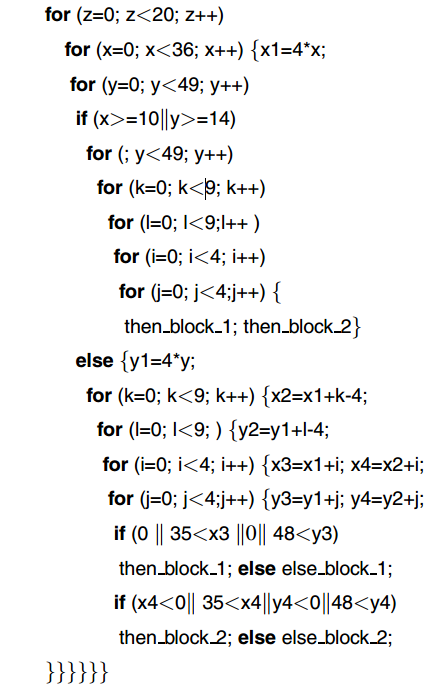


Hình 5.10. Chia tách quá trình xử lý ảnh vào các trường hợp thông thường và đặc biệt

Việc thực hiện chia tách vòng lặp bằng tay rất khó khăn và thủ tục dễ bị lỗi. Falk et al. đã công bố một thuật toán [Falk và Marwedel, 2003] để thực hiện một thủ tục mà cũng làm việc cho các kích thước lớn hơn một cách tự động. Nó được dựa trên một sự phân tích phức tạp của các truy cập tới các phần tử của mảng trong các vòng lặp. Các giải pháp tối ưu hóa được tạo ra bằng cách sử dụng các thuật toán di truyền. Đoạn mã sau cho thấy một nhóm vòng lặp từ tiêu chuẩn MPEG-4 thực hiện ước tính chuyển động.



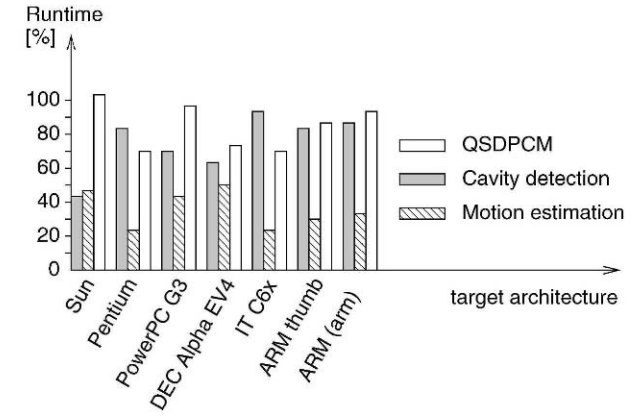
Sử dụng thuật toán của Falk, nhóm vòng lặp này được chuyển đổi thành như sau:



Thay vì các kiểm tra phức tạp ở vòng lặp trong cùng, chúng ta bây giờ có một câu lệnh điều kiện if chia nhỏ sau câu lệnh lặp for thứ 3. Tất cả các trường hợp thông thường được xử lý trong phần sau của câu lệnh này. Phần kia xử lý một số tương đối nhỏ các trường hợp còn lại.

Hình 5.11 chỉ ra số các vòng lặp có thể tiết kiệm được với việc chia tách nhóm vòng lặp trong rất nhiều các ứng dụng và bộ vi xử lý mục tiêu.

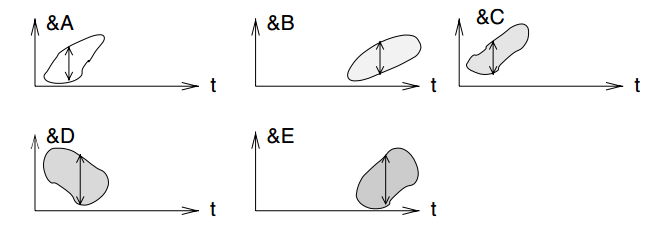
Với thuật toán ước tính chuyển động, số lượng vòng lặp có thể được giảm lên đến 75% (bằng 25% của giá trị ban đầu). Rõ ràng, việc tiết kiệm đáng kể là có thể. Tiềm năng này chắc chắn không nên bị bỏ qua.



Hình 5.11. Kết quả của việc chia tách vòng lặp

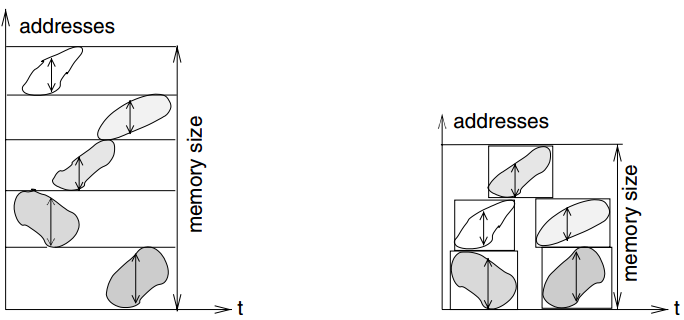
**5.2.5 Nén mảng – Array folding**

Một vài ứng dụng nhúng, đặc biệt trong lĩnh vực đa phương tiện, bao gồm các mảng lớn. Kể từ khi không gian bộ nhớ trong các hệ thống nhúng bị giới hạn, các tùy chọn cho việc giảm thiểu các yêu cầu bộ nhớ lưu trữ của các mảng cần được xem xét. Hình 5.12 thể hiện các địa chỉ được sử dụng bởi năm mảng như một hàm của thời gian. Ở bất cứ thời gian cụ thể nào chỉ một tập con của các phần tử mảng là cần thiết. Số lượng tối đa của các phần tử cần thiết được gọi là cửa sổ tham chiếu địa chỉ [De Greef et al., 1997a]. Trong hình 5.12, lượng tối đa này được chỉ ra bởi một mũi tên hai đầu.



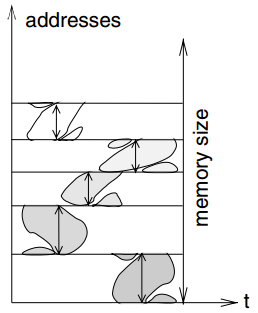
Hình 5.12. Các mô hình tham chiếu cho các mảng

Một cấp phát bộ nhớ cho mảng kiểu cổ điển được chỉ ra trong hình 5.13 (bên trái). Mỗi mảng được cấp phát tối đa không gian nó yêu cầu qua toàn bộ thời gian xử lý (nếu chúng ta xem xét các mảng toàn cục).



Hình 5.13 Các mảng chưa được nén (bên trái) và được nén liên mảng (bên phải)

Một trong những cải tiến có thể, việc nén liên mảng, được chỉ ra trong hình 5.13 (bên phải). Ở các khoảng thời gian không trùng lặp, các mảng có thể cùng chia sẻ không gian bộ nhớ. Một cải tiến thứ hai, việc nén nội mảng [De Greef et al., 1997b], được chỉ ra trong hình 5.14. Nó tận dụng các tập giới hạn những thành phần cần thiết **bên trong** một mảng. Bộ nhớ lưu trữ có thể được tiết kiệm ở chi phí cho các tính toán địa chỉ phức tạp hơn.



Hình 5.14. Các mảng đã nén nội mảng

Hải loại nén cũng có thể kết hợp lại.

Các dạng khác của chuyển đổi ở mức cao được phân tích bởi Chung, Benini và De Micheli [Chung et al., 2001], [Tan et al., 2003]. Có nhiều đóng góp bổ sung cho lĩnh vực này trong cộng đồng biên dịch.

Đặc biệt, hàm nội tuyến - inlining1 thay thế các lời gọi hàm với mã của các hàm đã được gọi. Sự chuyển đổi này cải thiện tốc độ của mã, nhưng kết quả làm gia tăng kích thước mã. Kích thước mã gia tăng có thể là vấn đề trong các công nghệ **SoC**. Các kĩ thuật in-lining truyền thống dựa vào người dùng khởi tạo các hàm nội tuyến. Đây cũng là một vấn đề trong hệ thống trên một con chip, khi kích thước của bộ nhớ lệnh rất quan trọng với các hệ thống như vậy. Do đó, điều quan trọng là khả năng ràng buộc kích thước bộ nhớ lệnh và cho phép thiết kế các công cụ tìm kiếm tự động các hàm cần nội tuyến trong một kích thước bộ nhớ nhất định. Các phương pháp được biết đến cho điều này bao gồm các kĩ thuật của Teich [Teich et al., 1999], Leupers et al. [Leupers và Marwedel, 1999], và [Palkovic et al., 2002]. Các kĩ thuật đó có thể được tích hợp vào một trình biên dịch hoặc có thể được áp dụng như một chuyển đổi mã nguồn tới mã nguồn trước khi sử dụng bất kì trình biên dịch nào.

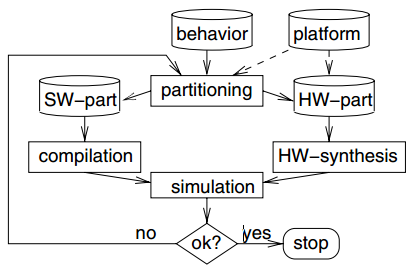
**5.3 Phân vùng phần cứng/phần mềm**

**5.3.1 Giới thiệu**

Trong quá trình thiết kế, chúng ta phải giải quyết các vấn đề thực hiện các đặc điểm kĩ thuật trong phần cứng hay dưới dạng các chương trình đang chạy trên bộ vi xử lý. Phần này mô tả một vài kĩ thuật cho việc ánh xạ này. Áp dụng các kĩ thuật này, chúng ta có thể để quyết định các phần phải thực hiện trong phần cứng hay trong phần mềm.

Với phân vùng phần cứng/phần mềm chúng ta nói đến việc ánh xạ các nút đồ thị tác vụ với phần cứng hoặc phần mềm. Một thủ tục tiêu chuẩn cho việc nhúng phân vùng phần cứng/phần mềm vào luồng thiết kế tổng thể được chỉ ra trong hình 5.15. Chúng ta bắt đầu từ một thể hiện chung của đặc tả, ví dụ trong dạng thức của đồ thị tác vụ và thông tin về nền tảng.

Với mỗi nút của các đồ thị tác vụ, chúng ta cần các thông tin liên quan tới các nỗ lực cần thiết và lợi ích nhận được từ việc lựa chọn một cách thực hiện nhất định của các nút này. Ví dụ, số lần xử lý cần phải được dự đoán trước (xem trang 127). Rất khó để dự đoán số lần cần thiết cho giao tiếp. Tuy nhiên, hai nhiệm vụ đang yêu cầu giao tiếp băng thông rất cao tốt hơn nên được ánh xạ tới các thành phần chung. Các phương pháp lặp được sử dụng trong nhiều trường hợp. Một giải pháp ban đầu cho vấn đề phân vùng được tạo ra, được phân tích và sau đó được cải thiện.



Hình 5.15. Cái nhìn tổng thể về phân vùng phần cứng/phần mềm

Một vài phương pháp phân vùng giới hạn việc ánh xạ các nút đồ thị tới phần cứng mục đích đặc biệt hoặc phần mềm đang chạy trên một bộ vi xử lý đơn. Các phân vùng như vậy có thể được thực hiện với các thuật toán phân vùng đôi cho đồ thị [Kuchcinski, 2002].

Các thuật toán phân vùng phức tạp hơn có khả năng ánh xạ đồ các nút đồ thị tới các hệ thống đa xử lý và phần cứng. Sau đây, chúng ta sẽ mô tả cách thức cái này có thể thực hiện được khi sử dụng một kĩ thuật tối ưu hóa chuẩn từ các hoạt động nghiên cứu, **lập trình số nguyên – integer programming**. Trình bày của chúng ta dựa trên một phiên bản đơn giản hóa của việc tối ưu được đề xuất cho công cụ đồng thiết kế (co-design tool) COOL [Niemann, 1998].

**5.3.2 COOL**

Đối với COOL, đầu vào bao gồm ba phần:

**Công nghệ mục tiêu:** Phần đầu vào này của COOL bao gồm thông tin về các thành phần nền tảng phần cứng sẵn có. COOL đáp ứng các hệ thống đa xử lý, nhưng yêu cầu tất cả các bộ vi xử lý là cùng loại, khi đó nó không bao gồm việc lựa chọn bộ vi xử lý bằng tay hay tự động. Tên của bộ vi xử lý được sử dụng (cũng như những thông tin về trình biên dịch tương ứng) phải bao gồm trong phần đầu vào này của COOL. Theo như phần cứng dành riêng cho ứng dụng được xem xét, thông tin phải đầy đủ để tổng hợp phần cứng tự động với tất cả các tham số được yêu cầu. Đặc biệt, thông tin về thư viện công nghệ phải được cung cấp.

**Các ràng buộc về thiết kế:** Phần thứ hai của đầu vào bao gồm các ràng buộc thiết kế như lưu lượng được yêu cầu, đỗ trễ, kích thước bộ nhớ tối đa, hoặc diện tích tối đa cho phần cứng dành riêng cho ứng dụng.