**Chương 5**

**THỰC THI HỆ THỐNG NHÚNG:**

**ĐỒNG THIẾT KẾ PHẦN CỨNG/PHẦN MỀM**

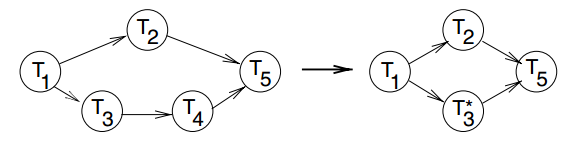
**Lựa chọn không gian thiết kế:** Trong hầu hết các trường hợp, có một vài các thiết kế đáp ứng các đặc tả. Lựa chọn không gian thiết kế là quá trình phân tích tập hợp các thiết kế có thể. Trong số những thiết kế đáp ứng các đặc tả, một thiết kế phải được lựa chọn.

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.

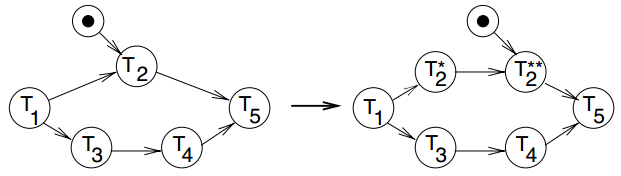
Sự sáp nhập các đồ thị tác vụ có thể được thực hiện bất cứ khi nào mà một số tác vụ Ti là tác vụ ngay trước của một số tác vụ Tj khác và Tj không có bất kỳ tác vụ nào khác ngay trước (xem hình. 5.3 với Ti = T3 and T­j = T4). Sự chuyển đổi này có thể đưa đến việc giảm chi phí trong chuyển ngữ cảnh nếu 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:

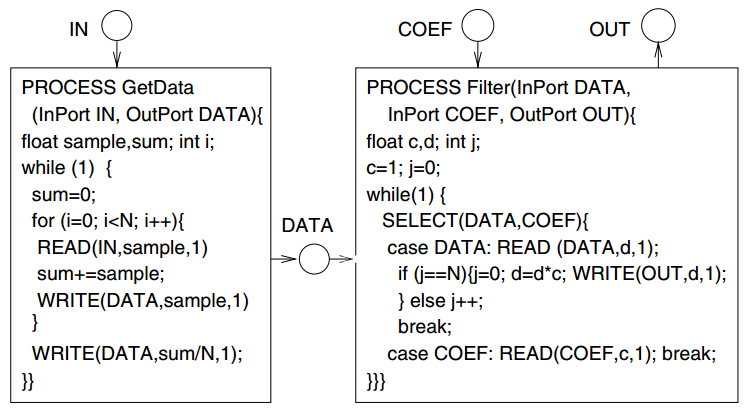
Các tác vụ có thể nắm giữ các tài nguyên (như lượng lớn bộ nhớ) trong khi chúng đang chờ cho một số đầu vào. Để sử dụng tối đa các tài nguyên, tốt nhất nên ràng buộc việc sử dụng các tài nguyên này trong khoảng thời gian mà các tài nguyên này thực sự được sử dụng. Trong hình 5.4, chúng ta giả định rằng 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

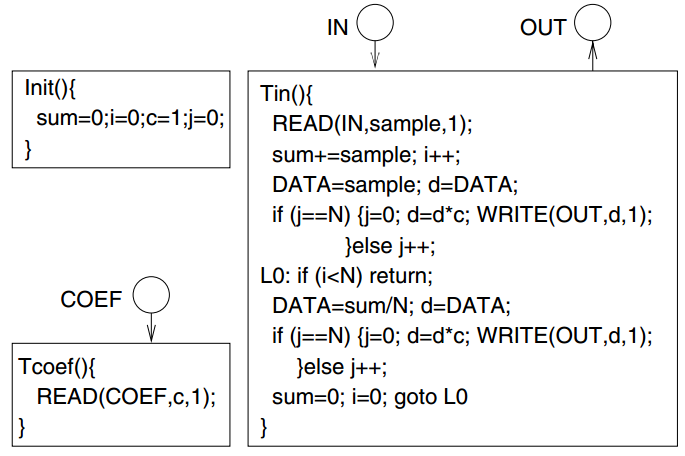
Người ta có thể cho rằng dù sao đi nữa thì các tác vụ nên giải phóng các tài nguyên như một lượng lớn bộ nhớ trước khi chờ đầu vào. Tuy nhiên, tính dễ đọc của các đặc tả ban đầu có thể liên quan đến việc thực thi các vấn đề trong giai đoạn thiết kế sớm.

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

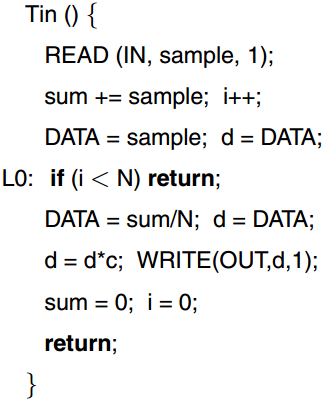
Ví dụ sử dụng các cổng đầu vào **IN** và **COEF**, cũng như các cổng đầu ra **OUT**. Giao tiếp liên tiến trình điểm tới điểm giữa các tiến trình được thực hiện thông qua một kênh đệm **DATA** đơn hướng. Tác vụ **GetData** đọc dữ liệu từ môi trường và gửi nó tới kênh **DATA**. Mỗi lần **N** mẫu được gửi, 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)

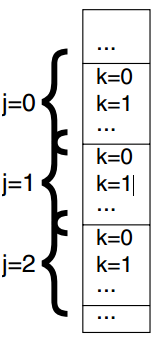
Phép gán này đặt tham số độ dài từ của biến a là 5 bit, độ dài từ phần thập phân là 4 bit, dấu để thể hiện (s), xử lý tràn với wrap-around (w), và mode làm tròn với truncation (t). Các tham số cho các biến mà được đọc trong một phép gán là được xác định bởi phép gán(s) với các biến đó. Kiểu dữ liệu **Fixed** 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**

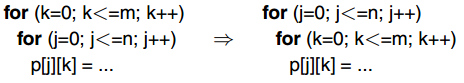
Đó là có một số các chuyển đổi vòng lặp mà có thể được áp dụng đối với các đặc tả. Sau đây là danh sách của các chuyển đổi vòng lặp tiêu chuẩn:

* **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.



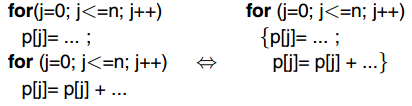
Hình 5.7. Bố trí bộ nhớ cho mảng hai chiều p[j][k] trong **C**

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:



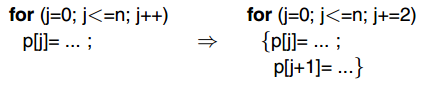
Các hoán vị như vậy có thể có một tác động tích cực vào việc tái sử dụng các phần tử của mảng trong bố nhớ đệm, khi bộ duyệt kế tiếp của thân vòng lặp truy cập một ví trí liền kề trong bộ nhớ. Các bộ nhớ đệm thường được tổ chức để mà các vị trí liền kề có thể truy cập nhanh hơn đáng kể so với các vị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

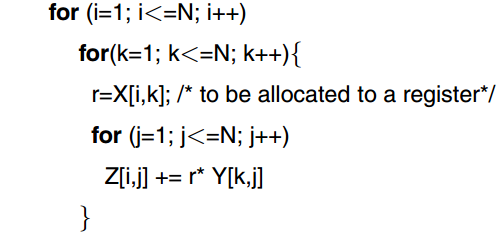
* **Dãn vòng lặp – Loop unrolling:** Loop unrolling là một chuyển đổi tiêu chuẩn tạo ra một vài thể hiện trong thân vòng lặp. Sau đây là ví dụ khi vòng lặp được dãn ra:



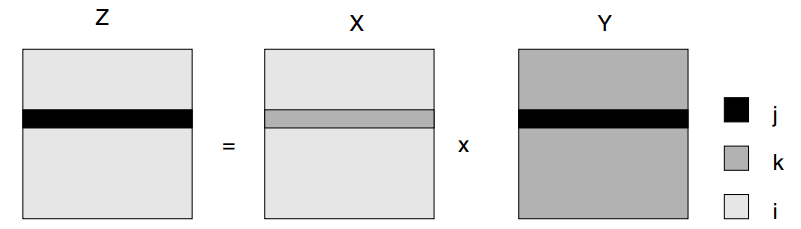
Số lượng các bản sao của vòng lặp được gọi là **yếu tố dãn – unrolling factor**. Unrolling factor có thể lớn hơn 2. Unrolling giảm chi phí vòng lặp (các nhánh trên mỗi xử lý của thân vòng lặp là ít nhất) và do đó thường làm tăng tốc độ. Một trường hợp đặc biệt, các vòng lặp có thể đã không được cuộn 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]:



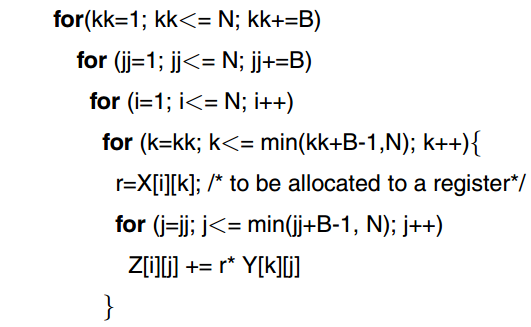
Chúng ta hãy xem xét các mô hình truy cập cho đoạn mã này. Phần tử chung X[i,k] được sử dụng bởi tất cả các lần lặp của vòng lặp trong cùng. Trình biên dịch thông thường sẽ có khả năng để phân bổ phần tử này tới một thanh ghi và tái sử dụng nó trong mỗi lần xử lý của vòng lặp trong cùng. Chúng ta giả 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).



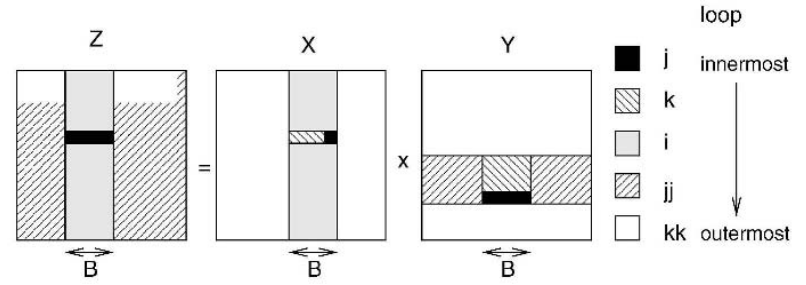
Hình 5.8. Mô hình truy cập cho nhân ma trận không khóa

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.



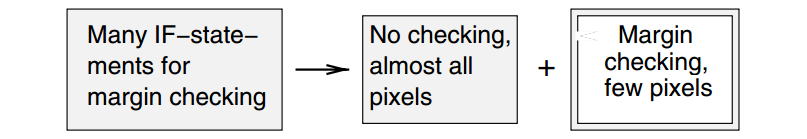
Hình 5.9. Mô hình truy cập cho nhân ma trận đã chồng/tạo khối

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**

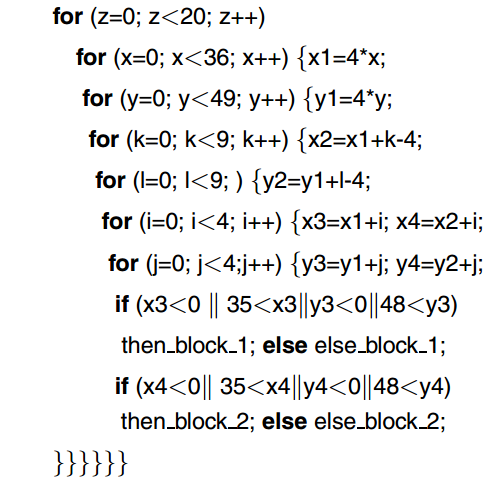
Tiếp theo, chúng ta sẽ thảo luận về việc chia tách vòng lặp như một cách tối ưu hóa khác có thể được áp dụng trước khi biên dịch chương trình. Có khả năng, việc tối ưu này cũng sẽ được thêm vào các trình biên dịch.

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.

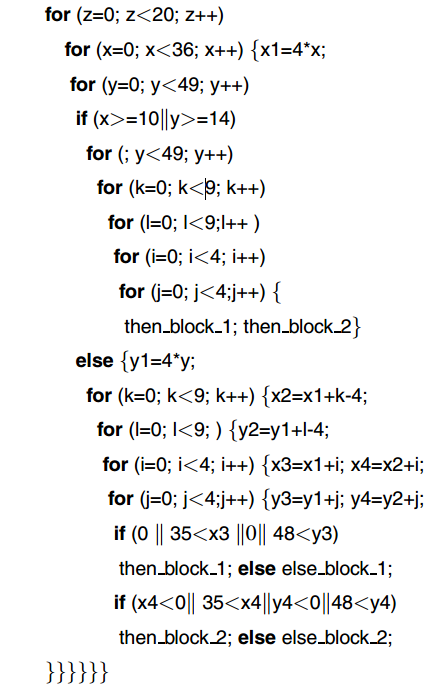


Hình 5.10. Chia tách quá trình xử lý ảnh vào các trường hợp thông thường và đặc biệt

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.



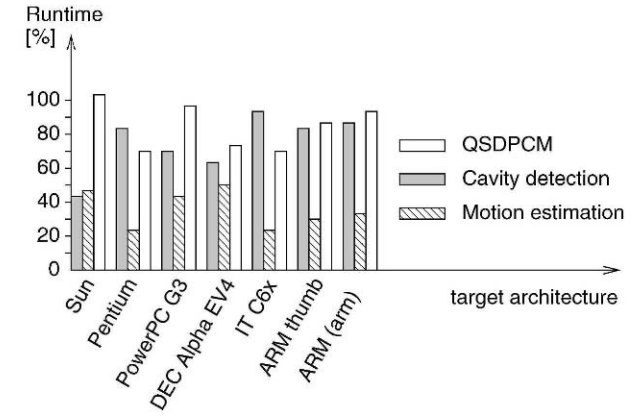
Sử dụng thuật toán của Falk, nhóm vòng lặp này được chuyển đổi thành như sau:



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.

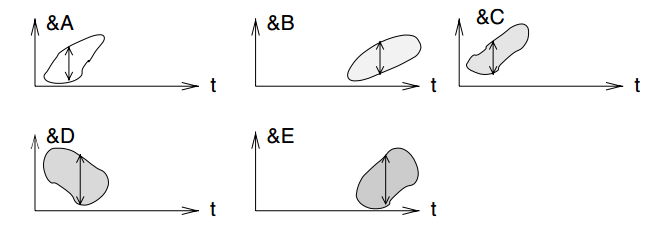
Với thuật toán ước tính chuyển động, số lượng vòng lặp có thể được giảm lên đến 75% (bằng 25% của giá trị ban đầu). Rõ ràng, việc tiết kiệm đáng kể là có thể. Tiềm năng này chắc chắn không nên bị bỏ qua.



Hình 5.11. Kết quả của việc chia tách vòng lặp

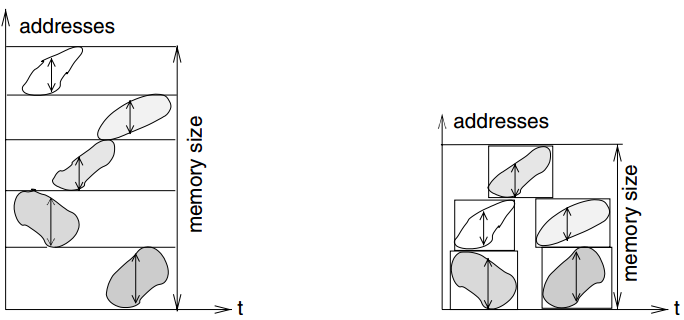
**5.2.5 Nén mảng – Array folding**

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.



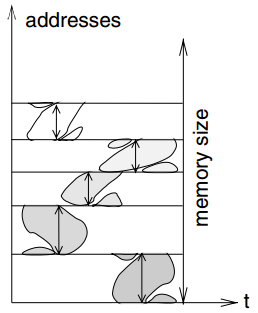
Hình 5.12. Các mô hình tham chiếu cho các mảng

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.

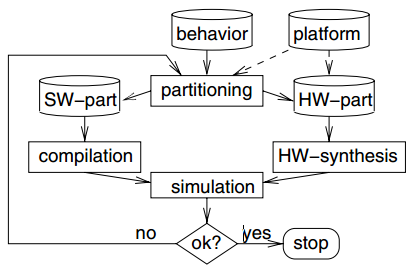
**5.3 Phân vùng phần cứng/phần mềm**

**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**

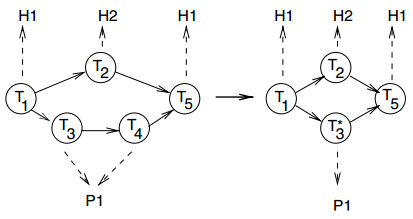
Đối với COOL, đầu vào bao gồm ba phần:

* **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.
* **Hành vi:** Phần thứ ba của đầu vào mô tả hành vi tổng thể cần thiết. Các đồ thị tác vụ phân cấp được sử dụng cho điều này. Chúng ta có thể nghĩ tới, ví dụ sử dụng đồ thị phân cấp của hình 2.46 cho điều này.

COOL sử dụng hai loại cạnh: cạnh giao tiếp và cạnh thời gian. Các cạnh giao tiếp có thể bao gồm các thông tin về khối lượng thông tin được trao đổi. Cạnh thời gian cung cấp các ràng buộc về thời gian. COOL yêu cầu được biết hành vi của mỗi nút lá – leaf node2 của đồ thị phân cấp. COOL mong đợi hành vi này được đặc tả trong VHDL3.

Với phân vùng, COOL sử dụng các bước sau:

1. **Chuyển đổ**i các hành vi **vào một mô hình đồ thị bên trong**.
2. **Chuyển đổi** các hành vi của mỗi nút lá **từ VHDL vào C**.
3. **Biên dịch tất cả các chương trình C** cho bộ vi xử lý mục tiêu được lựa chọn, các tính toán về kích thước chương trình kết quả, ước lượng thời gian xử lý kết quả. Nếu mô phỏng được sử dụng cho sau này, các dữ liệu mô phỏng phải có sẵn.
4. **Đồng bộ các thành phần phần cứng:** Với mỗi nút lá, phần cứng dành riêng cho ứng dụng được đồng bộ. Khi mà một số lượng tương đối các thành phần phần cứng có thể phải được đồng bộ, đồng bộ phần cứng không nên quá chậm. Có thể thấy rằng các công cụ đồng bộ thương mại tập trung vào đồng bộ ở mức cổng có thể quá chậm để có ích cho COOL. Tuy nhiên, các công cụ đồng bộ mức cao làm việc ở mức giao tiếp thanh ghi (sử dụng các thành phần như là địa chỉ, các thanh ghi, và bộ dồn kênh, chứ không phải là các cổng) cung cấp tốc độ đồng bộ vừa đủ. Các công cụ như vậy cũng có thể cung cấp các giá trị chính xác vừa đủ trong thời gian trễ và diện tích silicon cần thiết. Trong việc thực hiện thực tế, công cụ đồng bộ mức cao OSCAR [Landwehr và Marwedel, 1997] được sử dụng.
5. **Làm phẳng hệ thống phân cấp:** Bước tiếp theo là trích xuất ra một đồ thị tác vụ phẳng từ đồ thị luồng phân cấp. Khi không có việc sáp nhập hoặc chia tách các nút được thực hiện, thông tin cốt lõi được sử dụng bởi các nhà thiết kế được duy trì. Chi phí và thông tin hiệu suất đạt được từ việc biên dịch và từ đồng bộ phần cứng được bổ sung tới các nút. Điều này là một thực tế trong các ý tưởng cốt lỗi của COOL: **thông tin cần thiết cho phân vùng phần cứng/phần mềm được tính toán trước và nó được tính toán với độ chính xác tốt**. Thông tin này là cơ sở cho việc tạo ra các thiết kế với chi phí tối thiểu phù hợp với các ràng buộc.
6. **Tạo ra và giải quyết một mô hình toán học của vấn đề tối ưu hóa:** COOL sử dụng lập trình số nguyên (IP) để giải quyết vấn đề tối ưu hóa. Môt giải pháp IP thương mại được sử dụng để tìm ra các giá trị cho các biến quyết định giảm thiểu chi phí. Giải pháp là tối ưu đối với hàm chi phí từ các thông tin sẵn có. Tuy nhiên, chi phí này chỉ bao gồm một xấp xỉ thô của thời gian giao tiếp. Thời gian giao tiếp giữa bất cứ hai nút của đồ thị tác vụ phụ thuộc vào việc ánh xạ các nút đó tới bộ vi xử lý và phần cứng. Nếu các hai nút được ánh xạ tới cùng bộ vi xử lý, giao tiếp sẽ là cục bộ và vì vậy rất nhanh. Nuế các nút được ánh xạ tới các thành phần phần cứng khác nhau, giao tiếp sẽ không phải là cục bộ và sẽ chậm hơn. Mô hình hóa chi phí giao tiếp cho tất cả các ánh xạ có thể của các nút đồ thị tác vụ sẽ tạo ra mô hình rất phức tạp và do đó được thay thế những cải tiến lặp của giải pháp ban đầu. Thêm chi tiết về bước này sẽ được trình bày dưới đây.
7. **Các cải tiến lặp:** Để làm việc với ước tính tốt của thời gian giao tiếp, các nút liền kề mà ánh xạ tới cùng thành phần phần cứng bây giờ sẽ được sáp nhập. Việc sáp nhập này được chỉ ra trong hình 5.16.

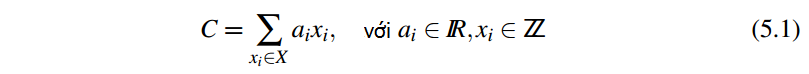


Hình 5.16 Sáp nhập các nút tác vụ mà ánh xạ tới cùng các thành phần phần cứng

Chúng ta giả định rằng tác vụ T1, T2 và T5 được ánh xạ tới các thành phần phần cứng H1 và H2, nơi mà T3 và T4 được ánh xạ tới bộ vi xử lý P1. Theo đó, giao tiếp giữa T3 và T4 là giao tiếp cục bộ. Vì vậy, chúng ta sáp nhập T3 và T4­ và giả định rằng giao tiếp giữa hai tác vụ không yêu cầu kênh giao tiếp. Thời gian giao tiếp bây giờ có thể được ước lượng với độ chính xác cải tiến. Đồ thị kết quả sau đó được sử dụng như đầu vào mới cho việc tối ưu hóa toán học. Bước trước và bước hiện tại là được lặp lại cho đến khi không còn các nút đồ thị nào được sáp nhập thêm nữa.

1. **Đồng bộ giao diện:** Sau phân vùng, tính logic kết dính trong giao tiếp bộ vi xử lý, phần cứng dành riêng cho ứng dụng và các bộ nhớ được tạo ra.

Tiếp theo, chúng ta sẽ mô tả bước 6 chi tiết hơn. Các mô hình IP cung cấp một phương pháp chung cho việc mô hình các vấn đề tối ưu hóa. Các mô hình IP bao gồm hai phần: một hàm chi phí và một tập các ràng buộc. Cả hai phần tài liệu tham khảo liên quan một tập của các biến số nguyên có giá trị. Các hàm chi phí phải là các hàm tuyến tính của các biến đó. Vì vậy, chúng phải có dạng chung



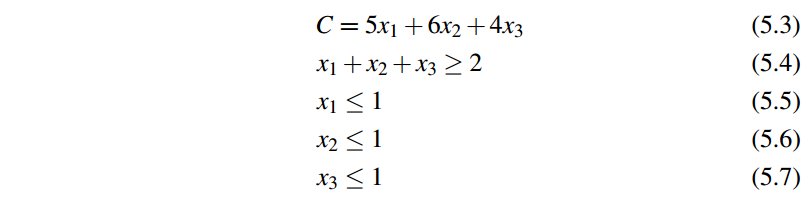
Tập *J* của các ràng buộc cũng phải bao gồm các hàm tuyến tính của các biến số nguyên có giá trị. Chúng phải có dạng



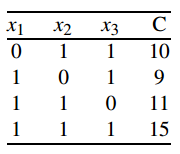
Chú ý rằng ≥ có thể được thay thế bởi ≤ trong biểu thức (5.2) nếu ràng buộc *bi,j* được thay đổi theo.

**Đinh nghĩa:** **Vấn đề lập trình số nguyên (IP-)** là vấn đề tối thiểu hàm chi phí (5.1) trong các ràng buộc được đưa ra trong biểu thức 5.2. Nếu tất cả các biến bao gồm 0 hoặc 1, mô hình tương ứng được gọi là một **mô hình lập trình số nguyên 0/1 (0/1-IP)**. Trong trường hợp này, các biến cũng được xác định như các **biến quyết định (nhị phân)**.

Ví dụ, giả sử rằng x1, x2 và x3 là các số nguyên không âm, tập các biểu thức sau thể hiện một mô hình 0/1-IP:



Do các ràng buộc, tất cả các biến là 0 hoặc 1. Có bốn giải pháp có thể. Chúng được liệt kê trong bảng 5.1. Giải pháp với chi phí bằng 9 là tối ưu.



Bảng 5.1. Các giải pháp có thể của vấn đề IP đã trình bày

Các ứng dụng đòi hỏi **tối đa** một số hàm **độ lợi** *C’* có thể chuyển đổi thành dạng bên trên bằng cách đặt.

Các mô hình IP có thể được giải quyết tối ưu sử dụng các kĩ thuật lập trình toán học. Không may, lập trình số nguyên là NP-đầy đủ và số lần xử lý có thể trở lên rất lớn. Tuy nhiên, nó là rất hữu ích cho việc giải quyết các vấn đề tối ưu miễn là các kích thước mô hình không quá lớn. Số lần xử lý phụ thuộc vào số lượng các biến và số lượng và cấu trúc của các ràng buộc. Trình giải quyết IP tốt (như Ip-solve [Berkelaar và et al., 2005] hoặc CPLEX) có thể giải quyết các vấn đề được cấu trúc tốt gồm vài ngàn biến trong thời gian tính toán chấp nhận được (ví dụ vài phút). Thêm các thông tin về lập trình số nguyên và **lập trình tuyến tính** có liên quan, tham khảo các cuốn sách về chủ đề (ví dụ Wolsey [Wolsey, 1998]). Mô hình các vấn đề tối ưu hóa như các vấn đề lập trình số nguyên rất có ý nghĩa bất chấp độ phức tạp của vấn đề: nhiều vấn đề có thể được giải quyết trong thời gian xử lý chấp nhận được và nếu chúng không thể, các mô hình IP cung cấp một điểm bắt đầu tốt cho các phỏng đoán.

Tiếp theo, chúng ta sẽ mô tả xem làm thế nào phân vùng có thể được mô hình hóa sử dụng mô hình 0/1-IP:

* Tập chỉ số *I* xác định các nút đồ thị tác vụ. Mỗi tương ứng với một nút đồ thị tác vụ.
* Tập chỉ số *L* xác định các **loại** nút đồ thị tác vụ. Mỗi tương ứng với một loại nút đồ thị tác vụ. Ví dụ, có thể có các nút mô tả các tính toán căn bậc hai, chuyển đổi Cosi rời rạc (DCT) hoặc chuyển đổi Fourier nhanh rời rạc (DFT). Mỗi trong số chúng là được tính như một loại.
* Tập chỉ số *KH* xác định các **loại** thành phần phần cứng. Mỗi tương ứng với một loại thành phần phần cứng. Ví dụ, có thể có các thành phần phần cứng đặc biệt cho DCT hoặc DFT. Trong đó một giá trị chỉ số cho thành phần phần cứng DCT và một cho thành phần phần cứng DFT.
* Mỗi thành phần phần cứng, có thể là nhiều bản sao hoặc thể hiện - “instances”. Mỗi thể hiện được xác định bởi một chỉ số.
* Tập chỉ số *KP* xác định các bộ vi xử lý. Mỗi xác định một trong các bộ vi xử lý (tất cả chúng là cùng loại).

Sau đây là các biến quyết định cần thiết cho mô hình:

* : Biến này sẽ là 1, nếu nút được ánh xạ tới loại thành phần phần cứng và ngược lại là 0.
* : Biến này sẽ là 1, nếu nút được ánh xạ tới bộ vi xử lý và ngược lại là 0.
* : Biến này sẽ là 1, nếu với ít nhất một nút loại *l* được ánh xạ tới bộ vi xử lý và ngược lại là 0.
* *T* là một ánh xạ từ các nút đồ thị tác vụ tới các loại tương ứng của chúng.

Trong các trường hợp riêng của chúng ta, hàm chi phí tích lũy tổng số chi phí của tất cả các đơn vị phần cứng:

*C* = chi phí bộ vi xử lý + chi phí bộ nhớ + chi phí phần cứng dàng riêng cho ứng dụng

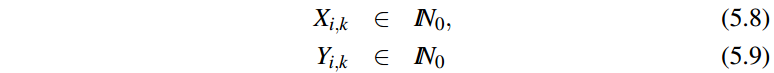
Chúng ta rõ ràng sẽ giảm thiểu tổng chi phí nếu không có các bộ vi xử lý, bộ nhớ và phần cứng dành riêng cho ứng dụng bao gồm trong thiết kế. Do các ràng buộc, điều này không phải là một giải pháp hợp lý. Chúng ta bây giờ có thể thể hiện một mô tả ngắn gọn về một vài ràng buộc của mô hình IP:

* **Các ràng buộc gán phép tính:** các ràng buộc này đảm bảo rằng mỗi phép tính được thực hiện trong phần cứng hoặc phần mềm. Các ràng buộc tương ứng có thể được công thức như sau:

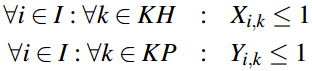
****

Một cách thuần túy, điều này có nghĩa như sau: trong tất cả các nút đồ thị tác vụ *I*, cái sau phải được xử lý: *i* được thực hiện trong phần cứng (đặt một trong các biến tới 1, với một vài *k*).

Tất cả các biến được giả định là các số nguyên không âm:



Các ràng buộc bổ sung đảm bảo rằng các biến quyết định và có 1 như một biên trên, do đó, trong thực tế là các biến 0/1 có giá trị:



Nếu chức năng của một nút nào đó của loại l được ánh xạ tới một vài bộ vi xử lý k, sau đó bộ nhớ lệnh của bộ vi xử lý này phải bao gồm một bản sao của phần mềm cho chức năng này:

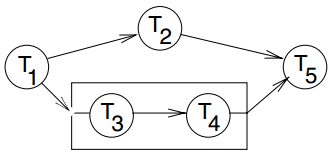


Một cách thuần túy, điều này có nghĩa là: với tất cả các loại l của các nút đồ thị tác vụ và với tất cả các nút i của loại này, cái sau phải được xử lý: nếu i được ánh xạ tới một vài bộ vi xử lý k (đã chỉ ra với đang là 1), sau đó phần mềm tương ứng với chức năng l phải được cung cấp bởi bộ vi xử lý k, và phần mềm tương ứng phải tồn tại trên bộ vi xử lý đó (đã chỉ ra với đang là 1).

Các ràng buộc bổ sung đảm bảo rằng các biến quyết định cũng là các biến 0/1 có giá trị:



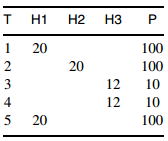
* **Các ràng buộc tài nguyên:** Tập hợp các ràng buộc tiếp theo đảm bảo rằng “không quá nhiều” các nút được ánh xạ tới cùng thành phần phần cứng ở cùng thời gian. Chúng ta giả định rằng, với mỗi chu kỳ đồng hồ, hầu như một phép tính có thể được thực hiện trên mỗi thành phần phần cứng. Thật không may, điều này có nghĩa rằng thuật toán phần vùng cũng phải tạo ra một lịch trình để xử lý các nút đồ thị tác vụ. Bản thân việc lập lịch trình cũng là một vấn đề NP-đầy đủ trong hầu hết các vấn đề có liên quan.
* **Các ràng buộc độ ưu tiên:** Các ràng buộc này đảm bảo rằng lịch trình cho việc xử lý các thao tác là phù hợp với các ràng buộc độ ưu tiên trong đồ thị tác vụ.
* **Các ràng buộc thiết kế:** Các ràng buộc này đặt một giới hạn trên chi phí của các thành phần phần cứng nhất định, như các bộ nhớ, các bộ vi xử lý hoặc diện tích phần cứng dành riêng cho ứng dụng.
* **Các ràng buộc thời gian:** Các ràng buộc thời gian, nếu có trong đầu vào COOL, được chuyển đổi thành các ràng buộc IP.
* Một vài các ràng buộc bổ sung, nhưng ít quan trọng không được liệt kê trong danh sách này.



Hình 5.17. Đồ thị tác vụ

Ví dụ: Trong phần tiếp theo, chúng ta sẽ thấy các ràng buộc này có thể được tạo ra cho đồ thị tác vụ trong hình 5.17 như thế nào (giống như trong hình 2.46).

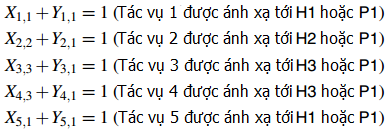
Giả sử rằng chúng ta có một thư viện thành phần phần cứng bao gồm ba loại thành phần H1, H2, và H3 với chi phí lần lượt là 20, 25, và 30 đơn vị chi phí. Hơn nữa, giả sử rằng chúng ta cũng có thể sử dụng bộ vi xử lý P với chi phí 5. Thêm vào đó, chúng ta giả định rằng bảng 5.2 mô tả số lần xử lý của các tác vụ của chúng ta trên các thành phần này.



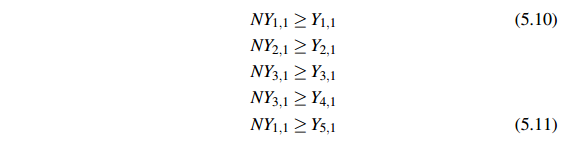
Bảng 5.2. Số lần xử lý của các tác vụ từ T1 tới T5 trên các thành phần

Các tác vụ từ T1 tới T5 có thể chỉ được xử lý trên bộ vi xử lý hoặc trên một đơn vị phần cứng dành riêng cho ứng dụng. Chính xác, các bộ vi xử lý được giả định là giá rẻ nhưng chậm trong việc xử lý các tác vụ T1, T2 tới T5.

Các ràng buộc gán phép tính sau phải được tạo ra, giả sử rằng tối đa một bộ vi xử lý (P1) được sử dụng:



Hơn nữa, giả định rằng các loại tác vụ T1 tới T5 lần lượt là *l* = 1, 2, 3, 3 và 1. Sau đó, các ràng buộc tài nguyên bổ sung sau đây được yêu cầu:



Biểu thức 5.10 có nghĩa: nếu tác vụ 1 được ánh xạ tới bộ vi xử lý, sau đó hàm *l* = 1 phải được thực hiện trên bộ vi xử lý đó. Hàm tương tự cũng phải được thực hiện trên bộ vi xử lý nếu tác vụ 5 được ánh xạ tới bộ vi xử lý (biểu thức 5.11).

Chúng ta đã không bao gồm các ràng buộc thời gian. Tuy nhiên, rõ ràng rằng bộ vi xử lý là chậm trong việc xử lý một vài tác vụ và phần cứng dành riêng cho ứng dụng yêu cầu với các ràng buộc thời gian dưới 100 đơn vị thời gian.

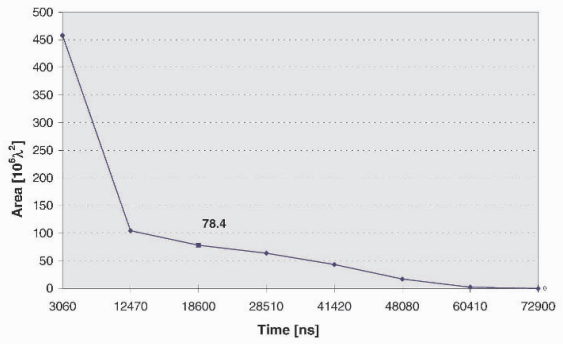
Hàm chi phí là:



với #() xác định số lượng các thể hiện của các thành phần phần cứng. Con số này có thể được tính toán từ các biến từng được giới thiệu nếu lịch trình cũng được đưa vào tài khoản. Với ràng buộc thời gian của 100 đơn vị thời gian, thiết kế chi phí tối thiểu bao gồm các thành phần H1, H2, và P. Điều này có nghĩa rằng các tác vụ T3 tới T4 được thực hiện trong phần mềm còn tất cả tác vụ khác trong phần cứng.

Nhìn chung, do độ phức tạp của vấn đề kết hợp việc phân vùng và lập lịch trình, chỉ số nhỏ các thể hiện vấn đề của vấn đề kết hợp có thể được giải quyết trong thời gian chấp nhận được. Do đó, vấn đề được chia tách một cách phỏng đoán vào vấn đề lập lịch trình và phân vùng: một phân vùng ban đầu dựa trên việc ước tính số lần xử lý và lịch trình cuối cùng được chạy sau việc phân vùng. Nếu nó chỉ ra rằng lịch trình đã quá khả quan, toàn bộ tiến trình phải được lặp lại với ràng buộc thời gian chặt chẽ hơn. Thực nghiệm với các ví dụ nhỏ đã chỉ ra rằng chi phí cho các giải pháp phỏng đoán chỉ lớn hơn 1 hoặc 2% chi phí của kết quả tối ưu.

Việc phân vùng tự động có thể được sử dụng cho việc phân tích không gian thiết kế. Trong phần tiếp theo, chúng ta sẽ trình bày các kết quả của một phòng thí nghiệm âm thanh – audio lab, bao gồm bộ trộn âm – mixer, bộ chỉnh tiếng vang – echo, bộ chỉnh âm lượng – fader, bộ chỉnh tần – equalizer, và các khối cân bằng – balance units. Ví dụ này sử dụng các công nghệ mục tiêu sớm hơn để chứng minh hiệu quả của việc phân vùng. Phần cứng mục tiêu bao gồm một bộ vi xử lý SPARC (chậm), bộ nhớ mở rộng, và phần cứng dành riêng cho ứng dụng được thiết kế từ một thư việc ASIC 1*µ* (đã lỗi thời). Tổng độ trễ cho phép được đặt tới 22675 ns, tương ứng với một tỉ lệ lấy mẫu 44.1 kHz, được sử dụng trong CDs. Hình 5.18 chỉ ra các quan điểm thiết kế khác nhau có thể được tạo ra bởi việc thay đổi ràng buộc độ trễ.



Hình 5.18. Không gian thiết kế trong audio lab

Đơn vị λ đề cập tới một đơn vị độ dài công nghệ phụ thuộc. Nó thực chất là một nửa khoảng cách gần nhất giữa các trung tâm của hai dây kim loại trên trên chip (cũng được gọi là *haft-pitch* [SEMATECH, 2003]). Quan điểm thiết kế ở bên trái tương ứng với một giải pháp được thực hiện hoàn toàn trong phần cứng, quan điểm thiết kế bên phải tương ứng với một giải pháp phần mềm. Các quan điểm thiết kế khác sử dụng một sự hỗn hợp phần cứng và phần mềm. Cái tương ứng với diện tích 78.4 λ2 là rẻ nhất đáp ứng thời hạn.

Rõ ràng, ngày nay công nghệ có ích lợi để cho phép một audio lab dựa trên 100% phần mềm. Tuy nhiên, ví dụ này cho thấy các phương pháp thiết kế cơ bản cũng có thể được sử dụng cho các ứng dụng đòi hỏi cao hơn, đặc biệt trong lĩnh vực đa phương tiện tốc độ cao, như MPEG-4.

**5.4 Các trình biên dịch cho các hệ thống nhúng**

**5.4.1 Giới thiệu**

Rõ ràng, các trình biên dịch và tối ưu có sẵn cho các bộ vi xử lý sử dụng trong các máy tính và việc tạo trình biên dịch được biết đến thường được sử dụng cho các bộ vi xử lý 32-bit. Với các hệ thống nhúng, các trình biên dịch chuẩn cũng được sử dụng trong nhiều trường hợp, khi chúng thường là rẻ hoặc thậm chí cung cấp miễn phí.

Tuy nhiên, có một vài lí do cho việc thiết kế các trình biên dịch và tối đặc biệt cho các hệ thống nhúng:

* Kiến trúc bộ vi xử lý trong các hệ thống nhúng đưa ra các tính năng đặc biệt (xem trang 100). Các tính năng này cần được khai thác bởi trình biên dịch để sinh ra mã hiệu quả.
* Các mức độ cao của tối ưu hóa quan trọng hơn tốc độ biên dịch cao.
* Các trình biên dịch có thể hỗ trợ tiềm năng để phù hợp và đáp ứng được các ràng buộc trong thời gian thực. Ví dụ, có thể có ích để đóng băng các dòng bộ nhớ đệm nhất định để ngăn chặn mã được thực thi thường xuyên từ việc loại bỏ và nạp lại nhiều lần.
* Các trình biên dịch có thể giúp giảm tiêu tốn năng lượng của các hệ thống nhúng. Các trình biên dịch thực hiện các tối ưu hóa năng lượng cần có sẵn.
* Với các hệ thống nhúng, có một số lượng rất lớn các tập lệnh. Do đó, có nhiều bộ vi xử lý mà các trình biên dịch nên có sẵn. Đôi khi thậm chí còn yêu cầu đáp ứng việc tối ưu các tập lệnh với trình biên dịch có khả năng định lại mục tiêu - **retargetable**. Với các trình biên dịch như vậy, tập lệnh có thể được xác dịnh như một đầu vào của một hệ thống tạo trình biên dịch. Các hệ thống như vậy có thể được sử dụng cho việc thử nghiệm thay đổi các tập lệnh và sau đó quan sát kết quả thay đổi trong mã máy được tạo ra. Điều này là một trường hợp riêng của thăm dò không gian thiết kế - **design space exploration** và được hỗ trợ, ví dụ, bởi các công cụ Tensilica [Tensilica Inc., 2003].

Một vài phương pháp đầu tiên cho các trình biên dịch có khả năng định lại mục tiêu được mô tả trong cuốn sách đầu tiên của chủ đề này [Marwedel và Goossens, 1995]. Các tối ưu có thể được tìm thấy trong nhiều cuốn sách gần đây của Leupers [Leupers, 1997], [Leupers, 2000a]. Trong phần này, chúng ta sẽ trình bày các ví dụ về kĩ thuật biên dịch cho các bộ vi xử lý nhúng.

Các kĩ thuật biên dịch cũng có thể phải hỗ trợ các kĩ thuật nén được mô tả trên trang 103 và 105.

**5.4.2 Biên dịch tiết kiệm năng lượng**

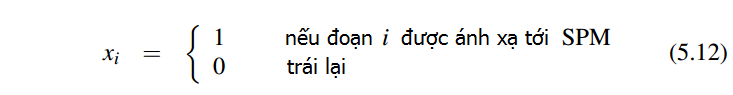
Nhiều hệ thống nhúng là các hệ thống di động phải chạy trên pin. Trong khi các nhu cầu tính toán trên hệ thống di động ngày càng tăng, công nghệ pin chỉ được cải thiện một cách chậm chạp [SEMATECH, 2003]. Do đó, khả năng đáp ứng của năng lượng là một nút cổ chai nghiêm trọng cho các hệ thống mới.

Tiết kiếm năng lượng có thể được thực hiện ở nhiều cấp độ, bao gồm cả quy trình công nghệ chế tạo, công nghệ thiết bị, thiết kế mạch, hệ điều hành và các thuật toán ứng dụng. Dịch đầy đủ từ các thuật toán tới mã máy cũng có thể giúp ích. Các kĩ thuật tối ưu mức cao như được thể hiện trên trang 157 tới 167 cũng có thể giúp giảm tiêu tốn năng lượng. Trong phần này, chúng ta sẽ xem xét các tối ưu trình biên dịch có thể giảm tiêu tốn năng lượng. Các mô hình năng lượng – **Power models** là các thành phần rất cần thiết của tất cả các tối ưu năng lượng. Một vấn đề chung của các mô hình năng lượng là độ chính xác thường rất hạn chế của chúng4.

* Một trong các mô hình năng lượng đầu tiên được đề xuất bời Tiwari [Tiwari et al., 1994]. Mô hình bao gồm cái gọi là các chi phí cơ sở và các chi phí liên lệnh. Các chi phí cơ sở của một lệnh tương ứng với năng lượng được tiêu tốn trên mỗi thực thi lệnh nếu một chuỗi vô hạn của lệnh đó được thực thi. Mô hình các chi phí liên lệnh tiêu tốn thêm năng lượng bởi bộ vi xử lý nếu các lệnh thay đổi. Năng lượng thêm này là cần thiết, ví dụ, do việc chuyển đổi các khối chức năng bật và tắt. Mô hình năng lượng này tập trung vào sự tiêu tốn trong bộ vi xử lý và không xem xét tới năng lượng đã tiêu tốn trong bộ nhớ hoặc trong các phần khác của hệ thống.
* Một mô hình năng lượng khác được đề xuất bởi Simunic et al. [Simunic et al., 1999]. Mô hình đó được dựa trên các bảng dữ liệu. Ưu điểm của phương pháp này này đó là sự đóng góp về sự tiêu tốn năng lượng của tất cả các thành phần của một hệ thống nhúng có thể được tính toán. Tuy nhiên, thông tin trong các bảng dữ liệu về các giá trị trung bình có thể ít chính xác hơn thông tin về các giá trị tối thiểu và tối đa.
* Một mô hình thứ ba đã được đề xuất bởi Rusell và Jacome [Russel và Jacome, 1998]. Mô hình này được dựa trên các phép đo chính xác của hai cấu hình cố định.
* Mô hình khác còn lại đã được đề xuất bởi Lee [Lee et al., 2001]. Mô hình này bao gồm một phần tích chi tiết về hiệu quả của đường ống dẫn – pipeline. Nó không bao gồm nhiều thao tác và các gian pipeline.
* Trình biên dịch tiết kiệm năng lượng encc từ Đại học Dortmund sử dụng mô hình năng lượng của Steinke et al. [Steinke et al., 2001]. Nó dựa trên các phép đo chính xác sử dụng phần cứng thực. Tiêu tốn của bộ vi xử lý cũng như của bộ nhớ được bao gồm.
* Tiêu tốn năng lượng của các bộ nhớ đệm có thể được tính toán với CACTI [Wilton và Jouppi, 1996].

Sử dụng các mô hình giống như một cái bên trên, các tối ưu trình hóa biên dịch sau đây đã được sử dụng cho việc giảm thiểu tiêu tốn năng lương:

* **Lập lịch trình tiết kiệm năng lượng:** thứ tự của các lệnh có thể được thay đổi miễn là ý nghĩa của chương trình không thay đổi. Thứ tự có thể được thay đổi để sao cho số lượng các lưu chuyển trên bus lệnh là tối thiểu. Việc tối ưu hóa này có thể được thực hiện trên đầu ra được tạo ra bởi một trình biên dịch và do đó không yêu cầu bất cứ thay đổi nào với trình biên dịch.
* **Lựa chọn lệnh tiết kiệm năng lượng:** thông thường có các chuỗi lệnh khác nhau cho việc thực hiện cùng một mã nguồn. Trong trình biên dịch chuẩn, số các lệnh hoặc số chu kỳ được sử dụng như một điều kiện (hàm chi phí) cho việc lựa chọn một chuỗi lệnh tốt. Điều kiện này có thể được thay thế bới năng lượng đã tiêu tốn bởi chuỗi lệnh đó. Steinke và những người khác đã chỉ ra rằng việc lựa chọn lệnh năng lượng thấp giảm việc tiêu tốn năng lượng vài phần trăm.
* **Thay thế hàm chi phí** cũng có thể trong các tối ưu trình biên dịch chuẩn, như thanh ghi đường ống, chuyển dịch mã bất biến vòng lặp – loop invariant code motion vv. Các cải tiến cũng ở trong vài phần trăm.
* **Khai thác hệ thống phân cấp bộ nhớ:** Như đã giải thích trên trang 118, các bộ nhớ nhỏ hơn cung cấp truy cập nhanh hơn và tiêu tốn ít năng lượng mỗi lần xử lý hơn. Vì vậy, một khoảng năng lượng đáng kể của thể được tiết kiệm nếu sự tồn tại của các bộ nhớ tạm (SPMs) có thể được khai thác bởi trình biên dịch. Với mục đích này, mỗi khối cơ bản và mỗi biến có thể được mô hình như một phần đoạn bộ nhớ *i*. Với mỗi phần đoạn, có một kích thước tương ứng *si*. Sử dụng mô tả sơ lược, nó có thể tính toán độ lợi *gi* của việc di chuyển đoạn *i* tới bộ nhớ tạm. Cho



Sau đó, mục tiêu là để tối đa

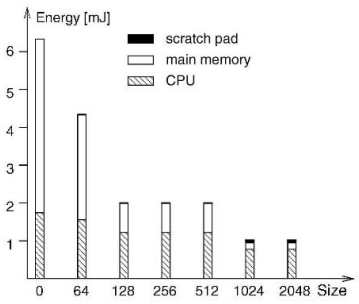


trong khi thỏa mãn ràng buộc về kích thước



với *K* là kích thước của SPM.

Vấn đề này được biết đến như một vấn đề ba lô – knapsack. Giải pháp của vấn đề này là một ánh xạ một tới một. Một mô hình lập trình số nguyên đưa đến một ánh xạ như đã được trình bày bởi Steinke et al. [Steinke et al., 2002b]. Đối với một số ứng dụng đo hiệu năng - benchmark, việc giảm thiểu năng lượng lên tới 80% đã được thấy, mặc dù kích thước của SPM chỉ là một phần nhỏ của tổng kích thước mã của ứng dụng. Các kết quả của chương trình sắp xếp nổi bọt được thể hiện trong hình 5.19.



Hình 5.19. Giảm thiểu năng lượng bởi việc ánh xạ dựa trên trình biên dịch tới bộ nhớ tạm cho sắp xếp nổi bọt

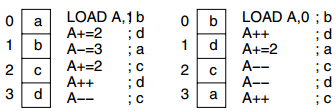
Rõ ràng, các SPM lớn hơn dẫn tới giảm thiểu tiêu tốn năng lượng trong bộ nhớ chính. Năng lượng cần thiết cho bộ vi xử lý cũng được giảm thiểu, khi ít chu kỳ chờ đời được yêu cầu. Điện áp cung cấp đã được giả định là không đổi.

Mã cũng có thể được sao chép động vào SPM, kết quả trong một ánh xạ nhiều tới một. Một mô hình lập trình số nguyên phản ánh vấn đề tối ưu hóa tổng quát hơn cũng đã được đề xuất bởi Steinke et al. [Steinke et al., 2002a]. Sử dụng mô hình tổng quát hơn này, độ lợi năng lượng có thể được gia tang, đặc biệt trong các ứng dụng mà SPM là quá nhỏ để chứa tất cả các điểm nóng – hot spots.

Của tất cả các trình tối ưu hóa trình biên dịch đã phân tích bởi Steinke, việc tiết kiệm năng lượng được kích hoạt bởi các hệ thống phân cấp bộ nhớ là lớn nhất.

**5.4.3 Biên dịch cho các bộ vi xử lý tín hiệu kĩ thuật số**

Các tính năng của các bộ vi xử lý DSP được mô tả trên trang 108. Các trình biên dịch cần khai thác điều này. Các kĩ thuật cho điều này có thể được minh họa bằng việc sử dụng các khối tạo địa chỉ như các ví dụ. Khả năng của việc tạo ra các địa chỉ “miễn phí” có một yếu tố quan trọng trong việc các biến cần được bố trí trong bộ nhớ như thế nào. Hình 5.20 cho thấy một ví dụ.

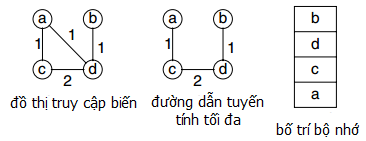


Hình 5.20. So sánh các bố trí bộ nhớ

Chúng ta giả định rằng trong các khối cơ bản, các biến **a** tới **d** được truy cập trong chuỗi **(b,d,a,c,d,c)**. Truy cập các biến này với việc đánh địa chỉ gián tiếp là cần thiết, lệnh tham chiếu tới biến **b** không được chỉ ra trong hình 5.20, khi tập trung hiện tại là trên việc tạo địa chỉ. Vì vậy, việc tạo địa chỉ cho việc truy cập tới biến kế tiếp (**d**) là được xem xét tiếp theo. Giả định rằng đó chỉ là một thanh ghi địa chỉ duy nhất **A**, **A** phải được cập nhật để trỏ tới biến **d**. Điều này yêu cầu việc thêm 2 tới thanh ghi. Một lần nữa, chúng ta bỏ qua các lệnh nạp biến, và chúng ta ngay lập tức xem xét việc truy cập tới **a**. Với điều này, chúng ta phải trừ đi 3, và cho truy cập tiếp theo chúng ta phải thêm 2. Giả định rằng phạm vi tăng và giảm tự động được giới hạn là ± 1, chỉ hai truy cập cuối chỉ ra trong hình 5.20 có thể được thực thi với các thao tác này. Tổng cộng, 4 lệnh cho việc tính toán các địa chỉ là cần thiết.

Ngược lại, với cách bố trí trong hình 5.20 (bên phải), 4 phép tính toán địa chỉ là các phép tính tăng và giảm tự động mà được thực thi trong mô hình song song với một số phép tính trong đường dẫn dữ liệu chính. Chỉ 2 chu kỳ cần thiết cho các tính toán địa chỉ với độ lệch lớn hơn 1. Một lần nữa, các lệnh thực sự sử dụng các biến không được chỉ ra.

Làm thế nào chúng ta tạo ra các bố trí bộ nhớ thông minh như vậy? Các thuật toán làm điều này thường bắt đầu từ một đồ thị truy cập (xem hình 5.21).



Hình 5.21. Cấp phát bộ nhớ cho truy cập chuỗi (b, d, a, c, d, c) cho một thanh ghi đơn A