**Chương 5**

**THỰC THI HỆ THỐNG NHÚNG:**

**ĐỒNG THIẾT KẾ PHẦN CỨNG/PHẦN MỀM**

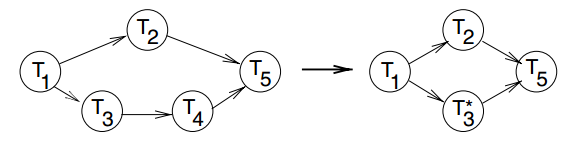
**Lựa chọn không gian thiết kế:** Trong hầu hết các trường hợp, có một vài các thiết kế đáp ứng các đặc tả. Lựa chọn không gian thiết kế là quá trình phân tích tập hợp các thiết kế có thể. Trong số những thiết kế đáp ứng các đặc tả, một thiết kế phải được lựa chọn.

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.

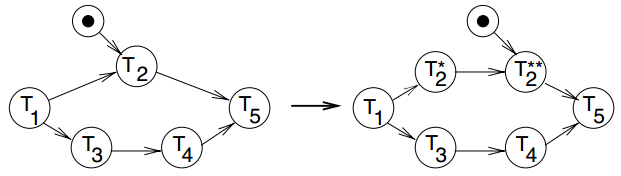
Sự sáp nhập các đồ thị tác vụ có thể được thực hiện bất cứ khi nào mà một số tác vụ Ti là tác vụ ngay trước của một số tác vụ Tj khác và Tj không có bất kỳ tác vụ nào khác ngay trước (xem hình. 5.3 với Ti = T3 and T­j = T4). Sự chuyển đổi này có thể đưa đến việc giảm chi phí trong chuyển ngữ cảnh nếu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

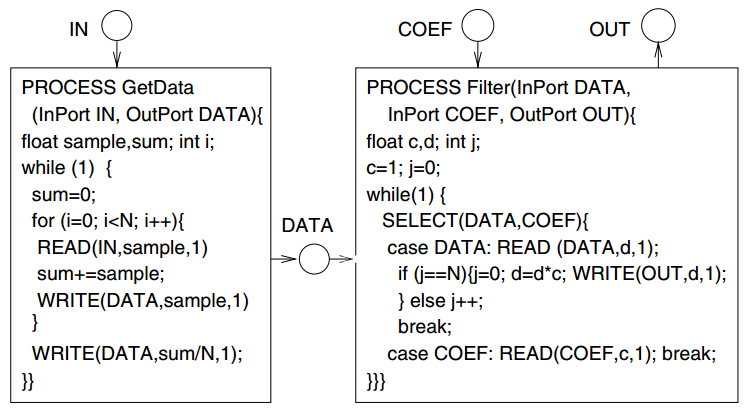
Các tác vụ có thể nắm giữ các tài nguyên (như lượng lớn bộ nhớ) trong khi chúng đang chờ cho một số đầu vào. Để sử dụng tối đa các tài nguyên, tốt nhất nên ràng buộc việc sử dụng các tài nguyên này trong khoảng thời gian mà các tài nguyên này thực sự được sử dụng. Trong hình 5.4, chúng ta giả định rằng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

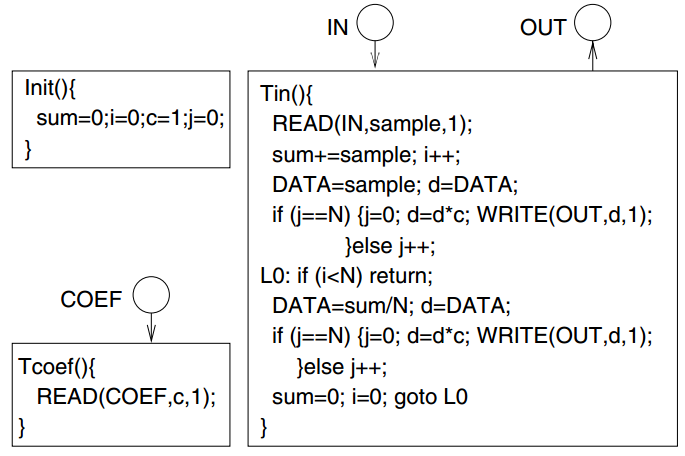
Người ta có thể cho rằng dù sao đi nữa thì các tác vụ nên giải phóng các tài nguyên như một lượng lớn bộ nhớ trước khi chờ đầu vào. Tuy nhiên, tính dễ đọc của các đặc tả ban đầu có thể liên quan đến việc thực thi các vấn đề trong giai đoạn thiết kế sớm.

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

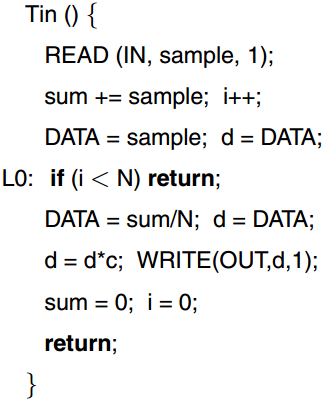
Ví dụ sử dụng các cổng đầu vào **IN** và **COEF**, cũng như các cổng đầu ra **OUT**. Giao tiếp liên tiến trình điểm tới điểm giữa các tiến trình được thực hiện thông qua một kênh đệm **DATA** đơn hướng. Tác vụ **GetData** đọc dữ liệu từ môi trường và gửi nó tới kênh **DATA**. Mỗi lần **N** mẫu được gửi, 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 lượng các chuyển đổi vòng lặp mà có thể được áp dụng với các đặc tả. Sau đây là một danh sách của các chuyển đổi vòng lặp tiêu chuẩn: