可重組式晶片系統之軟硬體共同設計

林尚威、黃駿賢、曾志毫、廖志峰、熊博安*

國立中正大學資訊工程學系 嵌入式系統實驗室

嘉義縣民雄鄉三興村160號
*E-mail: pahsiung@cs.ccu.edu.tw

摘要：

在一個晶片系統(System-on-a-chip, SoC)的設計流程中，包括了系統功能的需求分析與設計、軟硬體分割(Hardware/Software Partitioning)、排程(Scheduling)到最後的系統合成(System Synthesis)。而在一個可重組式晶片系統(Reconfigurable SoC)設計流程中，這些部份往往更加複雜。本文的重點在於提出了一個適用於設計可重組式晶片系統的設計流程，而在流程中的每個階段有很多方法可以使用，我們將許多文獻中所提到的方法作一個整理。

關鍵字：

可重組式晶片系統、軟硬體分割、排程、系統合成

壹、引言

利用積體電路來實現所需的演算法一般分為兩種方式。第一種是使用微處理器（microprocessor）執行以軟體方式所撰寫的演算法，其特性是彈性高效率差；第二種是使用特殊應用積體電路（ASIC），直接將演算法以硬體實現，其特性是彈性差效率高。利用可重組式運算（Reconfigurable Computing）的技術，我們可以同時獲得微處理器的彈性與特殊應用積體電路的效率[4]。

可重組式運算之系統架構通常包含微處理器，以及可重組式邏輯（Reconfigurable Logic），例如：現場可規劃閘陣列（FPGA）。微處理器主要負責程式中控制流程（Control Flow）的執行，而需要耗費大量微處理器的運算或關鍵性的輸出/入運算，則交由可重組式邏輯負責[2]。若是將可重組式運算系統實作在一個晶片系統（SoC）上，則稱為可重組式晶片系統（Reconfigurable SoC, RSoC）。藉由可重組式邏輯的輔助，我們可讓一個系統具備以硬體實作出多種不同演算法的能力，讓系統之效能與彈性獲得最大的發揮。

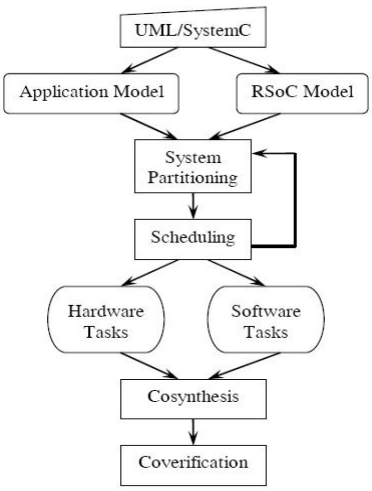
執行時期重新組態（Run-time Reconfiguration, RTR）[4]的技術，可解決可重組式邏輯資源有限，無法將應用程式的多個部份，都利用硬體來加速的問題。RTR 的概念與虛擬記憶體之概念相似，當應用程式中需利用硬體加速執行的部份過大，無法一次置入可重組式邏輯中時，就需先將欲改用硬體加速的部份進行切割（Partitioning），等到應用程式執行到那一部份時，再動態的替換所需的硬體加速部份到可重組式邏輯中。RTR 技術使得應用程式中的多個部份可以利用硬體來加速執行，整體效能可以獲得更大幅度的提昇。

FPGA 組態重新規劃所耗的時間，是可重組式運算系統的最大效能瓶頸所在。解決此一問題通常是利用分割（Partitioning）演算法與排程（Scheduling）演算法，減少重新規劃的次數。此外，也可利用快速規劃（Fast Configuration）的技術，將組態（Configuration）做預存（Prefetching）、壓縮（Compression）、重定位（Relocation）與快取（Caching）。

這篇論文接下來的內容安排如下：第貳節描述了我們提出的可重組式晶片系統之軟硬體共同設計方法論。第參節對統一塑模語言(Unified Modeling Language，UML)作一個簡單的介紹，因為我們用它來作為我們方法論的輸入。第肆節整理了許多文獻中所提出的分割(Partitioning)演算法。第伍節整理了許多文獻中所提出的排程(Scheduling)演算法。第陸節則整理了共同合成(Co-synthesis)的方法。第柒節是這篇論文的結論。第捌節是我們的未來展望。

貳、可重組式晶片系統之軟硬體共同設計方法論

在這節中，我們提出了一個適用於可重組式晶片系統之軟硬體共同設計方法論，其示意圖如圖一 所示。



圖一　可重組式晶片系統共同設計流程

這個流程以用 UML 和 SystemC 設計的系統功能為輸入。之後會有二個模型的產生，一個是應用程式模型(Application model)，用以描述系統的功能；另一個是可重組式晶片系統的模型(RSoC model)，用以描述可重組式晶片上的資源限制。接下來經過了系統分割(system partitioning)以及排程(scheduling)，我們得到了硬體工作(Hardware Task) 和軟體工作(Software Task)。然後是共同合成(Co-synthesis)，將軟體工作和硬體工作共同合成在一起。最後是共同驗證(Co-verification)的階段，用來驗證系統是否符合規格。接下的幾節會對分割(Partitioning)、排程(Scheduling)及共同合成(Co-synthesis)有較多的著墨。

參、統一塑模語言

統一塑模語言（Unified Modeling Language, UML）是一種標準化的軟體系統描述語言，其定義了一些圖形表示法與語意，可以運用在軟體系統的開發過程中，將所須的規程文件明確的訂定（Specifying），視覺化（Visualizing）、建構（Constructing）以及文件化（Documenting）[13]。 藉由 UML 視覺化的描述方式，使得複雜的軟體系統規格也可以用簡單明瞭的方式來呈現，如此一來我們可以減少系統規格訂定過程中可能發生的錯誤，簡化複雜的軟體系統設計流程。

UML 在軟體系統開發中的成功應用，使其逐漸也被廣泛的用於描述硬體系統。在描述硬體系統時，需要針對所需應用的領域對 UML 作擴充，彌補其描述硬體系統能力的不足。Real-Time UML 是針對及時系統設計所作的擴充。為了使 Real-Time UML 更加適於嵌入式系統的開發，並融入軟硬體共同設計技術，Real-Time UML 被更進一步的擴充為 Embedded UML [12]。各個應用領域對 UML 的擴充，促使 UML 新版本的制定。UML 目前的版本是第 1.5 版， UML 2.0 版的規格已經接近最終審查階段，UML 2.0 的制定，將使其可以被廣泛的應用於軟硬體系統與晶片系統的設計。

肆、軟硬體分割

在軟硬體分割的相關文獻中，我們可以找到很多種的分割演算法，而這些演算法大體上可以歸納為建構式(constructive)和反覆式(iterative)兩種。建構式演算法是一層一層的比較其解的優劣程度，再從各層中將最佳解抽出比較，找出一組問題的最佳解，這類演算法在時間複雜度上，比反覆式演算法來的快；相反地，反覆式演算法則是一層一層地尋找最佳解，即使可以找到比建構式演算法還要好的解，但其時間複雜度也比建構式演算法來的多。

在動態可重組式的系統中，有一個非常重要的議題，就是如何藉由分時的策略來使用相同可重組的硬體資源。而在這議題上，我們又不可忽略硬體重組所帶來之能量、記憶空間及時間的成本消耗。而且由於不同的分割方式及各個軟硬體節點間相依性的不同，會造成節點間不一樣的溝通成本支出。因此，在[6]、[5]中，都提出了針對這類成本消耗的解決方式。在 [6] 中，在對整個電路以網路流(Network-Flow)為基礎做出輸入模型，再根據時間區段的限制，利用ASAP(As Soon As Possible)及ALAP(As Late As Possible)的排程演算法將每個節點可以執行的時間區段計算出來，以及找出不須限定在某一區段執行的節點，再依次用α-Bounded Bipartitioning演算法將這些節點分至各個時間區段，最後再找出一組使用記憶體空間最少的最佳解。而[5] 也是以網路流(Network-Flow)為基礎，再利用溝通成本

及溝通權重等限制,反覆的用最小切集(min-cut)的演算法來求得最佳解。

[11] 是另一個反覆式的分割演算法，其輸入的模型是有向無迴圈圖(Direct Acyclic Graph)，演算法是兩層架構，在排程結束後，再用模擬退火為基礎的分割演算法來做軟硬體的分割。利用此軟硬體分割的方法論，可以在極短的時間內得到近似最佳解。

而在[3] 中是以資料流程圖(Data Flow Diagram)為輸入模型，而演算法是改進線性排程(List-Scheduling)的時間(temporal)分割演算法，此演算法利用區域最佳化，再依工作(Task)執行的先後順序來分割，而非利用其相依性，較適合使用在工作相依性低的電路中。使用光譜法(Spectral Method)的時間(temporal)分割演算法，可確保具有相依性的元件在分割前會被放置在相近的位置，以便在分割時可將這些元件分在同一個分割中，適合使用在工作相依性高的電路中。

伍、排程

若依排程的時間點來分，排程分為兩類：靜態排程(Static Scheduling)和動態排程(Dynamic Scheudling)。所謂的靜態排程是指各個工作在合成時就決定了什麼時間點該被執行；而動態排程則是指各個工作在執行時期才被決定執行的先後順序及時間點。以下分別整理出文獻中所提出的排程法。

一、 靜態排程

在作排程(靜態)時，將可重組式的平台環境 (Reconfigurable Environment)用一個模型(model)來表示是必要的。Reconfigurable System Design Model (RSDM) [14]就是這樣的一個模型，這個模型包括了資源限制 (Resource Constraints)、硬體元件庫 (Hardware Library)、工作系統 (Task System)、設計限制 (Design Constraints)、排程器 (Scheduler)、工作排程(Task Schedule)、高層次的硬體系統描述檔(High-level Hardware System Description)，如圖二所示。

資源限制描述了環境中資源(如Reconfigurable Logic、Embedded Memory)的最高數量。硬體元件庫包括了功能單元(Function Unit)及溝通元件元素(Communication Core Element)，其中功能單元又分為二類：處理器(Processor Core)和特定功能元件(Task Specific Core)。工作系統用有向無迴圈圖(Direct Acyclic Graph)表示系統中工作的關係，最後轉換為鄰接矩陣(Adjacency Matrix)。設計限制描述了系統設計完後應該要有的效能。排程器接受上述的輸入去作排程，排程後的輸出為工作排程及高層次的硬體系統描述檔。

排程器在作排程時會有四個步驟：合法順序產生器(Legal Sequence Generator)、排程產生程序(Process Generation Process)、溝通元件元素插入程序(CCE Insertion Process)、可靠度檢查(Realiability Check)。合法順序產生器確定了工作間的先後順序；排程產生程序將確定各個工作確實的執行時間點；溝通元件元素插入程序為功能單元及特定功能元件搭起橋梁；最後的可靠度檢查是為了確保這樣的排程有符合資源限制及設計限制。

二、動態排程

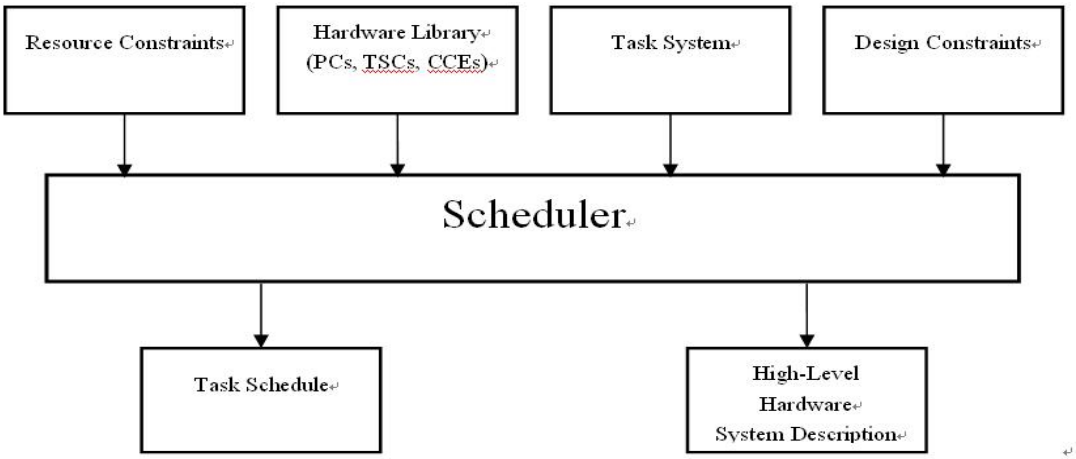
1. 可重組硬體架構的空間分配

大致上可將可重組式邏輯(Reconfigurable Logic)看成一個二維的空間，在使用的時候可分為三種型態：(a) X、Y兩軸任意長度，(b) 固定一軸的長度，另一軸任意長度，(c) X、Y軸皆為固定長度，也就是所有區塊大小相同。長度自由分配的好處是空間能夠更善加利用，但是會增加使用上的複雜度。相反的，固定空間大小比較容易配置，但可能會比較浪費空間。時間與空間的抉擇取決於使用者的需求。

2. 動態排程方法

大致上我們會將一個工作基本單位稱為task，這些工作(Task)也就是排程器的輸入，經由排程器再輸出適合的工作執行順序。

大多數的排程方法都是以串列排程(List Scheduling)為基礎且使用佇列(Queue)來暫儲預備執行的工作，每個工作都有屬於它的優先權，排程順序就依照優先權來決定。而優先權高低的裁定方法，就是整個動態排程器的核心。以下提出一些計算優先權的演算法：(1)First-In First-Out (FIFO) (2)Shortest Job First (SJF) (3)Earliest Deadline First (EDF) (4)As Soon As Possible (ASAP)。設計階段所求出的優先權，在動態重組的應用上大多不是適合的解，動態的優先權計算，才能符合現實的需求。例如在[1]中，他利用dyna_ASAP(Dynamic As Soon As Possible)、ALAP(As Late as Possible) 這兩個方法的結合



圖二　可重組式系統設計模型

來求出優先權的數值。dyna_ASAP表示最快什麼時候可以有空間使用，這在執行狀態才能決定，ALAP表示最晚什麼時候task必須被執行。當dyna_ASAP與ALAP數值越小優先權越高。公式如下：

Priority = - (dyna_ASAP + ALAP)

除了使用這些演算法決定優先權之外，還有其他的因素要考慮。在動態重組的應用上，重組是會耗費時間與空間的，因此一些用來減少重新組態(Reconfiguration)的方法，被增加到排程的過程中，已求得更少的時間與空間損耗。比起排定工作順序，重組對整體效能的影響更大。下一段介紹的就是工作配置的方法。

3.　空間配置(Placement)

3.1　減少重組次數的空間配置方法

一個應用會有許多的工作(Task)，每個工作都有自己的型態，在這裡型態可以看成它是在作什麼類型的運算。而這些工作部分會擁有相同的型態，也就是說這些工作會作類似的運算，也許只是輸入的部分不一樣。因此將一個區塊（Block）建構成符合這種型態的組態，就能夠讓多個同類型的工作在這個區塊上運作進而重複利用這個區塊。重複使用(Reuse)同一個硬體中的區塊就是減少動態重組次數的方法之一[9]、[10]。當原先在區塊中的工作執行完，接著短時間內這個工作又必須再執行一次，或是同樣型態的工作馬上要在這個區塊上運作，在再度利用這個區塊前的這段時間，盡量不要去重新組態這個區塊，以免改變成新型態後馬上又要改回原來的型態。也就是說在必須重新組態一個區塊時，盡可能挑最近較少用到的類型區塊作重組。這個方法可以明顯的看出重組次數的減少，時間跟能源的損耗也會因此而減少。

預先處理(Prefetch)[9]，可以減少重組區塊的時間。當重複使用的問題考過後，知道接下來的工作必須安排在哪個區塊，而這個區塊的型態並不適合這項工作，目前也已經結束使用，那我們可以在工作準備執行之前預先建構好這個區塊，只要工作一到執行的時間，馬上就可以在區塊上執行，減少重組過程的這段時間。

當一個工作在所有的區塊中找不到適合的型態來直接使用時，也許會有好幾個區塊目前都不會被重複使用，可以拿來做重組。這時要在這些區塊中挑出一個最適合的來作重組，比如說挑最少使用到的區塊，或是挑執行下一個相同類型的工作時間間隔最長的區塊。

3.2　區塊(Block)大小不一的情況

以上都是假設所有的區塊大小全部相同的情況下。在[7]中，他將整個FPGA分成大小不一的許多區塊，這種方法可以充分的利用空間，但是要再加上別人提出的重複使用、預先處理這些減少重組次數的方法，會增加許多工作配置的複雜度。

在工作配置的條件上，區塊最大的空間必須能夠容納最大的工作，當工作要選擇區塊的時候，必須從最小的區塊開始挑起，如果沒有適合的區塊可以使用再往更大的空間去尋找，盡量利用最適合的大小空間，讓整塊可重組式邏輯(Reconfigurable Logic)充分的使用。但是空間、時間、能源，不可能全部同時得到最好的安排，在選擇了最適合的空間大小時，也許有一些不同類型的工作也等著使用這個區塊，這個時候就看是要安排其餘等待的工作至較大的

空間，或是持續等待，不斷的重組這個區塊，浪費時間與能源。這之間的取捨，就看使用者的要求而定。

陸、共同合成

在軟硬體共同合成(Hardware/Software Co-synthesis)相關的文獻中，[8]提出了在可重組式晶片系統上作低電量消耗(Energy-Efficient)的軟硬共體同合成方法。其主要的概念在於將可重組式晶片系統模型(Reconfigurable SoC Model)上的各個元件，依據耗電量的多寡區分出不同的運作狀態(Operating State)。然後將應用程式模型(Application Model)，也就是一個個的工作(Task)，對可重組式晶片系統模型上的各個元件作排列組合形成一個有向無迴圈圖(Direct Acyclic Graph)。這個圖代表了各個工作在各個元件上執行所會消耗的電量。最後，在這個圖上用動態規劃演算法(Dynamic Programming)即可找出耗電量最低的工作執行情況，也就是什麼工作在什麼元件上執行組合起來的耗電量會最低。

柒、結論

在這篇論文中，我們提出了一個適用於設計可重組式晶片系統的方法論，在方法論中的各個階段，我們整理了許多文獻提出的演算法，而這些演算法可以套用在我們的方法論中，使得我們的方法論可以具有設計不同特性系統的彈性。

捌、未來展望

在我們提出的可重組式晶片系統之軟硬體共同設計方法論中，最後一個階段是共同驗證(coverification)，這個階段主要是將軟硬體合在一起之，驗證是否符合系統規格。這個部份是我們將來要實現的。

玖、參考文獻

1. B. Mei, P. Schaumont, and S. Vernalde (2000), A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems, *Proceedings of the 11th ProRISC Workshop on Circuits, Systems, and Signal Processing*.
2. T. Beierlein, D. Fröhlich, and B. Steinbach, (2003). UML-based codesign of reconfigurable architectures. *Proceedings of the Forum on Specification and Design Languages (FDL'03)*, pp. 285-296.
3. C. Bobda (2002), CoreMap: A rapid prototyping environment for distributed reconfigurable systems, *Proceedings of the 13th IEEE International Workshop on Rapid System Prototyping (RSP'02).*
4. K. Compton, and S. Hauck, (2002). Configurable computing: a survey of systems and software, *ACM Computing Surveys*, Vol. 34, No. 2. pp. 171-210.
5. D. N. Rakhmatov and S. B. K. Vrudhula (2002), Hardware-software bipartitioning for dynamically reconfigurable system, *Proceedings of the 10th International Workshop on Hardware-Software Codesign* (CODES), pp. 145-150, ACM Press.
6. H. Liu and D. F. Wong (1999), Circuit partitioning for dynamically reconfigurable FPGAs, *Proceedings of the ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays*, pp. 187-194, ACM Press.
7. H. Walder and M. Platzner (2003), Online scheduling for block-partitioned reconfigurable devices, *Proceedings of the Design Automation and Test, Europe* (DATE), pp. 290-295, ACM Press.
8. J. Ou, S. Choi, and V. K. Prasanna (2004), Energy Efficient Hardware/Software Co-Synthesis on Platform FPGAs, *International Journal on Embedded Systems*.
9. J. Resano, D. Mozos (2004), Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware, *Proceedings of the 41st Design Automation Conference*, pp. 119-124, ACM Press.
10. J. Noguera and R. M. Badia (2003), System-level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures, *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded System* (CASES), pp. 73-83, ACM Press.
11. K. S. Chatha and R. Vemuri (2000), An iterative algorithm for hardware-software

partitioning, hardware design space exploration and scheduling, *Design Automation for Embedded Systems*, pp. 281–293.

12. G. Martin, L. Lavagno, and J. Louis-Guerin, (2001) Embedded UML: a merger of real-time UML and co-design. *Proceedings of the 9th International Symposium on Hardware/Software Codesign*, pp. 23-28, Copenhagen, Denmark.
13. J. Rumbaugh, G. Booch, and I. Jacobson, (1999). *The UML Reference Guide*. Addison Wesley Longman.
14. S. M. Loo and B. E. Wells (2003), Task scheduling in a finite resource reconfigurable hardware/software co-design environment, *INFORMS Journal of Computing*.