**什么是多处理器片上系统（MPSOC）**

随着商业和大规模计算密集型科研应用对计算平台性能上越来越严格的要求，使得多处理器片上系统(MPSOC)的研究受到越来越广泛的关注。由于多处理器片上系统在单芯上集成了多个异构处理单元，使其可以完成更加复杂的功能，在图像处理、多媒体和嵌入式系统等应用领域前景广阔。多处理器片上系统通过对不同应用场景的定制化设计来满足市场对计算平台性能的苛刻要求，这种灵活的，定制化的设计促使多处理器片上系统成为新一代SOC的主流设计趋势。[1]Tilak Agerwala, Siddhartha Chatterijee. Computer architecture: Challenges and opportunities for the next decade[J]. IEEE Micro. 2005. 25(3): 58-69.

多处理器片上系统(MPSoC)是一种由多个处理器单元和专用功能模块构成的一个复杂的单芯片集成计算系统，通常是为嵌入式应用程序而设计，用于满足应用程序对于系统在计算性能，实时性，功耗与成本等方面的需求[1] W. Wolf, “The future of multiprocessor systems-on-chips,” in Proc. 41st Assoc. Comput. Machinery/IEEE Design Automat. Conf. (DAC), 2004, pp. 681–685.

MPSOC不是仅仅是简单的实现片上多处理器(chip multiprocessor)的片上集成，片上多处理器强调将更多的处理单元放在单片上用于提高单位面积晶体管密度，在设计时并不考虑应用的需求，而MPSOC实现的是定制化的集成芯片，通过定制体系结构来满足不同应用在成本和功耗等方面的需求。

MPSOC体系结构常常用于多任务应用程序的计算，有效的利用MPSOC体系结构的异构性可以有效的满足多任务应用程序在性能，功耗和成本等方面的要求，典型的MPSOC体系结构如下图所示，由3部分组成：

（1） 处理器单元，包括处理器（Processor）、数字信号处理芯片（DSP）、专用集成电路（ASIC）用于执行计算任务；

（2）存储（Memory）模块用于保存数据；

（3）通信网络（Network）用于组件之间的通信。



图1 MPSOC体系结构

**MPSOC上的映射与调度问题**

通信、多媒体和计算机网络等领域，将对计算平台固有的成本压力转换成对多处理器片上系统的性能要求，如何高效的发挥多处理器片上系统的计算性能成为目前研究的热点。

MPSOC上的组件通过片上互连总线连接成为一个整体，为适应不同种类应用程序对不同组件的应用需求MPSOC通常被设计成异构的，AMD公司在2006年提出基于“融合”技术的多核处理器产品路线图，实现了同构片上多处理器系统向异构片上多处理器系统的转化。在MPSOC上我们使用不同种类的处理器来加速应用程序的不同部分，从而优化多任务应用程序的性能。MPSOC结构的复杂性使得对应用性能的优化提出了更高的要求。

近年来，对于MPSOC的研究上围绕系统开发，系统安全，编程模型与任务调度等方面的研究已经拥有了一些重要成果。为了优化应用程序的性能，例如，程序执行时间，需要把一个复杂的任务分解，将分解后的子任务映射到异构处理器上，寻找一组合适的映射关系用于满足实时性，成本与功耗等约束条件成为MPSOC系统亟需解决的关键问题之一。然而多处理器上任务的调度与映射问题属于NP难问题[2]。J. Kim, S. Lee, H.l Shin, Y. Lee and H. Bae, “Effective task mapping and scheduling techniques for heterogeneous multi-core systems based on zone refinement,” Computer Sciences and Convergence Information Technology (ICCIT), 6th International Conference. 2011, pp. 363 – 366.

由于异构嵌入式系统自身结构的复杂性与多任务应用程序规模的不确定性，使得传统方法在处理多处理器上任务的调度与映射问题时失去了有效性，特别是针对大规模的多任务应用程序的调度与映射问题。改进算法的效率并构建任务映射与调度机制成为一个亟待解决的问题。

当前研究提出许多基于启发式的方法来优化任务的映射与调度问题，例如禁忌搜索算法（Tabu Search）[3-4]

T. Wiangtong, P.Y.K. Cheung and W.Luk, “Comparing three heuristic searchmethods for functional partitioning in hardware–software code-sign.” Design Automation for Embedded Systems, 6(4):425–449, 2002.

P. Eles, Z. Peng, K. Kuchcinski and A. Doboli, “System level hardware/software partitioning based on simulated annealing and tabu search.” Design Automation for Embedded Systems, 2:5–32, 1997

遗传算法（Genetic Algorithms）[5]

M. Grajcar, “Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system." In DAC’99: the 36th ACM/IEEE conference on Design Automation, pages 280–285, 1999

以及Kernighan-Lin [6]方法

F. Vahid and T.D. Le, “Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning.” Design Automation for Embedded Systems,2(2):237–261, March 1997.

以上方法被证明能够在合理的时间内获得对优化问题的一个局部最优的结果。但是这些方法在处理问题时只关注了映射与调度问题的一个方面，即单一的处理调度或映射问题，并且当求解空间的约束条件比较苛刻时，算法在问题求解上的性能就会严重下降，优化的结果往往不尽人意。蚁群算法[7][8][9]

G. Wang, W. Gong, B. DeRenzi and R. Kastner, “Ant colony optimizations for resource and timing constrained operation scheduling.” IEEE Transactions on Computer-Aided Design of Integrated Circuitsand Systems, 26(6):1010–1029, June 2007.

P.C. Chang, I.W. Wu, J.J. Shann and C.P. Chung, “ETAHM: An energy aware task allocation algorithm for heterogeneous multiprocessor.” In Proceedings of DAC ’08, pages 776 –779, june 2008.

F. Ferrandi, P.L. Lanzi, C. Pilato, D. Sciuto and A. Tumeo, “Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems.” IEEE Trans. on Computer-AidedDesign of Integrated Circuits and Systems, 29(6):911–924, June 2010.

是另外一个在处理映射与调度问题中比较常用的优化方法，使用蚁群算法优化问题时对问题的调度与映射的优化是同时进行的，当优化问题的规模比较小时，蚁群算法显示了其有效性和可行性。但是当面对大规模的优化求解问题时，由于优化空间的增加以及计算复杂度的上升，蚁群算法对与该问题的求解上失去了有效性。因此，目前还没有一个通用的，能够高效的处理复杂应用程序在多处理片上系统的调度与映射问题的优化算法。

本文基于抽象模型给出任务的形式化定义，然后进行调度策略与算法的设计

目标体系结构的抽象（参考论文）

本文我们抽象出一个通用的异构多处理器片上系统（MPSOC）的体系结构用于进行一般化说明，其形式化定义如下：

A=PUC (1)

其中，P表示一组异构的处理单元，用于处理应用程序的不同部分，C表示一组通信组件，用于进行处理单元之间进行数据的传输。一个抽象异构多处理器片上系统的一个简单示例，如下图所示，它由四个异构的处理单元，分布式内存和单一的系统总线作为通信组件组成的。



图2 抽象目标体系结构

多处理器片上系统的任务调度与映射属于资源受限的优化问题，体系结构中每一个组件都关联一组相关的资源Q（例如：内存大小）[10]

P. Brucker, A. Drexl, R. Mohring, K. Neumann and E. Pesch,“Resource constrained project scheduling: Notation, classification, models, and methods,” Eur. J.Operat.Res.,vol.112, no.1, pp. 3–41, Jan.1999

这些资源可以划分成两类：可再生资源R（renewable）和不可再生资源N（nonrenewable）。

（1）可再生资源R即当组件相关的资源在任务释放以后可以再次提供给后续的任务继续使用。例如：本地内存，当任务结束后内存得到释放提供给后续任务继续使用，但资源的类型也与应用程序使用的方式有关，若是在分配内存时应用程序采用的是静态分配的方式，则当任务结束后内存依然为任务所占有，这时的内存又可以被理解为不可再生资源。

（2）不可再生资源N即当组件相关的资源在分配给单一任务后，即便任务已经执行完成但是其占用的资源也不能再次提供给后续的任务使用。例如：现场可编程门阵列（FPGAs）一般情况下就被认为是不可再生的资源。

在图2中我们假设与处理单元P0，P1和p2相关的资源q0，q1和q2为可再生资源，例如，数据内存单元。与处理单元p3相关的资源q3为不可再生资源，例如，采用静态分配的内存单元。我们用Akq来表示组件pk∈A所关联的资源q∈Q的数量，当任务t分配在组件qk上时任务t所占用的资源数量要满足Akq的要求。

对于图2中目标体系结构上的任务映射与调度问题我们还做如下考虑：

（1）在组件上执行任务时，任务要满足一下两个方面的要求：一、任务的种类和组件可执行任务的种类相符，例如，计算任务只能在计算单元上执行，通信任务只能在数据传输总线上执行；二、任务消耗的资源数量要满足组件当前所拥有的资源数量。

（2）如图2所示，每一个组件都关联一个局部内存，当一个计算任务在组件上运行时，对数据的存取均来至与其相关的局部内存。当任务执行结束以后，其产生的数据也存储在局部内存中，因此，如果两个有数据依赖的任务映射到同一个组件上执行，任务之间的数据传输是不消耗任何资源的，因为并不需要发生实际的数据传输任务[11]。

R. Niemann and P. Marwedel, “An algorithm for hardware/software partitioning using mixed integer linear programming, ” Design Automat. Embedded Syst., vol. 2, no. 2, pp. 125–163, Mar.1997.

此外数据的传输可以通过系统总线或DMA进行传送，但是数据在传送时只能和其中的一个传送组件相关联。通过上面的假设，我们可以很容易的将当前目标体系结构的传输模型向其他模型进行扩展[12]。

T.-Y. Yen and W. Wolf, “Communication synthesis for distributed embedded systems,” in Proc. IEEE/Assoc, Comput. Machinery int. Conf. Comput-Aided Design (ICCAD), 1995, pp. 288-294.

例如在共享内存系统中数据的传输过程将会分为以下两步，一、由源执行单元局部内存到共享内存的数据传输；二、由共享内存到目标执行单元局部内存中的数据传输。在本文中，我们假设：一、所有的数据传输都是直接由局部内存到局部内存进行传输的；二、不同内存单元之间数据传输所消耗的时间只和传输的数据量的大小以及通信总线的带宽有关。

（3）为了简化任务的优化调度，我们假设每个组件在同一时间最多只能运行一个相关任务，即当有多个任务分配在同一执行组件上时，这些任务需要线性的执行。

**多任务应用程序**

什么是多任务应用程序

一个多任务应用程序由N个子任务组成，任务之间具有时序关系和数据的传输依赖，每个子任务在运行时需要占用单一的执行单元。多任务应用程序在执行时具有以下特点：

一、多任务应用程序的并行执行并不意味者所有子任务都需要在同一时间执行，而是在同一时间至少存在一个子任务可以被执行。

二、子任务只有在其所有上游任务已经完成并且获得执行硬件的基础上才能够被执行。

三、多任务应用程序完成的时间为最后一个子任务运行结束的时间。

因此，当多任务应用程序在单CPU系统上执行时，所有的子任务在同一时间上只能有一个在执行，应用程序的执行的总时间和应用程序串行执行的时间相同。为了加快多任务应用程序的执行时间，其执行平台往往是上文所提到了多处理器片上系统，应用程序在执行时，可以并行的执行，但是在调度任务和各个处理单元之间的映射关系上，不同的映射将会导致程序执行时间的不同，例如，在最差的情况下，应用程序中的所有计算任务都被映射到同一个CPU单元上执行，这样应用程序完成的时间和单个CPU执行的时间是相当的，这种映射关系肯定不是我们想要的；我们所期待的结果就是将子任务合理的映射到多个处理单元上，让应用程序充分利用多处理器片上系统的计算优势，加快任务的执行速度，缩短程序的执行时间。

**多任务应用程序的抽象**

在处理异构嵌入式系统上问题的映射和调度时需要将多任务应用程序抽象成能够在目标平台上进行调度的模型。本文我们将多任务应用程序抽象成一个有向无环图（DAG），其形式化定义如下：有向无环图*G* = (*T, E)，*顶点T表示计算任务，E ⊆T×T表示任务之间的依赖关系，即存在e（t’, t）∈E时表明任务t’是任务t的父任务，任务t只有当任务t’执行完成以及和任务t相关的数据传输完成后才能执行。其中，每条边都关联一个数据传输大小的标量，表明由父任务到子任务之间数据传输的大小。一个多任务应用程序的抽象模型如下图所示。拥有这类模型的应用程序主要有科学计算和多媒体应用程序，这类程序在执行时往往具有很的高并行。



图3 多任务应用程序模型

由于上述的多任务应用程序的运行时行为在根据任务进行划分后是可以静态预测的，因此，对多任务应用程序的优化可以是离线进行的。由于进行子任务划分后，任务之间可能会参数较大的数据传输，因此，在进行多任务应用程序的优化时对通信的开销也是必须要考虑的因素。

我们将G=（T,E）和上文定义的目标平台A=PUC进行关联后对多任务应用程序进一步进行以下抽象描述：

任务j(job)被定义为在目标平台的组件上执行的一个活动单元，多任务应用程序的每一个执行任务和任务之间的传输行为都被抽象成一个j，整个多任务应用程序被抽象成一个任务的集合用J来表示。

执行点i被定义成资源和时间的组合，用来代表任务j在组件ak上执行时所需要的最低的资源要求和需要占用的时间，即任务j在ak上执行的时间。在具体的目标平台上，每一个任务j都有一到多个执行点可供选择来执行自身的任务，只要执行点满足任务j所需的约束执行的条件，若是不满足任务j是不能分配到该执行点上执行的，例如，计算任务j不能被分配到传输总线上进行来执行。此外，并不是所有的处理单元都可以用来执行计算任务，例如。一个任务包含了不能合成的结构。任务j在不同的执行点i上的组合，构成了应用程序在硬件上的解决方案。执行点的集合I包含了任务集合J在目标平台A上所有可用的执行点，表格I对多任务应用程序G在目标平台A上执行点的集合I进行了展示。表格中的每一项表明了任务j在执行点i上执行时需要的时间（例如，时钟周期）和需要消耗的资源数量（内存大小），对于每一个计算型的任务j，在P3这个计算单元上可以有两个不同的执行点，这是因为这两个执行点是根据任务在P3上消耗的资源数量不同而产生的，是任务在执行时占用资源和时间的平衡的结果，一般情况下，占用的资源数量越多，任务运行时消耗的时间就越小。

表I 任务J和执行点I之间的映射关系表

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| job | pe0 | | pe1 | | pe2 | | pe3 | | | | cc | | |
| i0 | | i1 | | i2 | | i3 | | i4 | | i5 | | |
| t | rr | t | rr | t | rr | t | rr | t | rr | t | rr |
| A | 3 | 6 | 7 | 4 | 3 | 8 | 6 | 3 | 2 | 9 | null | null |
| B | 5 | 6 | 2 | 2 | 7 | 9 | 2 | 3 | 1 | 12 | null | null |
| C | 6 | 8 | 2 | 9 | 2 | 10 | 10 | 9 | 12 | 2 | null | null |
| D | 9 | 12 | 4 | 8 | 19 | 5 | 7 | 13 | 7 | 9 | null | null |
| E | 3 | 6 | 7 | 4 | 3 | 8 | 6 | 3 | 2 | 9 | null | null |
| F | 8 | 2 | 7 | 12 | 1 | 12 | 2 | 9 | 11 | 7 | null | null |
| G | 12 | 3 | 12 | 7 | 1 | 9 | 11 | 12 | 9 | 10 | null | null |
| (A,B) | null | null | null | null | null | null | null | null | null | null | 2 | 8 |
| (A,C) | null | null | null | null | null | null | null | null | null | null | 4 | 8 |
| (B,E) | null | null | null | null | null | null | null | null | null | null | 5 | 8 |
| (C,E) | null | null | null | null | null | null | null | null | null | null | 7 | 8 |
| (C,F) | null | null | null | null | null | null | null | null | null | null | 3 | 8 |
| (D,F) | null | null | null | null | null | null | null | null | null | null | 1 | 8 |
| (E,G) | null | null | null | null | null | null | null | null | null | null | 2 | 8 |
| (F,G) | null | null | null | null | null | null | null | null | null | null | 2 | 8 |