工学硕士学位论文

支持动态任务调度的多核分布式操作系统设计

DESIGN OF A DYNAMIC SCHEDULING SUPPORTED DISTRIBUTED OS FOR MPSOC

胡新安

哈尔滨工业大学

2011 年 6 月

国内图书分类号：TN402　　学校代码：10213
国际图书分类号：621.38　　密级：公开

工学硕士学位论文

支持动态任务调度的多核分布式操作系统设计

硕 士 研 究 生：胡新安
导　　师：喻明艳 教授
申 请 学 位：工学硕士
学　　科：微电子学与固体电子学
所 在 单 位：微电子科学与技术系
答 辩 日 期：2011 年 6 月
授予学位单位：哈尔滨工业大学

Classified Index: TN402
U.D.C.: 621.38

Dissertation for the Master's Degree in Engineering

DESIGN OF A DYNAMIC SCHEDULING SUPPORTED DISTRIBUTED OS FOR MPSOC

Candidate: Hu Xin'an
Supervisor: Prof. Yu Ming-yan
Academic Degree Applied for: Master of Engineering
Specialty: Microelectronics and Solid-State Electronics
Affiliation: Dept. of Microelectronics Science and Technology
Date of Defence: June, 2011
Degree-Conferring-Institution: Harbin Institute of Technology

摘　要

多核系统资源分布的全局复杂性给资源管理和利用带来困难，利用多核操作系统管理全局资源能有效提高系统资源利用率，因此对多核操作系统进行研究具有重要意义。本文使用非均衡模式设计多核分布式操作系统，需要解决节点间通信和系统资源利用的问题。为解决通信问题，本文移植了实验室已有的兼容于 MPI 的消息库，节点间通信通过应用层、消息层和硬件抽象层中的功能模块完成。本文工作的主要目标是为用户透明使用多核系统资源提供支持，为此实现了运行时调度功能，以及基于运行时调度结果的并行应用执行。

主控节点实现了资源统计和运行时调度功能。统计模块建立多个接收进程实现资源信息的收集，通过使用共享内存创建资源池以实现全局资源信息存储。在此基础上采用动态任务调度技术，在程序执行时进行任务调度和分派，以当前可用系统资源信息为依据，将并行任务分配到运算节点上执行，从而有效提高系统资源利用效率以及应用加速比。此外，为保证可扩展性，统计模块在读入网络配置文件后将自动建立统计环境，实时地为动态任务调度提供与运算节点通信所需的映射文件。

运算节点提供了扩展层功能，通过接收主控节点发送的配置信息，运算节点只统计所需信息项。通过在系统初始化时设置映射信息，资源统计任务和加载任务能够在不同网络规模下与主控节点完成通信从而保证系统的可扩展性。为支持运行时调度，运算节点接收主控节点以消息的形式传递的调度映射结果，使用分页机制为并行任务分配存储调度信息的私有空间，并通过参数传递将调度结果等信息传递给并行任务，从而保证各并行任务正确执行当前功能。

主控节点和运算节点协作完成并行应用的调度、分发、任务建立和执行，实现了运行时调度和执行并行应用的功能，为有效利用系统资源提供了支持。最后，在多核仿真平台上，通过执行并行应用对分布式多核操作系统进行了验证，验证结果表明该多核操作系统能够正确地执行基于 MMPI 的并行应用程序；测试了一个并行应用执行过程中调度、分发、任务建立和执行各阶段的开销，评估了应用规模和网络规模对调度结果与各阶段执行开销的影响，为系统及算法优化指出了方向。本文开发的多核操作系统可以用于嵌入式多核系统资源管理，评估调度算法的执行开销和调度结果等，具有较高的应用价值。

关键词：多核系统；非均衡模式；分布式操作系统；运行时调度；分页机制

Abstract

It is difficult to manage the global resource because of complexity for the MPSoC systems. The key requirement of effective utilization of MPSoC and speedup execution of parallel programs relies on management of the resource by OS for the MPSoC systems, so it's necessary to study OS for MPSoC. The asymmetric Multi Processing model is applied to design a distributed OS for a NoC-based MPSoC, in which nodes are divided into Control node and Operation nodes. A costuming Linux runs on the control node as the interface between end-users and the hard-subsystem, while an extended uC/OS on each operation node to get high performance and support execution of parallel applications.

There are two modules on the control node. The resource stat module creates multiple processes to collect resource information and stores these in the stat pool which is constituted of shared memory. Dynamic scheduling technique is adopted to assign tasks to operation nodes by the scheduling module. In addition, the stat module automatically establishes the stat environment and provides required mapping file for the two modules after reading the network configuration file so that the system will hold high scalability.

The uC/OS is replanted to ARCA3 and task switch, interrupt handle, network interface driver, and dynamic memory management are implemented. In the OS extension layer, the stat task collects required information determined by the configuration send from control node. By setting the mapping result at the initialization stage, the stat task and load task can communicate with the control node accurately regardless of the scale of the network. Then the load task gets the mapping result, uses paging system and parameter passing to create, load and execute MMPI tasks assigned by control node.

The control node cooperates with operation nodes to schedule, assign, load and execute parallel programs. The distributed OS is validate, a test case with four tasks executes on a three-core platform, which includes one control node. The result proves the validity of the OS function, and the cost of every stage such as scheduling, assigning, loading and executing is evaluated, then the influence to the cost of each stage of the system size and application size is estimated. The distributed OS can be used in the embedded systems and to evaluate dynamic scheduling algorithms.

Keywords: MPSoC, Asymmetric Multi-Processing model, Distributed OS, Runtime Scheduling, Paging System

目　录

摘　要 I
Abstract II
第 1 章 绪论 1
1.1 课题背景及研究意义 1
1.2 国内外研究现状 3
1.2.1 多核操作系统研究 3
1.2.2 动态调度研究 4
1.3 课题主要研究内容 7
1.3.1 主控节点操作系统研究 7
1.3.2 运算节点操作系统研究 7
1.3.3 运行时任务调度 8
1.4 论文结构 8
第 2 章 系统总体设计结构 9
2.1 多核硬件系统 9
2.2 多核操作系统设计模式 10
2.2.1 裸核模式 10
2.2.2 均衡模式 11
2.2.3 非均衡模式 11
2.3 非均衡模式下系统结构 12
2.3.1 主控节点和运算节点 12
2.3.2 系统运行过程 13
2.3.3 并行应用执行过程 15
2.4 本章小结 17
第 3 章 主控节点操作系统 18
3.1 主控节点操作系统概述 18
3.2 资源统计模块 18
3.2.1 资源池建立 19
3.2.2 资源信息统计 20
3.2.3 映射文件生成 21
3.3 并行调度模块 21

3.3.1 通信域的分配 22
3.3.2 运行时调度 24
3.3.3 动态调度算法 25
3.4 本章小结 26
第 4 章 运算节点操作系统 27
4.1 运算节点操作系统概述 27
4.2 uC/OS移植 29
4.2.1 任务建立与切换 29
4.2.2 中断处理 31
4.2.3 DMA_NI驱动开发 33
4.2.4 内存管理 35
4.3 系统扩展层 37
4.3.1 资源统计 38
4.3.2 MMPI任务加载 40
4.3.3 虚拟内存的分页机制 42
4.3.4 并行编程 44
4.4 本章小结 45
第 5 章 功能验证与测试 46
5.1 仿真平台设置 46
5.2 操作系统基本属性 46
5.3 功能验证 47
5.3.1 实验设计 47
5.3.2 验证结果 48
5.4 性能测试 49
5.4.1 uC/OS下MMPI通信性能 50
5.4.2 应用规模和网络规模对各阶段开销影响 50
5.5 本章小结 52
结 论 53
参考文献 54
攻读学位期间发表的学术论文 58
哈尔滨工业大学硕士学位论文原创性声明 59
哈尔滨工业大学硕士学位论文使用授权书 59
致 谢 60

第1章 绪论

1.1 课题背景及研究意义

随着集成电路制造工艺的发展和应用需求的增长，单处理器 SoC 已不能满足日益复杂的应用需求，由此带来的并行处理的需求使得集成电路设计向多核系统（Multiprocessor System-on-Chip，MPSoC）[1,2]发展。MPSoC 是指在单一芯片上集成多个处理器的复杂 SoC，它将多个可编程处理器作为设计核心，其电路规模与片上复杂度远超单核系统的设计[3]。影响 MPSoC 性能的关键在于核与核之间的通讯效率[4,5]，MPSoC 可以采用单一总线、多总线、点对点、片上网络 NoC[6,7]（Network-on-Chip, NoC）等多种通信方式[8]。其中 NoC 是为了解决片上多核系统中全局通信问题而提出的一种全新方案，它具有良好的扩展性和并行通信能力，全局异步局部同步（Globally Asynchronous Locally Synchronous, GALS）[9]机制以及高 IP 重用度等优点[10]，越来越多的基于 NoC 通信架构的多核系统被提出和实现[11-14]。NoC 注重于解决多个节点之间的通信问题，以提高通信的稳定性和性能为目标，MPSoC 注重于多核系统，以利用多核并行计算能力，提高多核系统性能为目标，两者的结合体现了 SoC 的发展从追求单核的指令级并行、某些总线架构的多核系统中的进程级并行到任务级并行的趋势[15]。

MPSoC 系统面临的一个重要问题在于对多核系统资源的利用[16]。MPSoC 系统资源具有复杂性[17]，系统中的计算资源分布在各个节点，通信资源则包括路由网络中的路由器及通信链路。系统中每个节点的资源使用和运行状态各不相同，NoC 网络中各个路由器及各段链路的占用也是不相同的，并且多核系统缺少从全局管理多种多样资源的机制。但是与多核系统全局资源复杂化的特点相对应的则是多核系统中单个节点的简单化，特别是同构 MPSoC[18]。长久以来人们通过提高单核处理器的性能来满足新的应用需求，单核处理器的性能不断提升的同时其规模越来越庞大，并最终受到功耗约束、设计生产力差距等问题的困扰[19]，通过复制简单模块来获得具有较高处理能力的系统是促进 MPSoC 发展原因之一[20]。同构 MPSoC 基于瓦片的设计概念而产生，通过简单复制与规则连接生成多核系统，具有可扩展性强[21]、容易提高运行资源利用率[22]等优点。在这样的多核系统中，各个节点的规模与复杂度都较低，而由于

资源的分布带来了全局资源监测以及管理的复杂性。

对多核系统资源的利用与并行应用的执行密切相关，并行应用在 MPSoC 系统上执行包括调度映射、分发和执行等几个步骤，其中调度和映射是决定并行应用执行性能的一个关键因素[23]。任务的映射调度，是在并行应用任务信息和多核系统资源信息的基础上，通过映射和调度的机制把并行应用的任务安排到多核系统中执行和通信。任务映射就是将并行应用中的任务，用一定的规则对应到多核系统中的处理单元（即 IP 核）上去执行，由于一个处理单元上可能被分配了多个任务，所以还需要通过任务调度来合理的安排每一个处理器核上任务的执行顺序，调度和映射的过程如图 1-1所示。任务调度和映射是在已有的资源上根据并行应用中任务的参数特征进行系统资源的分配，可以提高系统资源的利用效率[24]。

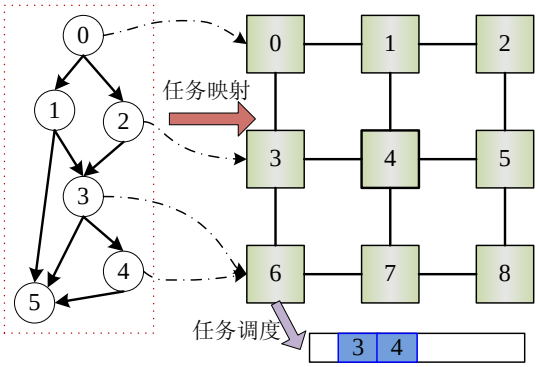


图1-1 调度和映射过程

并行应用的调度映射分为静态调度和动态调度两种[25]。静态调度是在任务执行前已经确定映射和分配结果，例如在编译时确定调度结果或在线下进行调度，任务执行时根据调度结果被分配到不同的处理节点调度执行。由于在运行前确定了调度映射，因此任务与处理节点之间的分配关系在运行前已然确定，静态调度下对任务的监控和调试比较方便。并且由于调度与执行分离，静态调度可以使用非常复杂的调度算法，能够充分考虑和利用并行应用的各种特性从而获得较好的调度效果。动态调度是在并行程序执行时根据调度规则、多核系统可用资源以及并行任务自身的特性来分配并行任务。静态调度不能利用系统的实时资源情况信息，当系统资源发生变化时，无法根据资源分布做出动态调整，因而不能合理的利用系统资源，浪费系统的并行处理能力。但是由于动态调度是在并行应用执行时调度，其开销也成为并行应用执行开销的一部分，动态调度的开销过大会影响并行应用的执行效率，因此动态调度算法一般较为简单，能够在短时间内做出调度和映射。

操作系统为并行应用的分发和调度提供支持[26,27]，此处，操作系统还具有

一个重要作用就是监测、管理系统资源[27]。多核操作系统可以监测和统计多核系统的资源并将资源信息交予调度映射算法使用，开发多核操作系统对提高并行应用利用多核系统资源效率、增加并行程序的可移植性和系统的可扩展性具有重要意义[29,30]。本文设计了应用于基于 NoC 的 MPSoC 的分布式多核操作系统，通过检测多核系统全局资源变化，并在并行应用执行时进行运行时调度，为并行程序利用系统资源提供支持。

1.2 国内外研究现状

1.2.1 多核操作系统研究

Andrew Baumann 等人[31]认为为了利用多核系统的资源，必须接受并利用其网络功能的本质特点，这需要在设计 OS 时借鉴分布式系统的观点，把多核系统看作一组通过网络互连的独立的节点，并假设在底层没有节点间的资源共用。文中提出了微内核操作系统的三个设计原则：核间通信显式进行；操作系统结构与底层硬件无关；以复制而不是共享的观点看待资源。此外，Chao WANG 等[32]也认为基于 NoC 的 MPSoC 与分布式系统有很多相似之处，NoC 网络对应着网络连接，通过 NoC 互连的多核系统可以看作是分布式系统的芯片化实现。分布式系统中动态任务调度已得到了广泛研究，但两者的节点规模、可使用的资源等设计约束的不同，使得宏观上的分布式系统的研究结果并不能直接应用到 MPSoC 系统上。

Vincent Nollet 等人[33]通过在操作系统中集成通信资源管理来合理地将并行应用中的任务映射到多个节点，在系统中实现了动态信息统计、动态注入率控制、操作系统控制自适应路由等功能。其独到之处在于将一个操作系统拆分成多个部分，其中操作系统内核运行于主节点上，而每个从节点则执行有限的操作系统功能。操作系统的某项功能实现就如同远程调用（remote procedure calling, RFC），主节点将参数通过消息发送到对应的从节点，从节点完成相应的功能后通过消息将结果返回给主节点。

Wei HU 等人[34]认为前文提出的集中式控制的系统会形成热点并会导致主节点和从节点之间负载的不平衡。该文提出的 OS 由一个微内核和一系列运行于这个内核之上的多个软件模块组成，每个节点执行微内核或者软件模块或者两者都有。基于微内核的操作系统仅包含最基本的、很少的功能在内核中，而其他功能模块是在用户态下实现。微内核系统中内核与功能模块的分离使得其适用于分布式系统，包括 NoC 架构。在实现中内核与软件模块被划分为不同

的部件，每个核上运行一个基本的微内核，然后不同的软件模块分布到不同的核上执行，图 1-2是该文提出的基于微内核的分布式多核操作系统的结构示意图。应用程序通过运行于每个核上的微内核调用系统功能模块，当该功能模块不在当前节点上时，产生发送到其他节点的消息，由其他节点完成相应的功能。这种将操作系统由分布在一个核上的转变为分布到多个核上的设计使得现有程序不需要修改就可以运行，但是其同样存在热点的问题，由其评估结果可以看出其文件系统模块接收了超过 80%的消息，所以文件系统模块所在的节点也会成为系统中的通信热点。

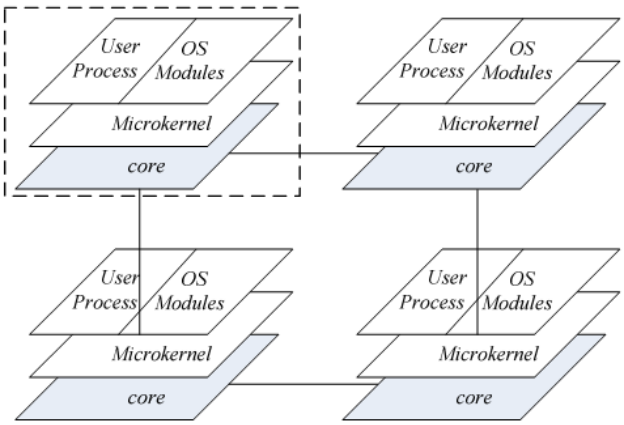


图1-2 基于微内核的分布式操作系统

1.2.2 动态调度研究

S.Vakili 等[35]比较了静态调度和动态调度，指出静态调度在寻找调度映射的最优解上具有潜在的优势，因为程序员或者编译器能够完整地了解应用程序并能够比较各种不同解的优劣。而动态调度需要迅速地根据可用资源信息和待调度任务的特性来完成调度，因此不可能详细地比较各种方案。另一方面，对于静态调度，系统资源必须在调度前已经确定并且不能改变，因此静态调度限制了并行程序的扩展性，而动态调度则不存在这个问题。在静态调度的多核系统中，控制信息和同步信息包含到程序中，但动态调度并非如此，因此一般需要一个专门的调度单元来完成控制和同步。调度单元可以由操作系统的模块、中间件或者硬件实现。该文中提出了 EvoMP 系统结构，其中的动态调度单元（Dynamic Scheduling Unit, DSU）是由硬件实现的，图 1-3是 EvoMP 系统的体系结构图。EvoMP 系统使用了共享存储器，存储器及内存管理单元总是挂接在 NoC 网络中 ID 为 00 的节点上。内存管理单元包含一个指令解析的单元用于寻找 load 和 store 指令，并负责将数据发往需要的节点，将接收到的数据写

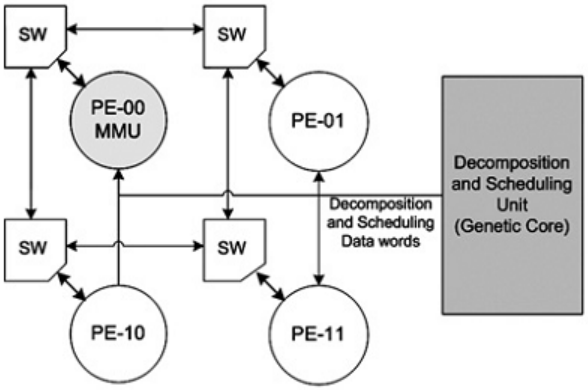


图1-3　EvoMP 系统中的调度单元

入对应存储单元。

J. Barbosa 等[36]指出动态调度和静态调度的目的并不完全相同：静态调度追求最小化单个应用的调度开销，获得尽量短的执行时间；动态调度的目的是提高整个系统的性能，特别是在多核系统执行多个并行应用的情况下。动态调度可以从调度策略（scheduling strategy）和调度算法两个方面来进行研究。调度策略决定何时、对哪些对象进行调度，调度算法决定如何根据并行任务的特性和系统负载的情况来分配和调度并行任务。文中提出了两种调度策略，分别是即时策略和批策略，即时策略只对新的并行应用中的任务进行调度；批策略则在有新的应用执行时，对各个节点上已分配但还未执行的任务进行回收，和新的应用一起重新进行调度。从调度结果上比较，批策略能具有更好的性能，但批策略在实现上需要关注各个节点上分配任务的执行情况，其过程更加复杂，在回收-重分配的过程中可能会引入额外的拷贝开销，因此当应用的规模较小时，批策略不会提升性能。

Juan 等[37]使用了中间件来实现任务的映射。该系统中包含了一定数目的可重配置单元(Reconfigurable Units, RU)以及处理器、DSP、GPU 等，在可重配置单元上任务可以被重新配置并执行，由中间件负责将任务分配到处理单元上。为了在找到一个较好的调度方案的同时不引入过多计算开销，该系统使用了一种混合的设计/运行时映射方法。在设计时并行应用的任务图被分析并得到一些在调度时会使用的有用信息。该系统使用权重、延时以及迁移率来描述任务的特性，权重决定运行时调度算法重配置的次序，重配置的第一批单元将对任务图中的关键路径产生重大影响；延时指明了每个任务重配置引入的延时，延时较大的任务将被赋予较高的优先级；迁移率则用来在重配置过程中取

消重配置任务所在节点所进行的调度优化。此外，他们还比较了进行重配置前后的并行应用的性能变化以及软件和硬件实现调度器的性能开销。软件实现产生了巨大的开销，这是由于调度中使用了复杂的数据结构以及较多的软硬件通信。图 1-4是多核系统的任务映射过程以及采用不同调度算法的调度结果。

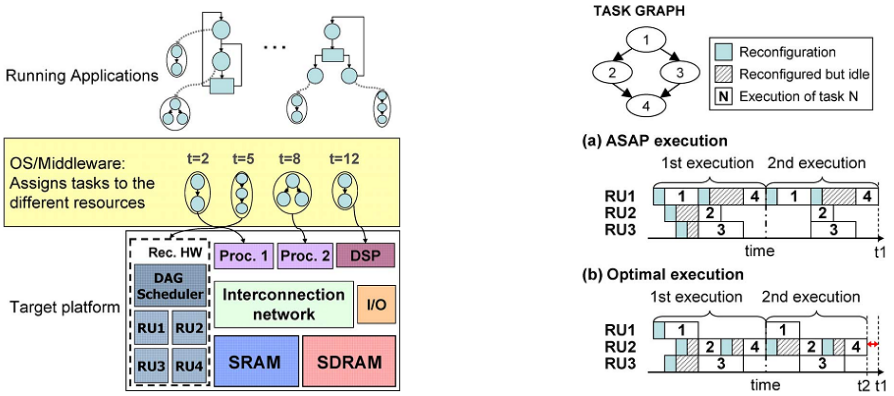


图1-4 任务映射过程及不同调度算法下调度结果

Zexin Pan 等[38]使用专门的硬件调度模块（Dynamic Task Scheduling Unit, DTSU）来完成动态任务调度，DTSU 模块中包含 Task table、Task Issue Unit、Task Priority Assignment Unit、Task Queue、Task Scheduler 等几个子模块，模块的示意图如图 1-5所示。系统中的处理单元分为 Idle、Reconfiguration、Wait、Switch 和 Run 等几个状态，在不同的状态下处理节点可执行不同的操作。DTSU 单元用来执行任务调度算法，根据处理节点的资源信息，包括处理节点所处的状态、任务的表示号和任务类型等来调度任务队列中的任务。DSTU 在两种情况下执行，第一种是系统中某个逻辑单元或处理器进入了空闲状态，第二种是任务队列中有新的并行应用进入时。

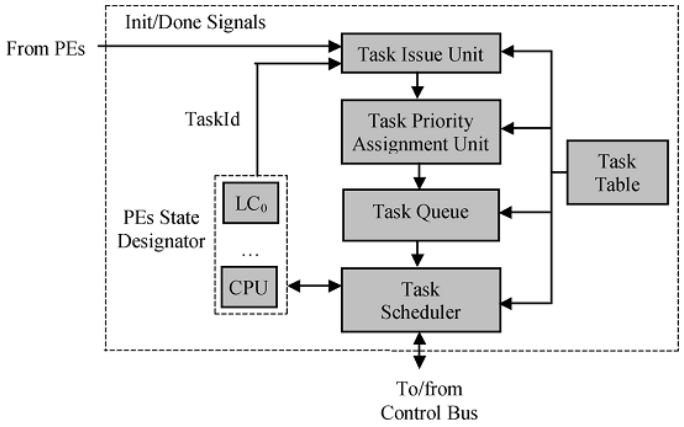


图1-5 硬件调度模块 DTSU 的结构示意图

1.3 课题主要研究内容

为了调高多核系统的性能，本文使用动态调度策略来调度并行应用，在并行程序执行时由操作系统中的调度映射模块根据系统信息决定并行任务分配。本课题主要的研究内容是在实验室已有的基于 M5 的多核仿真平台上[39]实现开发支持动态调度的分布式多核操作系统。已有多核系统仿真平台模拟的是一个同构系统，所有节点使用 ARCA3 处理器，每个节点上运行独立的 Linux 操作系统。该多核系统采用分布式存储器，各个节点具有独立的地址空间，节点上的存储器默认为无限，能够存储下应用程序以及相关数据。采用 NoC 作为通信架构，不同节点之间采用多核消息传递接口（MPSoC Message Passing Interface，MMPI）[40]进行通信。

本文 MPSoC 系统的硬件子系统保持不变，仍采用基于 NoC 的通信架构和分布式存储器。为支持动态调度，需要对系统中各个节点的资源信息进行实时监测和统计；在有新的任务进入系统时，需要根据新任务的特性和当前系统资源进行任务映射和调度。本文的研究内容主要分为以下几部分：

1.3.1 主控节点操作系统研究

对于主控节点的操作系统，需要完成系统的初始化，提供对 MPI 通信机制的支持，统计运算节点的资源信息，分析系统负载分布情况。在有新的并行应用提交给系统时，主控节点的操作系统需要根据应用的特性进行任务映射和调度，并将并行任务分发给各个运算节点。主控节点不执行并行应用，只负责监测整个系统的运行情况，控制多核系统中运算节点执行并行任务。

1.3.2 运算节点操作系统研究

运算节点执行主控节点分配给其的任务，只执行计算任务，运算节点的操作系统只须提供最基本的任务调度即可。作为片上系统，片上存储器的容量十分有限，因此应严格定制运算节点的操作系统。此外，运算节点上可以使用操作系统，也可以使用一个提供最基本调度功能的中间件。但无论使用操作系统还是中间件，都应该尽量减小其所需的存储空间，提高执行效率。并行计算中不同任务间传输数据、主控节点收集资源信息以及分配任务都需要在不同节点间进行通信，因此运算节点操作系统需要提供对 MMPI 库的支持。

1.3.3 运行时任务调度

任务调度分为调度策略和调度算法两部分，调度策略决定何时、由谁、来调度哪些对象，调度算法决定如何映射和调度任务。当使用静态调度时，主控节点在有应用到来时根据并行程序和映射结果将任务发送到不同运算节点上执行。对于动态调度，则是在系统已经在运行某个或某些并行应用的情况下有新的并行应用到来，此时主控节点需要实时统计所有运算节点的统计信息，根据系统的负载情况来映射和调度并行应用。由于动态调度可使用的系统资源是变化的，并且动态调度是并行应用执行时进行的调度，因此为了减少系统资源变化带来的影响，降低运行时开销，动态调度算法应该采用计算量较少、复杂度较低的算法。

1.4 论文结构

第一章为绪论，介绍了多核分布式操作系统的研究背景，已有的多核操作系统研究，动态调度和静态调度，并给出了本文的主要研究内容；

第二章介绍了多核系统的总体设计结构，阐述了多核操作系统的设计模式，介绍了硬件系统结构和分布式操作系统的运行过程，以及在实验室已有的多核平台下和本文的多核系统下并行程序的执行过程。

第三章介绍了主控节点操作系统的设计，阐述了主控节点选用 Linux 操作系统的原因，并介绍了在 Linux 资源统计模块和并行调度模块的实现。

第四章介绍了运算节点操作系统的设计，介绍了 uC/OS 操作系统的结构，描述了移植 uC/OS 系统中完成的工作以及为提供资源统计功能加载并行应用而在操作系统扩展层提供的功能。

第五章对多核分布式操作系统进行了评估，首先验证了分布式操作系统的正确性，然后从通信开销和并行程序执行过程对该系统执行并行应用的性能进行了测试。

最后是本文的结论。

第2章 系统总体设计结构

多核操作系统是影响多核系统资源利用效率的重要因素，其设计与多核硬件体系结构紧密相关。本章首先介绍了多核操作系统设计中使用的多核系统硬件结构，然后论述了多核操作系统设计模式，应用其中的非均衡模式提出了分布式多核操作系统设计结构。

2.1 多核硬件系统

本文多核操作系统所使用的多核硬件系统整体结构如图 2-1所示，该系统包括通过片上网络连接的多个处理器以及通过多级总线连接的 I/O 设备和片外存储器。所有节点中有一个节点作为主控节点运行主控操作系统，其他节点作为运算节点。

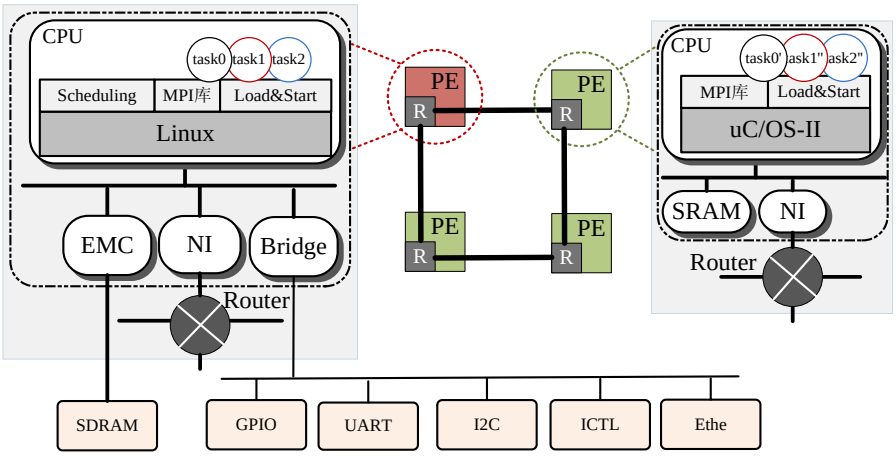


图2-1 多核硬件系统结构图

图 2-1中左侧是主控节点结构示意图，主控节点中包含一个 ARCA3 CPU， CPU 通过 AHB 总线与网络接口（Network Interface，NI）、外部存储器控制器（External Memory Controller，EMC）和一个总线桥相连，CPU 通过总线桥访问低速 I/O 设备，包括 GPIO、UART 等，主控节点通过网络接口和片上网络与其他节点相连。主控节点通过存储器控制器可以访问一个大容量的片外存储器。图 2-1右侧是运算节点结构示意图，运算节点结构简单，仅包含一个 ARCA3 CPU、容量有限的片内存储器和用于通信的 NI。

本文中的多处理器硬件系统采用分布式存储器结构，各个节点拥有私有的存储器并拥有独立的地址空间，各个节点的 CPU 执行的程序与所需的数据存

储在本地存储器中。由于使用分布式存储器结构，不同节点间的通信通过 MMPI 来完成，MMPI 是一套定制的、针对嵌入式应用的消息接口，可以高效地完成数据传输。分布式多核系统具有良好的扩展性，能够扩展到更大规模的系统，在性能提升上具有更大的潜力。本文设计了应用于分布式多核系统的多核操作系统，其整体结构具有良好的模块化特性，能够充分利用分布式多核系统的扩展性。

该多核硬件系统通过 M5 仿真平台模拟实现。M5 是由密歇根大学设计的一个使用面向对象设计、可灵活配置、事件驱动的仿真平台，M5 仿真器能够提供全系统仿真（Full System Mode, FS）。在 FS 模式下，M5 模拟了包括 CPU、存储器、总线以及多种外设在内的整个计算机系统，并能够运行操作系统代码。M5 使用面向对象设计模块化描述系统中的各个模块，能够更真实的模拟硬件模块的各项功能，并且也可避免修改一个模块影响到系统中其他模块。M5 使用 C++完成对硬件模块的建模，使用 Python 来完成对系统的描述、构建以及对硬件模块的配置，这种实现方式可以在不修改硬件建模的情况下搭建不同的仿真系统并对硬件模块进行不同的配置。M5 还具有强大的调试功能，可以通过追踪标记来控制 M5 记录不同硬件模块的活动信息，跟踪硬件模块执行的动作、时间以及与软件的交互等，为操作系统的调试提供了极大的便利。本文对实验室已有的 M5 多核仿真平台进行了修改，精简了运算节点的结构，仅保留构成最小系统所需的 Timer、中断控制器和本地存储器等模块，为主控节点保留了通过总线连接的多种低速 I/O 设备，从而构建出运行分布式多核操作系统的硬件平台。

2.2 多核操作系统设计模式

操作系统的设计有三个目标：作为用户/计算机接口，作为资源管理器并具有功能的易扩展性[41]。多核操作系统的设计首先必须为并行应用最大化利用多核系统资源提供支持，此外还需对系统启动、应用加载、文件系统、网络等通用功能提供良好支持。多核操作系统的设计有三种通用的模式，分别是裸核模式、均衡模式和非均衡模式[42]，下面对这三种模式予以详述。

2.2.1 裸核模式

裸核模式（Bare Metal Mode）下多核节点上不执行操作系统，只执行单一的进程或线程，该模式下并行程序的 API 是与处理器架构相关的。多核系统并不能完全采用裸核模式，一般情况是在系统中某些核上运行操作系统，提供对

硬件系统的抽象，而其他核上执行任务线程。

在裸核模式下，对于未运行操作系统的节点，所有的处理能力都用来执行任务线程，消除了操作系统的执行开销，并行应用可获得最好的性能。但应用裸核模式设计的多核系统有许多缺点，首先由于其 API 与处理器架构相关，需要开发专用的 API，与之相关的并行应用程序具有硬件相关性，增加了并行应用程序的开发难度；其次缺乏观察运行于裸核模式的节点的手段，用户无法监测这些节点的状态，也无法调试执行的并行程序，这些核对于系统相当于一个“黑盒”；最后，为应用裸核模式开发的多核系统编写的程序的可移植性和扩展性很差。除专用 API 外，裸核模式下的软件设计还必须考虑可用的处理单元的数目，并将程序拆分成相同数目并行任务，一旦系统节点数目发生变化时并行程序不能利用增加的计算能力，某些情况甚至需要重新拆分并行任务。

2.2.2 均衡模式

均衡模式（Symmetric Multi Processing model）下各个核上执行完整的并且具有相同或相似功能的操作系统，例如 Linux 或者 Windows 操作系统。均衡模式下操作系统为底层硬件提供了较高程度的抽象，具有成熟的同步机制与负载平衡机制，给资源管理带来了极大的便利。在服务器或桌面应用中的并行机采用的正是均衡模式，这种模式对于密集计算的应用具有较好的执行效果。但较高程度的抽象一方面有助于提高并行程序的可移植性，并且易于进行并行编程，但另一方面也带来较为显著的开销，导致性能下降。此外，均衡模式下使用的操作系统多使用共享部件，例如共享 Cache 或共享存储器作为多核之间的通信途径，这会成为多核系统的性能瓶颈，也限制了系统的可扩展性。在某些实时应用中，使用共享部件会给系统的运行带来不确定性，这也限制了其在多核系统中的应用。

2.2.3 非均衡模式

非均衡模式（Asymmetric Multi Processing model）下各个核上执行相互隔离的操作系统，这些操作系统可以是功能完全的操作系统，也可以是轻量级的嵌入式操作系统。由于系统中某些节点可以执行轻量级的操作系统，因此系统可以获得较好的性能，同时由于各个节点上的操作系统为底层硬件提供了抽象，也为资源管理提供了基础，并且应用非均衡模式开发的多核系统具有良好的可扩展性和并行程序的可移植性。非均衡模式的问题在于各个节点运行相互隔离的操作系统，每个节点形成一个独立的系统，因此对分布式的全局资源的

管理较难实现，对并行应用的加载和执行也很复杂。

非均衡模式提供了一种折中方案，既对底层硬件提供了一定的抽象，又因运行在某些节点上的轻量级操作系统保证了较好的性能。此外，使用非均衡模式结合虚拟化技术可以使多核系统的多核特性对于使用者透明，已有的单核应用可以运行于一个单独的节点上从而使多个应用并行执行，这种情况下多核系统的并行特性可以被利用并且已有的应用不须修改，大大拓展了多核系统的应用前景。本文应用非均衡模式设计了应用于基于 NoC 的 MPSoC 分布式多核操作系统，该系统中节点分为主控节点和运算节点两类，每一类节点上运行不同类型的操作系统，两种操作系统形成一种分层结构，实现了对多核系统全局资源的管理，为用户/多核系统提供了一个良好的接口，并为并行应用利用多核系统资源提供支持。

2.3 非均衡模式下系统结构

多核系统与分布式系统有许多相似之处，许多分布式系统的研究结果可应用于多核系统。分布式操作系统是供互连的计算机使用的操作系统，展现给用户的是一个普通的集中式操作系统，提供用户透明访问不同机器资源的能力。本文采用非均衡模式设计应用于多核系统的分布式操作系统来管理多核系统资源。分布式操作系统的设计需要解决两方面问题：首先是为不同节点间的通信提供支持；其次是提供透明利用系统资源的能力。

本文的多核分布式操作系统将 I/O 访问、文件系统等功能集中于某个或某些节点，而计算工作由其他节点完成，根据完成工作的不同节点被划分为主控节点和运算节点。运算节点的结构简单，运行于其上的操作系统只须具备最简单的任务调度和支持消息库的功能，可以裁减到非常小，从而保证运算节点具有很好的性能。主控节点不进行并行任务的计算，只完成资源监测、并行任务的调度映射以及执行 I/O、文件系统等非计算密集型操作，这样可充分利用多核系统的计算能力，并且系统具有较好的扩展性。

2.3.1 主控节点和运算节点

本文中多核系统中的节点分为主控节点和运算节点两类，主控节点和运算节点形成一种 Server-Client 结构。两类节点的硬件结构在2.1节中已有描述，每个节点上运行一个独立的、相互隔离的操作系统，主控节点上运行一个 Linux 操作系统，运算节点上执行着一个高度定制化的 uC/OS-II 操作系统。

主控节点的主要功能是通过运算节点操作系统检测多核系统全局资源变

化，在并行应用到来时根据系统资源分布及任务信息进行并行任务的调度和映射。此外，多核系统提供了并行计算能力，而 I/O 操作的性能取决于 I/O 速度，因此本系统中 I/O 操作全部由主控节点完成，而并行计算由运算节点完成，主控节点不参与并行计算。本文中的分布式多核系统作为用户/多核系统的接口，多核系统底层硬件的多核特性、网络互连、以及网络规模等性质对于用户透明，用户在执行并行应用时可以不去关心这些底层的硬件特性。在用户将并行应用提交给主控节点后，操作系统会自动进行并行调度，并根据调度结果将并行程序分发给各个运算节点；运算节点操作系统根据接收到的调度结果与并行应用程序建立、加载并完成并行任务的执行。主控节点和运算节点形成一种分层的、集中控制式的结构，运算节点操作系统可以看作是主控节点操作系统与运算节点的中间层，主控节点通过运算节点操作系统监测和管理运算节点资源，从而实现对整个多核系统的资源管理。当多核系统的节点数目在一定范围内变化时，这种变化可以对用户完全透明，不需要分布式操作系统做任何修改，对系统中原有节点也没有影响，从而保证了多核系统良好的扩展性。

在本文的多核系统中，并行程序的运行过程分为调度、分发和执行三步，首先由主控节点进行并行应用的调度和映射，然后主控节点将并行程序分发给运算节点，最后运算节点建立并行任务并执行。由于 MMPI 程序在执行时才读入映射结果以了解其他任务的映射情况，所以基于 MMPI 的并行编程与底层硬件无关，开发人员和用户都不需要了解硬件资源。主控节点操作系统为整个多核系统的资源提供了抽象，在本文多核系统上开发的并行程序与硬件关联性小，具有较好的可移植性。采用非均衡模式设计思想、将系统划分为主控节点和计算节点的实现方式，本文多核系统具有良好的扩展性和可移植性，同时解决了非均衡模式下全局资源管理和并行应用加载与执行的问题。

2.3.2 系统运行过程

本文中分布式多核操作系统主要解决的问题是为并行应用提供利用多核系统并行计算能力的途径。并行应用运行于多核系统上分为以下几步，首先是调度映射，即决定并行任务分别运行于哪些节点上，其次是并行任务的分发，并行任务在执行时运行于不同的节点上，需要将并行任务分发到不同的节点上之后才能执行并行任务的计算，最后是各个节点执行并行计算。这需要主控节点和运算节点协作完成并行应用的执行，分布式多核操作系统的运行过程如图2-2所示。

并行程序的调度映射由主控节点完成，对并行程序进行调度映射需要获得

系统资源分布信息以及并行任务的执行、通信信息。在本文的多核系统中，并行任务的相关信息由用户给出，而系统资源分布信息则由操作系统统计得到，图 2-2中的过程①和②即资源信息统计过程。其中过程①是运算节点操作系统统计所在节点的资源信息并将统计结果发送到主控节点，过程②是主控节点操作系统将接收的资源信息以统一的格式存储在专门用于存储统计信息的空间中，主控节点上用于存储系统资源信息的空间称为资源池。由于多核系统在整个运行过程中资源信息时刻都会发生变化，所以主控节点上的资源信息收集与

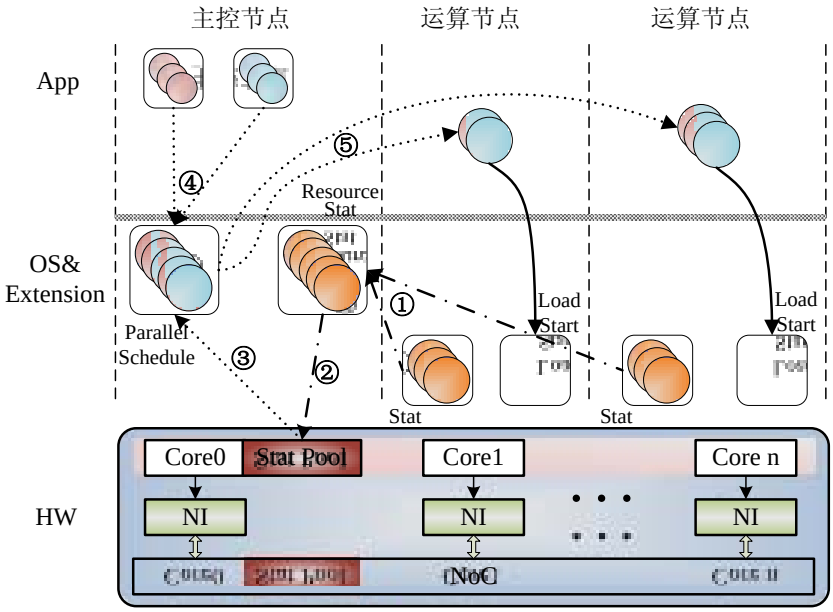


图2-2 分布式多核操作系统运行示意图

运算节点的资源信息统计模块需要在系统启动之后立即开始执行，并在系统运行的整个过程中循环检测系统资源。在 Linux 下资源统计模块以后台程序的形式运行；在运算节点上，资源信息统计模块是操作系统在开始执行任务之前建立的循环任务，并在系统开始调度任务后即开始执行。运算节点采用异步策略统计和发送资源信息，主控节点开辟多个进程接收运算节点发送的资源信息。

资源统计为并行应用的调度和映射提供了基础，图 2-2中的③④⑤三个过程是系统调度并行应用和分发并行任务的过程。调度并行应用需要了解并行任务信息及多核系统资源分布，其中过程③即主控节点并行调度模块获取有资源统计模块收集的系统实时资源信息，过程④即并行调度模块获取并行任务信息。在获取并行任务信息和系统资源分布后，调度模块根据调度算法进行并行应用的调度，然后为并行应用分配通信域，并将调度结果写入在 MMPI 程序执行时传递给并行应用的通信子（Communicator，也称作通信器或通信域）中。

在产生调度结果后，调度模块会根据这个调度结果将通信子及并行应用程序分发到分配了任务的计算节点上，由运算节点为并行任务分配资源并根据调度结果建立一个或多个任务并完成并行任务的执行。

2.3.3 并行应用执行过程

本文的非均衡系统是在实验室已有的多核仿真平台下设计的，并行程序的执行过程借鉴了许多已有的设计思路。本节首先阐述了在已有的多核仿真平台上 MMPI 程序执行过程，然后针对非均衡多核系统的不同之处描述了非均衡下并行应用执行过程的实现思路。

2.3.3.1 Linux 下 MMPI 程序执行过程

并行应用的执行需要并行应用程序和映射结果两部分内容。为了利用多核系统节点间的并行处理能力，并行应用程序会被分发到不同节点上，在各个节点上执行不同的并行任务，节点之间需要进行数据的传递以完成任务的通信和同步。因此并行任务需要了解其他任务的分布情况，这要求并行任务在执行之前获得映射结果。

已有多核仿真平台下节点操作系统 Linux 支持文件系统，并行应用程序与映射结果以文件的形式存放在文件系统中，通过在 shell 中调用 MMPI 程序来执行并行应用，同时将调度映射文件作为参数传递给并行应用程序。MMPI 程序在执行过程中首先会进行系统初始化，初始化过程由 MPI_Init 函数完成，称为基于映射文件的初始化过程。在系统初始化过程中，MMPI 程序读入映射文件，将映射结果存入全局通信子中。通信子为 MMPI 任务服务，存储着并行应用的总任务数，执行并行应用的节点数，各个任务的分配信息，当前节点的节点号、分配的任务数以及分配给当前节点任务的标识号。其中前一部分信息在映射文件中必须给定，与节点相关的信息是在系统初始化过程中由前一部分信息解析得到。通信子中存储着各个任务的映射信息，MMPI 任务在执行过程中通过查询通信子可获得与之通信的目标任务所在的节点信息，从而将任务之间的通信转换为节点间的通信并完成数据的传输。

在读入映射结果后，MPI_Init 函数会根据所有任务的分配信息解析出当前节点分配的任务数目及任务标识号。随后 MPI_Init 函数根据解析出的信息使用 fork 函数为并行任务创建子进程。在 Linux 下 fork 函数生成的子进程会复制父进程的代码段、数据段和堆栈段，各个子进程的代码段和数据段与父进程拥有相同的内容，因此各个子进程拥有私有的全局通信子。子进程会根据分配

给当前节点的任务序列将通信子中的当前任务号修改为该子进程需要执行的任务的标识号，之后各个子进程开始并行任务的执行。虽然执行的是相同的程序，但由于任务标识号不同各个子进程执行了 MMPI 程序中不同的部分，即子进程执行了并行应用中不同的任务，这就实现了并行任务的分配和执行。

2.3.3.2 非均衡系统下 MMPI 程序执行过程

在 Linux 下并行应用的执行采用的是静态映射策略，在执行时已经完成了调度映射，调度映射结果以文件的形式传递给 MMPI 程序。本文的非均衡系统中有两点不同：首先运算节点不支持文件系统，MMPI 程序和映射结果都不能以文件的形式存在；其次是支持动态映射，在 MMPI 程序执行时才根据系统资源分布进行调度和映射。因此本文的 MMPI 程序执行过程与已有的 Linux 操作系统下的执行过程不同。

主控节点并行应用调度模块为并行应用程序的执行提供了支持，并行程序的运行需要调用并行调度模块来完成，用户需要提供并行程序的任务信息、并行程序的二进制机器码文件和用于发送任务以及映射结果的 MMPI 映射文件。这里需要说明的是该映射文件用于主控节点和运算节点之间传输调度结果和并行程序，供并行调度模块使用，并且在网络规模确定后即可获得，与并行应用和映射结果无关。本文的非均衡系统中并行应用的执行分为映射调度、映射结果分发、程序分发、并行任务的建立执行等几个过程。并行应用的执行需要主控节点和运算节点协作完成，两种节点需要完成的功能各不相同，主控节点和运算节点必须实现的功能如图 2-3所示。

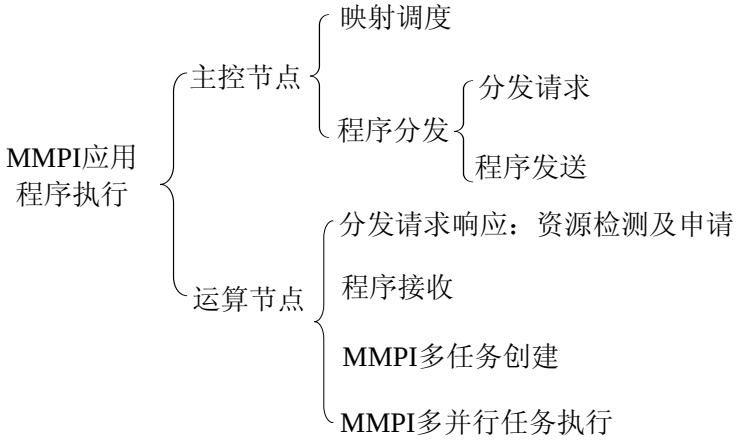


图2-3 支持并行程序执行的节点功能要求

主控节点上首先需要进行 MMPI 并行应用的任务调度，并将调度结果和所需资源发送给运算节点，这些信息实际上也是进行并行程序的发送请求。运算

节点接收到程序发送请求后会进行资源检查以确定能否满足并行应用资源要求，并将检测结果返回主控节点进行应答。由于主控节点是根据各个节点的资源信息进行调度的，所以一般情况下程序分发请求能够得到成功应答。主控节点在接收到成功应答后会将并行应用程序发送到所有分配了任务的运算节点。至此 MMPI 程序执行过程中主控节点的工作全部完成。

运算节点上会建立 MMPI 的任务建立模块，该任务从运算节点操作系统启动开始循环执行，接收主控节点发送的程序发送请求，根据分发请求申请程序空间及其他所需资源，如果能够获得资源则返回成功应答，否则返回失败应答。主控节点在接收到运算节点返回的成功应答后会将二进制可执行文件中的指令流发送到运算节点。运算节点接收指令流并将其存储到已申请的程序空间中，并根据接收分发请求中获得的任务映射信息来建立多个任务。在运算节点上需要解决的问题是如何建立多个 MMPI 任务。在 Linux 下 MMPI 多任务的支持是通过 fork 函数建立多个子进程实现，运算节点 uC/OS 系统仿照了 fork 函数复制了并行程序的数据段，为每个建立的并行任务提供了私有的全局通信子。随后运算节点会将任务映射结果和任务标识号通过参数传递给并行任务。uC/OS 下的 MMPI 程序的系统初始化过程是从 MMPI 应用程序的参数列表指向的结构体中获取任务映射信息以及当前节点号，通过获取的信息设置任务的私有通信子并完成任务的执行。

2.4 本章小结

本章在介绍多核硬件系统的基础上，讨论了三种多核操作系统设计模式，并指出非均衡模式为性能和编程提供了一个良好的折中，采用非均衡模式设计多核操作系统结合本文的硬件系统将具有良好的可扩展性、可移植性以及较高的性能。本文分布式多核操作系统设计的重点在于解决非均衡模式带来的全局资源管理和并行应用加载执行的问题。随后本章描述了应用非均衡模式设计的多核系统的结构，系统中节点分为主控节点和运算节点，形成一种分层结构实现对全局资源的管理。在这种分层结构下，系统的运行过程包括资源统计、并行应用的分发及执行几部分，其中并行应用的执行包括映射调度、映射结果分发、并行程序分发以及运算节点上并行任务的创建执行等几个过程。

第3章 主控节点操作系统

主控节点完成多核系统的 I/O 操作，监测系统全局资源的实时变化，作为用户/多核系统的接口，为用户提交的并行应用提供支持。主控节点操作系统选用了 Linux 操作系统，通过资源统计模块和并行调度模块来实现资源统计、并行应用的调度及分发。

3.1 主控节点操作系统概述

主控节点需要实现的功能决定了主控节点必须选用一个通用操作系统，大部分嵌入式操作系统一般不提供与用户交互的接口，并且支持的设备也有很大局限性。实验室已有的多核仿真平台节点上运行的是 Linux 操作系统，其能够满足主控节点操作系统的要求。Linux 是一个开源的操作系统，其源代码可免费获得并且可以任意修改；采用模块化设计，方便裁减，并且提供图形化的定制界面；支持多种文件系统，可以在主控节点运行可执行文件；大量的硬件和软件厂商支持 Linux，可以减少驱动程序的开发；最后，Linux 成熟稳定，适应性强，可适用于大型机、桌面级应用到嵌入式系统中等各种场合[43]。Linux 具有众多优点使得其被选为主控节点操作系统。

本文对主控节点运行的 Linux 系统进行了定制。操作系统内核包含了文件系统，利用 RAM Disk 技术加载文件系统，文件系统中仅包含一个精简的 BusyBox 以及用户应用程序。为缩小文件系统的尺寸，文件系统没有包含动态链接库，这要求运行的可执行程序必须使用静态编译。在 M5 仿真平台上，主控节点利用串口与主机进行交互，用户可以通过串口在 BusyBox 中输入指令，执行并行应用。本章在 Linux 下实现资源统计和调度并行应用的功能，资源统计模块要求在主控节点操作系统启动时开始执行，在 BusyBox 的启动脚本 rcS 中加入调用资源统计模块的指令。通常内核在初始化之后就会执行该脚本，在 rcS 脚本的最后加入资源统计模块的调用能够保证操作系统在启动后立即执行资源统计模块。执行并行应用时，并行调度模块根据实时资源信息进行并行任务的调度，并将并行程序发送到各个运算节点，完成并行程序的调度。

3.2 资源统计模块

资源统计模块实现主控节点监测和收集全局资源信息的功能。该模块在主

控节点操作系统启动之后立即开始执行，并在系统运行的整个过程中循环执行，接收运算节点发送的资源信息，将其格式化统一存储在主控节点开辟的资源池空间中。为了保证系统的可扩展性，资源统计模块的统计功能需要能够随着网络规模的变化而变化。通过读入网络配置文件，资源统计模块能够根据网络规模开辟资源池空间并检测网络中每一个运算节点的资源信息。资源统计模块的入口参数是 MMPI 映射文件名，但该文件并不需要由程序员提供，该文件在资源统计模块开始执行时并不存在，而是由资源统计模块在读入网络配置文件后生成。资源统计模块没有输出，其统计的资源信息存储在资源池中，并行调度模块在执行时从资源池中获取全局资源信息。资源统计模块的执行过程如图 3-1所示，主要包括资源池建立、运算节点信息统计以及资源统计模块和并行调度模块映射文件的生成等功能。

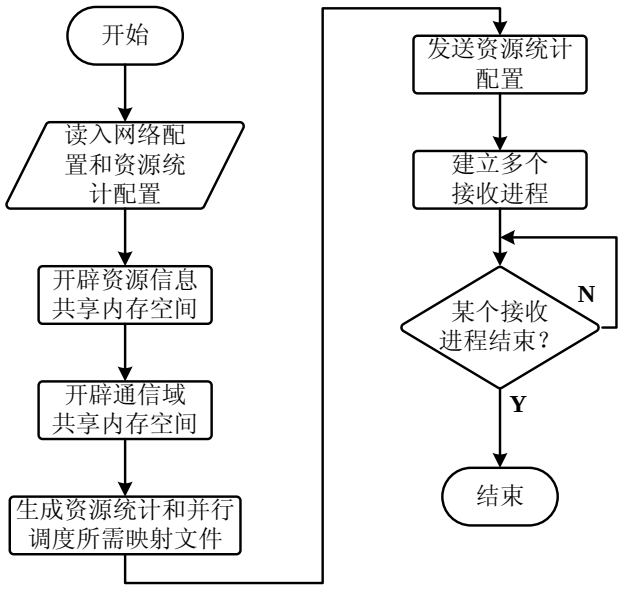


图3-1 资源统计模块流程图

3.2.1 资源池建立

资源统计模块的统计功能需要能够容忍系统规模的变化，该模块在启动时读入 NoC 网络配置文件和资源统计配置文件。NoC 网络配置文件描述了通信网络的规模，指定了主控节点的坐标，根据 NoC 网络配置文件可以确定网络中节点数和主控节点。资源统计配置文件决定了需要统计的信息的种类，根据该配置文件可以得到存储每个节点资源信息所需的空间。

并行调度模块会使用资源统计模块收集的实时资源信息，两者分属不同进程，这需要在进程间传递数据。Linux 下 IPC（Inter-Process Communication）

机制包括消息队列、信号量和共享内存等。消息队列使用的是链表式的队列，适用于传递数量不确定的数据；信号量则用于进程间同步；共享内存适用于大批量数据传输。此外，进程间还可以使用文件来完成数据的传递，但文件读写属于 I/O 操作，频繁的文件读写会影响程序性能。本文选用共享内存来在资源统计模块与并行调度模块之间共享资源信息。

资源统计模块在读入 NoC 网络配置文件和资源统计配置文件后即可确定网络节点数以及每个运算节点以及存储每个运算节点资源信息所需的空间大小，因此即可开辟一段大小固定的存储空间用于收集全局资源信息，该段空间即为资源池。

3.2.2 资源信息统计

资源统计的策略采用的是异步统计策略，即运算节点在资源发生变化时才会将新的资源信息发送给主控节点，因此运算节点发送资源信息的次数是不同的，某些节点发送的资源信息次数多，某些节点发送次数少。由于 MMPI 中的 MPI_Gather 函数的每次执行在接收到除根节点外其他所有节点的数据后才能结束，所以在实现上不能使用 MPI_Gather 来接收资源信息。资源统计模块为每一个运算节点创建一个接收进程，在每个接收进程中循环调用 MPI_Recv 接收资源信息。由于 MPI_Recv 可以阻塞并挂起进程，当没有资源信息到来时，接收进程就会阻塞执行并挂起。

多个进程并发执行，可以独立的接收各个运算节点发送的资源信息，接收进程在接收新的资源信息后会更新资源池中对应节点的数据。由于多个进程都需要对同一块共享内存空间进行写操作，所以需要在更新资源池信息时进行互斥，防止两个接收进程同时修改资源池中信息。此外，资源统计模块也会与并行调度模块同时访问资源池，为了防止资源统计模块在并行调度模块访问资源信息的过程中修改资源信息，也需要互斥两个模块对资源池的访问。资源统计模块的各个接收进程以及并行调度模块在访问资源池之前使用信号量申请函数来检查是否有其他进程已经获得了资源池的访问权，如果是则当前进程阻塞，否则当前进程可以访问资源池同时对资源池上锁，直到资源池访问结束，当前进程释放对资源池的占用。

除资源信息外，MMPI 程序使用的通信域的存储空间也由资源统计模块通过共享内存来开辟。通信域也是全局系统资源，其存储空间由资源统计模块开辟，由并行调度模块负责为并行应用分配，因此也须在资源统计模块中使用共享内存来存储通信域的分配和使用情况，供并行调度模块为并行应用分配未使

用的通信域值。

3.2.3 映射文件生成

资源统计模块在读入资源信息统计配置文件后会将配置结果发送给系统其它所有节点，各个接收进程会接收其它节点发送的资源信息，所以资源统计模块会与系统中运算节点上的资源统计任务进行 MMPI 通信。资源统计功能可以看作是一个 MMPI 并行应用，资源统计模块在执行时需要读入映射文件以了解其他资源统计任务的映射情况。

在本文的多核系统中，该映射文件不需要用户提供，而是由资源统计模块自动生成。映射文件说明任务的映射情况，其中设置了总的任务数目、节点数目、任务映射到节点的信息等。资源统计模块与运算节点资源统计任务之间的数据传输是确定的，包括资源统计模块广播发送的资源统计配置，以及资源统计模块创建的多个接收进程点对点接收运算节点发送的资源信息。对于资源统计这个应用，每个运算节点只与主控节点通信，两个运算节点之间没有任何数据传输，并且任务与节点是一一对应的，所以可以将任务号设置为每个节点的节点号，这样可以完成通信并且能够由资源统计模块自动生成映射文件。这个映射文件仅与网络规模相关，资源统计模块在读入 NoC 配置文件后即可生成该映射文件。在该映射文件中，任务数目与节点数目相同，每个节点上只有一个任务并且任务号与节点号相同。

资源统计模块作为资源统计这个 MMPI 应用在主控节点上的一个任务，在 BusyBox 中被调用时需要遵从 MMPI 程序执行的格式，也就是将第二个参数设置为映射文件名。虽然这个映射文件是由资源统计模块自动生成，但由于 MMPI 初始化过程中获取映射文件的过程固定，所以在 BusyBox 需要将映射文件名作为参数传递给资源调度模块。此外，并行调度模块也需要与运算节点通信，其通信特点与资源统计模块完全相同，因此资源统计模块产生的映射文件也可以提供给并行调度模块使用。本文多核系统在资源统计模块中生成该映射文件也是因为资源统计模块能够先于并行调度模块执行，并行调度模块执行时可以直接使用该映射文件。

3.3 并行调度模块

并行调度模块在用户执行并行应用时读取资源统计模块收集资源池中的全局资源信息，结合并行应用的任务信息应用调度算法将并行应用的任务分配到各个运算节点。在得出调度结果后，并行调度模块会将调度结果以及并行应用

的程序信息发送到分配了计算任务的运算节点，运算节点在检测资源后返回应答，并行调度模块根据应答信息决定是否向运算节点发送并行应用程序，并行调度模块执行过程如图 3-2所示。

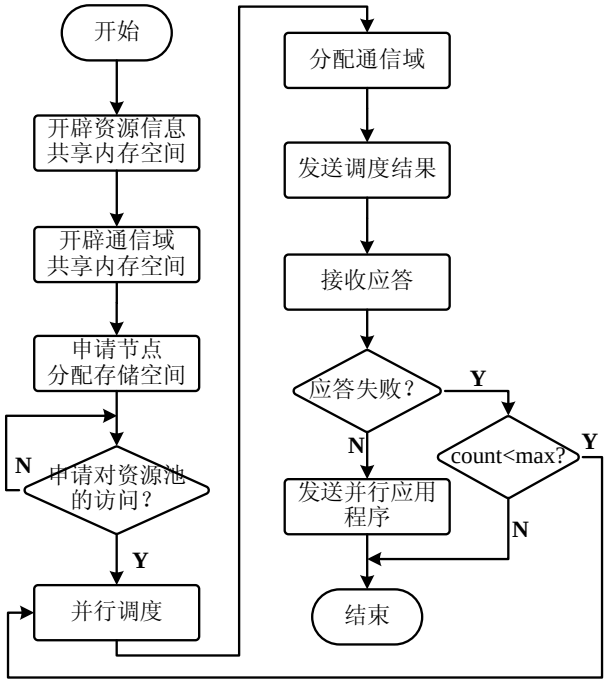


图3-2 并行调度模块流程图

并行调度模块的入口参数有三项：MMPI 映射文件、并行程序文件以及并行程序的任务描述文件。其中 MMPI 映射文件由资源统计模块根据网络规模生成；并行程序文件是并行调度模块发送到各个运算节点上有运算节点执行的二进制机器码文件；任务描述文件包含任务的计算量、任务通信以及执行先后等的描述，任务描述与系统资源信息是进行并行调度所需的，并行程序任务描述文件供调度算法使用。并行调度模块没有输出，动态调度算法在得出调度结果后调度模块会根据调度结果将并行程序分发出去执行。并行调度模块在并行程序执行时被调用，完成并行程序调度和发送后就会退出，而不是像资源统计模块那样循环执行。

3.3.1 通信域的分配

MPI 程序的通信域标识了一组通信进程，能够提供一个相对独立的通信区域。通信域提供了一种封装机制，允许各个应用有其自己的独立的通信域及进程计数方案，不同的消息在不同的通信域中进行传递，不同通信域中的消息互不干涉。MMPI 程序中使用通信域和通信的目标任务来惟一确定接收进程，本

文的多核系统对多 MMPI 应用提供了支持，不同的通信域可以区分不同并行应用相同节点对之间的通信，下面以 MPI_Send 函数为例来说明通信域的使用。MPI_Send 函数的参数列表中包括目的任务号、消息标志和通信域等几项内容，如图 3-3所示。这些内容会被封装到消息信封（envelop）中，消息信封是驱动层及通信硬件用来区分消息的单元。

通信域可以确定一组通信进程，本文的多核系统对并行编程提出约束，即不同的并行应用使用不同的通信域值，因此通过通信域可以区分不同并行应用任务之间的通信。在同一个并行应用内，源任务号和目标任务号可以区分不同任务之间的通信；对于同一并行应用相同任务对之间的通信可以进一步通过设置消息标志 tag 来区分，此时 MPI_Send 的参数中的 tag 仅须在两个任务间多次通信时进行设置。通过设置通信域、源和目的任务的任务号、消息标志，MMPI 保证任意时刻的消息具有唯一性，从而保证正确通信。

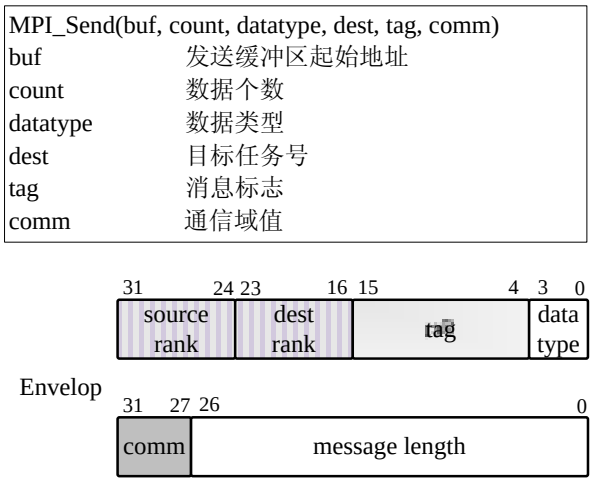


图3-3 MPI_Send 参数列表

在 MMPI 实现中通信域的值被封装到消息信封低 32 位中的高 5 位，如图3-3所示，通信域可取范围为 0 到 31。MPI 标准使用了三个特殊的通信域 MPI_COMM_NULL、MPI_COMM_SELF 和 MPI_COMM_WORLD，其值分别为 1、2 和 3。本文多核系统分配给资源统计模块的通信域的值为 4，并行调度模块的通信域的值为 5，因此普通并行应用的通信域的取值范围为 6 到 31。在本文的多核系统下并行应用执行的过程中，通信域的值并非由程序员指定，而是由并行调度模块在调度时进行分配，并在 MMPI 并行程序执行时通过传递给应用程序。并行调度模块在分发并行应用程序前分配通信域值，然后将通信域值作为发送请求的一个元素发送给运算节点，运算节点在创建并行任务时将通信域值作为参数传递给并行任务，由并行任务来读入创建进程传递的通信域值，这样由主控节点分发程序来保证并行应用程序中的通信域的值由并行调度

模块确定，并且不同并行应用中的通信域值不会相同。

分配通信域的值之后，调度模块将通信域作为通信子的一个成员，将任务调度结果和通信域的值写入通信子结构体中，并将该结构体分发到运算节点。通信域是一种全局系统资源，在整个系统运行过程中都需要记录使用情况，并行调度模块在执行完毕后会立即退出，因此由资源管理模块负责开辟通信域存储空间，3.2.2节中说明了通信域存储空间的开辟，通信域存储空间是一个包括 32 个字符元素的数组，下标代表对应的通信域的值，数组元素值为 0 代表对应的通信域值还未被使用，可以分配给并行应用，否则就说明该通信域值已被分配。并行调度模块在调度时会在这个数组中将查询到的最小的可分配的通信域值写入通信子并发送给各个运算节点。

3.3.2 运行时调度

运行时调度是并行调度模块核心的功能，调度映射需要获知资源分布和任务描述。并行调度模块的入口参数中任务描述文件包含了并行应用中各个任务的计算量、任务的通信关系等信息，而所需的资源分布信息由资源统计模块存储在资源池中。并行调度模块在并行应用执行时读入任务信息和资源信息，根据调度算法完成对并行应用的调度，调度结果以通信子结构体的形式给出。

资源池是资源统计模块开辟的共享内存空间，能够被两个模块访问到，其中资源统计模块只对资源池进行写操作，并行调度模块只进行读操作。为防止资源统计调度模块在调度模块调度并行应用时修改资源池中信息，两者使用信号量同步对资源池的访问。在调度模块执行时，资源统计模块已开辟资源池并建立了用于互斥访问资源池的信号量，调度模块只须申请信号量请求访问资源池即可。Linux 下的 IPC 机制为每个 IPC 实现都分配一个键值，不同的进程可以使用这个键值对同一个 IPC 实现进行访问。一般情况下这个键值可以使用 ftok 函数来产生，ftok 函数使用文件系统路径和一个数值来产生键值。在不同的模块中调用 ftok 函数时设置相同的路径和数值即可获得对应 IPC 实现的键值，通过这个键值调度模块能够访问资源统计模块建立的资源池和信号量。

运行时调度产生的调度结果指定了并行应用中任务分配到节点的情况。为了减少通信量，调度模块只向分配了计算任务的运算节点发送调度结果和并行应用程序。因此调度结果还应该包括一个数组来说明各个运算节点是否分配了并行任务，该数组称为任务分配表。显然，任务分配表的长度与网络中节点数相同。并行调度模块根据网络规模信息申请任务分配表空间，因此需要资源统计模块在读入网络规模文件时将网络规模信息存储在共享空间中。与通信域类

似，任务分配表的下标值表示节点号，数组元素值为 0 表示该节点没有分配任务，为 1 则表示该节点分配了任务。任务分配表在调度过程中配置，调度完成后调度结果发送给哪些运算节点由任务分配表配置结果决定。

3.3.3 动态调度算法

本文多核系统中的动态调度是并行应用执行时由主控节点并行调度模块完成，只调度一个并行应用中的任务。动态调度相对于静态调度的优势在于可以根据系统资源的实时变化分配并行任务，因此动态调度算法的输入必须包括实时资源信息以及任务信息。本文的动态调度算法的设计目标包括三个方面：减少算法自身执行开销，降低通信功耗和减少并行应用执行时间。图 3-4是动态调度的流程图。

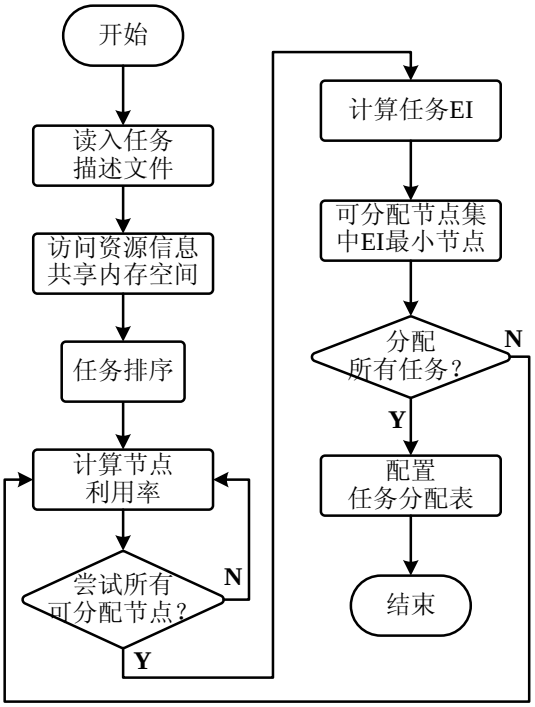


图3-4 动态调度算法流程图

首先需要清空任务分配表，任务分配表决定了并行调度模块是否向运算节点发送调度结果及并行应用程序。由于任务分配表中的初始随机值可能不为0，为了防止在分配失败的情况下调度模块仍然发送调度结果和并行程序，调度模块在调度之前将任务分配表中所有项的值设置为 0。在清空任务分配表后，根据从任务描述文件中读入的任务信息，调度模块会对并行应用中的任务做一个排序，排序的主要依据是任务的 t_level 值（Top Level）。t_level 指的是任务图的入口任务到当前任务的最长路径的长度，其中路径长度指的是路径上

各个并行任务执行时间之和。t_level 衡量了并行应用中各个任务之间执行的先后关系，显然 t_level 值越大，任务执行的时间越晚。根据 t_level 进行的任务排序的结果中靠前的任务需要尽早执行，因为这些任务的阻塞会延迟后续任务的执行，所以在调度时需要优先调度 t_level 值小的任务。调度模块首先调度入口任务，即 t_level 值为 0 的任务。本文的调度算法设计目标包括降低通信功耗和减少执行时间，在调度某个任务时，首先检测该任务能否分配成功，如果任务分配到任意节点上都不能满足调度周期的约束要求，分配失败；如果任务分配到某个节点执行可以满足调度周期要求，该节点为可分配节点。某个任务可以有多个可分配节点，在所有可分配节点中，存在一个使任务的通信开销最小的节点，该节点即任务所分配到的节点。通信开销与被调度任务的所有父任务相关，其值为该任务与其各个父任务相互通信的通信量与跳步数之积的和。根据通信开销最小这个优化目标可以在可分配节点序列中寻找到适合分配的唯一节点，调度模块循环执行直到所有任务分配完毕即完成了任务的调度。

所有任务中比较特殊的是入口任务，由于入口任务没有父任务，所以入口任务对所有可分配节点的通信开销都为 0，可以分配到任意一个可分配节点上。本文的动态调度算法将入口任务尽量放到 2D Mesh 网络的中间位置，如果有多个入口任务，那么入口任务放置到相隔尽量远的节点上。由于入口任务的特殊性，所以动态调度算法首先映射所有的入口任务，在找出各个入口任务的可分配节点集后，按照上述规则分配各个入口任务。在分配入口任务后，普通任务的可分配节点集的确定与入口任务相同，然后计算出普通任务的通信开销，根据通信开销确定普通任务映射的节点。

3.4 本章小结

本章描述了主控节点操作系统的设计与实现。首先说明了主控节点选用 Linux 操作系统的原因，以及在本文的非均衡多核系统中，Linux 系统能够支持 I/O 操作、文件系统等并为用户/计算机接口提供支持。之后本章着重介绍了在主控节点操作系统下监测多核系统全局资源和动态调度并行应用的实现。主控节点资源统计模块创建多个接收进程收集各个运算节点的实时资源信息，使用共享内存创建了存储资源信息的资源池。并行调度模块在并行应用执行时根据任务信息和资源池中节点资源信息进行运行时调度，在调度完成后向分配了计算任务的运算节点发送调度结果和并行应用程序。主控节点上的 Linux 操作系统通过资源统计模块和并行调度模块实现了对多核系统资源的监测和管理。

第4章 运算节点操作系统

运算节点完成节点资源信息的统计、更新与发送，并在并行应用执行时接收主控节点发送的调度结果，检查自身资源，接收并行程序，根据调度结果为并行应用建立多个任务，将调度结果作为参数传递给并行任务，完成并行任务的执行。运算节点操作系统为以上功能的实现提供支持，并且能够保证运算节点具有较高的性能。本文选用 uC/OS 作为运算节点操作系统，实现了 uC/OS 下的 MMPI 通信库，并提供了基于 MMPI 的并行任务加载和执行的支持。

4.1 运算节点操作系统概述

本文采用非均衡模式设计分布式操作系统，主控节点和运算节点上执行的操作系统是不同的，两者形成分层结构，运算节点操作系统作为中间层为主控节点操作系统监测和管理各个运算节点硬件资源提供支持。运算节点上的操作系统首先必须能够保证运算节点具有较高性能，这就要求运算节点操作系统的实现较为简单，能够提供基本的操作系统功能即可。其次运算节点操作系统需要对节点资源监测和发送提供支持。最后运算节点还要能够支持 MMPI 应用的加载执行。本文选用 uC/OS 作为运算节点操作系统，图 4-1是定制后的 uC/OS 操作系统结构示意图，操作系统包括已有的 uC/OS 内核和功能扩展层。

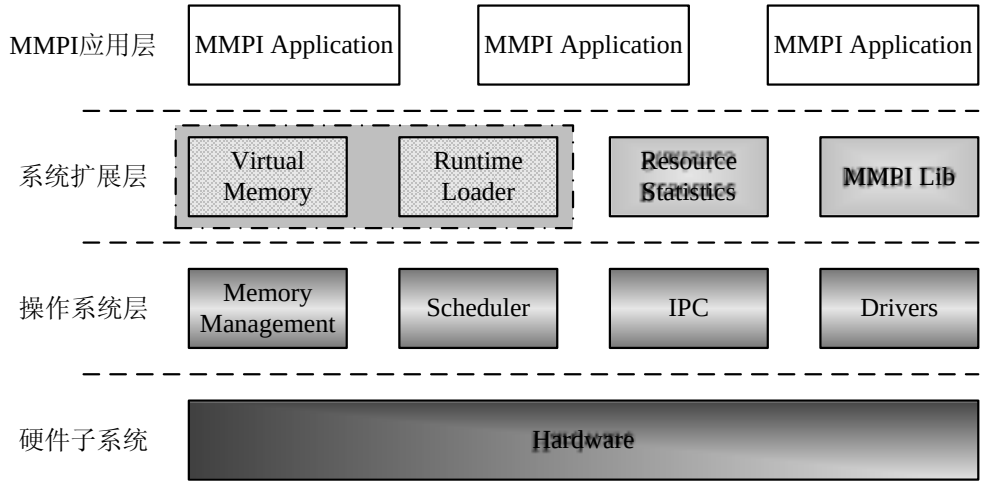


图4-1 运算节点 uC/OS 操作系统结构示意图

uC/OS 是一个支持多任务的嵌入式实时操作系统，构思巧妙、结构精炼，具备操作系统所需的基本功能。uC/OS 操作系统绝大部分代码用 C 语言编写，

与底层硬件无关，只有少部分与处理器密切相关的代码是用汇编语言编写，因此只需做很少的工作就可以移植到 ARCA3 处理器上[44]。uC/OS 提供的功能虽然有限，但能够满足运算节点需要，并且由于实现简单因而操作系统开销相对于 Linux 操作系统减少了很多，可以保证运算节点在执行并行应用时具有较高性能。扩展层是为了支持多核 MMPI 并行应用而提供的扩展功能，包括简单的虚拟内存分页机制、资源分配、信息统计和信息发送、运行时程序加载以及 MMPI 通信消息库。扩展层中的 MMPI Lib 为节点间通信提供了支持，资源统计则为主控节点提供了当前节点的实时资源信息，虚拟内存分页机制和运行时加载为 MMPI 任务的加载执行提供保证。

图 4-2是运算节点 uC/OS 操作系统启动过程示意图，在完成 uC/OS 的移植后，需要在 uC/OS 上实现资源统计和并行应用的执行两项功能。系统启动后首先完成初始化工作，初始化过程主要是完成初始化操作系统的全局变量、管理队列等，并配置运算节点的硬件模块。运算节点只包含了构建最小系统所需的模块，包括 ARCA3 CPU、片内存储器、网络接口、Timer 和中断控制器等。系统初始化后建立四个任务，分别是用于 DMA_NI 接收消息的两个中断服务任务 DMA_NI_MISS_Task 和 DMA_NI_RCV_Task，用于接收主控节点发送的映射结果、并行应用程序并建立并行任务的 MMPI_Create，统计并发送节

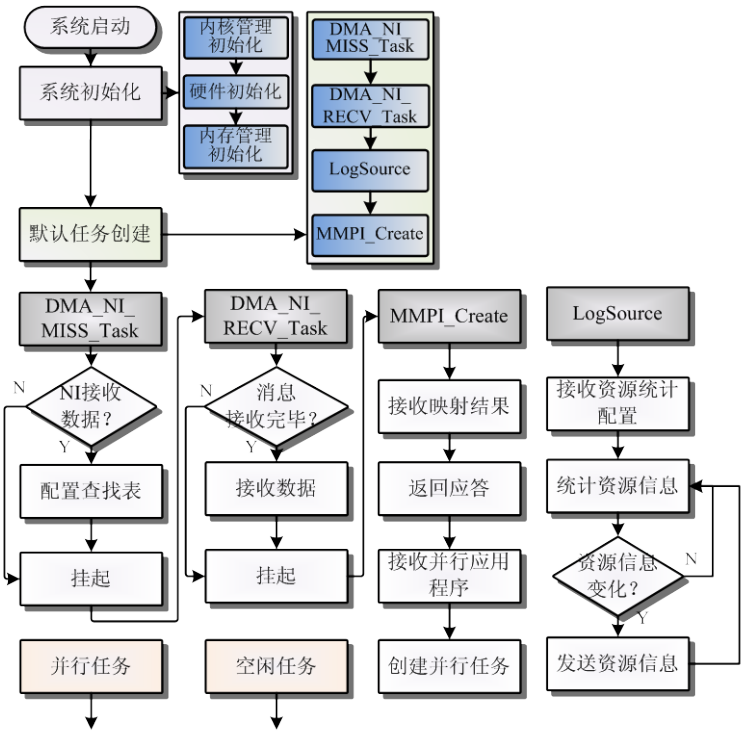


图4-2 uC/OS 操作系统启动过程示意图

点资源信息的资源统计任务。这四个任务的优先级按上文所述顺序从高至低，中断要求能够迅速执行，因此两个中断服务进程在四个任务中具有最高的优先级。运算节点上统计的资源信息通常只是在建立并行任务和并行任务运行结束时变化，因此应当在 MMPI_Create 执行后统计节点资源，所以 MMPI_Create 的优先级比资源统计任务的优先级更高。

系统在建立任务后调用 OSStart 开始调度任务执行，首先是两个中断服务进程的执行，一般情况下系统启动时各个节点之间没有通信，两个中断进程在检查 NI 的状态后会释放对 CPU 的占用。MMPI_Create 会接收主控节点发送的调度结果与并行应用程序，由于 MPI_Recv 在没有接收到数据时会阻塞程序执行，因此 MMPI_Create 会在等待接收时挂起。资源统计任务会等待接收主控节点发送的资源统计配置，与 MMPI_Create 相同，也会在等待接收时挂起。

4.2 uC/OS 移植

运算节点上的 uC/OS 操作系统首先要完成基本的任务调度、中断处理、内存管理等基本功能。uC/OS 的大部分功能实现与底层硬件无关，只有少部分使用汇编实现的功能，例如设置异常向量表、任务切换、异常处理等需要使用 ARCA3 的汇编指令重新实现，图 4-3是 uC/OS 操作系统的结构图。本节实现了 ARCA3 处理器上 uC/OS 的移植，包括任务创建和切换、中断处理、uC/OS 下 DMA_NI 驱动开发以及内存管理几个方面的内容。

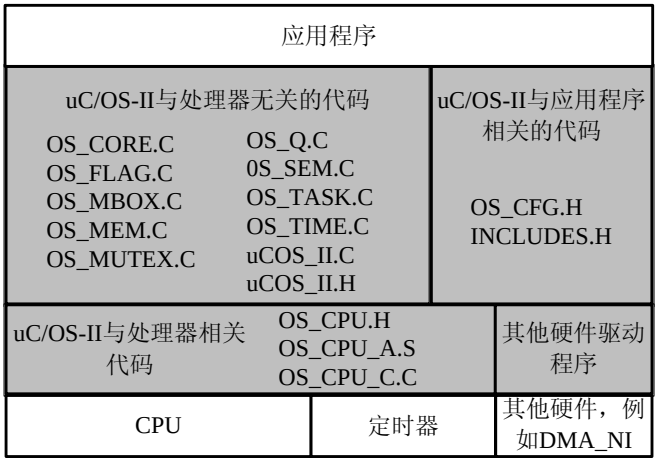


图4-3 uC/OS-II 体系结构

4.2.1 任务建立与切换

uC/OS 中的任务据有一个唯一的优先级，系统中可用优先级数目有限，总计 64 个优先级别。任务的优先级越高，其优先级别的数值越小。uC/OS 是一

个抢占式内核，当有多个就绪任务等待调度时，优先级最高的任务总能获得CPU 的使用权。uC/OS 操作系统在初始化过程中会创建一个空闲任务，当系统中没有其他任务执行时，系统就会运行空闲任务，空闲任务被赋予了最低的优先级，一旦有其他任务处于就绪状态，空闲任务就会被阻塞。除空闲任务外，操作系统还会根据配置创建一个统计任务，该任务的优先级仅比空闲任务高，当系统中有其他任务就绪时统计任务和空闲任务都会被阻塞，如果 CPU 没有其他任务执行，那么统计任务可以被调度执行，此时可以认为 CPU 处于空闲状态，资源统计任务根据自身执行所占的时间比例来计算 CPU 利用率。

uC/OS 中的任务与进程的概念类似，通过任务建立函数 OSTaskCreate 来创建新的任务，由于 uC/OS 任务的创建过程与 MMPI 的并行任务创建有关，所以本节将简要介绍其过程。OSTaskCreate 函数的入口参数包括任务执行代码存储地址、任务的参数、堆栈指针以及任务的优先级。其中执行代码存储地址指明了所创建的任务的二进制机器码存储在内存中的位置；任务的参数可以传递一些在任务执行时需要使用的变量值；栈空间是程序运行时由操作系统负责分配的空间区域，通常用来保存局部变量、函数参数、任务切换时的现场数据等；优先级既是任务的 ID，又决定了任务执行的先后顺序，在任务创建时必须指定并且不能使用已分配的优先级。uC/OS 在任务创建过程中将执行代码的存储地址、任务参数保存到分配的栈空间中，当任务执行时通过现场切换将栈中的数据载入 CPU 的寄存器中，程序即可执行并读取传递的任务参数。

uC/OS 中的调度工作可以分成两部分，首先是在任务就绪表中查找到优先级别最高的任务；其次是任务的切换。调度过程涉及到当前正在执行的任务和即将执行的任务。为了保证当前任务再次被调度时能够继续执行，并且在恢复时能够“无缝”的在中断点继续执行，需要把当前任务的断点现场数据进行保存。断点现场保存需要确定保存哪些断点数据以及这些数据保存到什么地方，本文将断点数据保存到当前任务的栈中，恢复时从当前任务的栈顶读出现场数据。对于断点数据的内容，为了保证任务恢复执行时与中止之前各种状态完全相同，需要保存 CPU 的控制寄存器和通用寄存器。

ARCA3 CPU 有 5 个控制寄存器和 32 个通用寄存器。控制寄存器包括 SR、ESR、EPC、DSR 和 DPC，其中 SR 是 CPU 的状态寄存器，EPC 和 ESR 用于在发生非调试异常时保存 PC 和 SR，异常发生时硬件会自动将 PC 值和 SR 值保存到这两个寄存器中。ARCA3 CPU 的 PC 寄存器不可访问，对程序员来说透明，所以说在保存现场时可以使用 EPC 来获得 PC 的值。DSR 和 DPC 用来在发生调试异常时保存 SR 和 PC，在调试异常中返回使用这两个寄存器

来恢复 SR 和 PC 寄存器的值。本文使用 trap 指令触发异常来保存任务现场，由于非调试异常发生时 PC 的值会被写入 EPC 寄存器中，在执行 trap 指令后 EPC 中保存了下一条指令的地址，在恢复时只须将保存的 EPC 的值载入到 PC 中即可恢复当前任务的执行。这与 ARCA Linux 操作系统的实现不同，Linux 在现场切换之后的语句处设置了一个特殊的标号，在保存现场时保存这个标号的地址值，返回时通过跳转到这个标号值实现了程序的继续执行，未采用该思路的原因在于 uC/OS 系统在中断退出函数 OSIntExit 中也进行了一次任务调度，为了实现普通任务调度的现场保护能够用于中断调度，故采用了 trap 指令来实现现场保护。在进行现场保护时，现场数据都是保存到任务的栈中，恢复时也是从栈中载入，因此在系统中运行多个任务时只要将堆栈指针由当前任务的栈指针替换为即将执行任务的栈指针，那么就可以说处理器已经完成任务的切换。这就是说任务的切换的实质就是断点数据的切换，而断点数据的切换就是栈指针的切换，栈的使用在任务的切换中至关重要。

任务在创建后的第一次调度执行也可以看作一次断点恢复，只是这次任务的“中止”发生在任务执行前。为保证调度模块操作的统一性，任务创建时通过堆栈初始化保证栈中存储的内容与任务切换保存的现场的一致性。ARCA3 CPU 的寄存器使用规则即二进制编程接口（Application Binary Interface, ABI）规定通用寄存器中的 r1 存储栈指针，r2~r7 在过程调用时用来传递参数，r18 用作链接指针，存储过程调用后返回的地址。在堆栈初始化时，存储 r1、r2 和 r18 的单元需要分别设置为任务创建函数传递的栈指针、任务的参数，以及任务退出函数的指针。

最后，由于栈在程序执行过程有着非常重要的作用，栈空间的大小对程序执行和系统性能有很大影响，设置过大会浪费存储空间，过小又会可能导致栈溢出。栈溢出是系统运行中极其严重的问题，为了解决栈空间设置的问题，本文在设计过程使用了一种测试方法，在分配时将栈空间填充为特殊的数值（例如 0x44bb55aa），在系统运行一段时间后查看栈空间，如果大部分空间存储的仍然是这个数值，那么栈的空间分配过大，反之如果没有或只有很少部分空间仍存储该值，那么栈空间分配过小，需要适当扩充。

4.2.2 中断处理

中断的处理过程与硬件体系结构相关，本文多核系统中的节点结构的设计参照了 ARCA3 GT3000 系统，GT3000 系统中的处理流程如图 4-4所示。在外部中断发生后，CPU 硬件保存状态寄存器并关闭中断，然后根据设置的异常

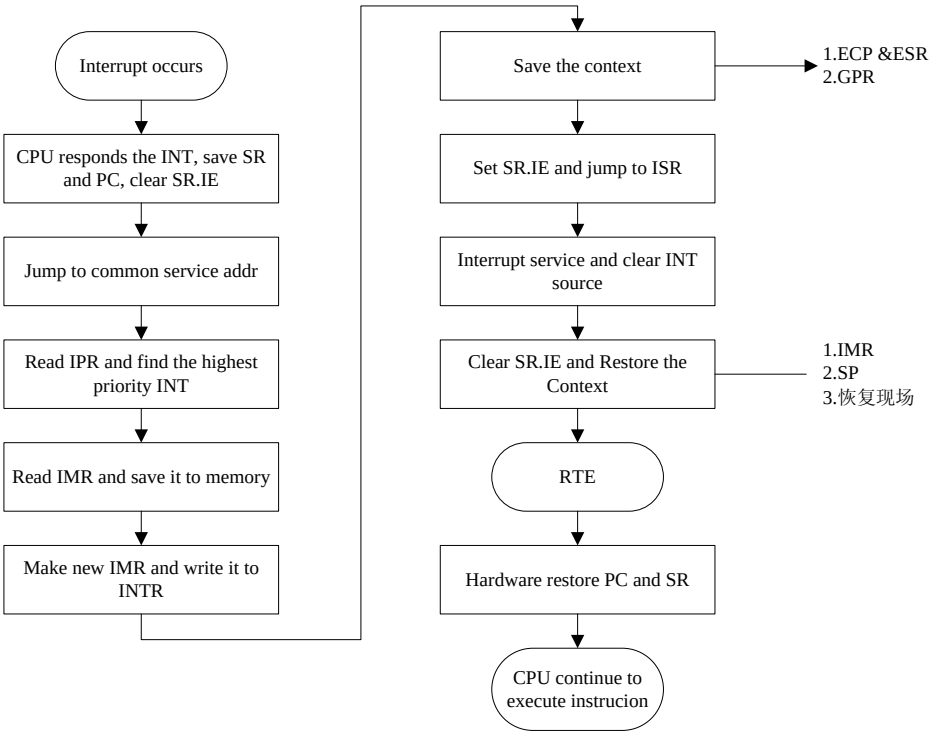


图4-4 GT3000 中断处理流程

向量表跳转到通用中断处理程序。中断处理程序首先读取中断控制器相关寄存器的值获取中断源，找到其中优先级最高的中断源。在寻找到最高优先级的中断源后重新设置中断控制器中的 IMR（Interrupt Mask Register）寄存器来屏蔽低优先级的中断。需要注意的需要将当前优先级最高的中断源也屏蔽掉，否则该中断源会持续有效，CPU 会一直被“新的”中断打断程序的执行并循环进入中断处理程序而无法完成任何中断处理。在重新设置 IMR 后，除非有更高级别的中断产生，否则 CPU 会先去执行当前中断的处理函数。

由于修改了 IMR 寄存器，因此中断处理程序首先将 IMR 保存到堆栈中，然后保存被中断执行的任务现场。本文运算节点的中断处理实现与该流程不同，主要体现在现场的保护次序。GT3000 系统首先保存 IMR，然后保存 CPU 的控制寄存器和通用寄存器。在运算节点上，uC/OS 操作系统在中断结束后可切换到其他任务，而不是必须返回到被中断执行的任务，这就要求任务切换和中断切换具有统一的现场保护，即任务调度时被打断执行的任务可以在中断中被调度继续执行，反之在中断中被打断执行的任务也可以通过任务切换来继续执行，因此中断现场保护和普通切换现场保护必须一致。而中断处理流程中保存的 IMR 寄存器在任务调度时没有保存，所以进入中断处理后首先保存 CPU 的控制寄存器和状态寄存器，然后保存 IMR，完成中断服务后在进行任务调度

和现场切换之前恢复 IMR 寄存器，这样被中断任务的栈中不会保存 IMR 寄存器，如果中断级调度切换到新的任务就保存当前任务的堆栈指针，此时被中断任务的现场保护完成，保证了任务切换和现场切换的现场保存的一致性。

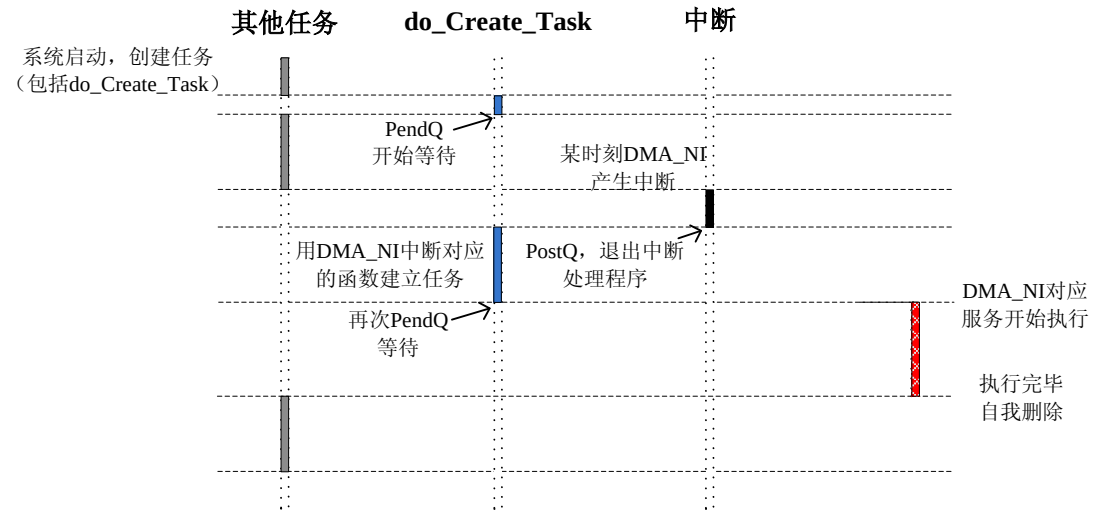
4.2.3 DMA_NI 驱动开发

外设驱动程序是操作系统内核和硬件之间的接口，编写驱动程序模块应满足以下主要功能：①对设备初始化；②把数据从内核传送到硬件和从硬件读取数据；③读取应用程序传送给设备的数据和回送应用程序请求的数据。uC/OS 中没有统一的设备驱动接口，对硬件的操作和控制通过函数来完成。

DMA_NI 实现的功能是节点间数据的发送和接收[45]。由于 uC/OS 始终处于内核态，发送端不需要在用户空间和系统空间之间拷贝数据，可以配置 DMA 直接将数据从存储器发送到网络接口的缓冲器中，这种传输方式不需要 CPU 的参与因而减少了拷贝开销。DMA_NI 某时刻只能被一个进程使用，uC/OS 提供了信号量来同步多个进程对 DMA_NI 的使用。对于接收端，DMA_NI 在接收到消息的第一个数据包后会发出缺失中断，CPU 在缺失中断的处理程序中配置 NI 中的查找表，配置完成后 DMA_NI 根据查找表中的地址配置 DMA 将数据从网络接口的 FIFO 中写入到本地存储器，消息接收完毕后 DMA_NI 发出接收中断通知 CPU 接收消息。在接收端 CPU 与网络接口的交互开销是影响通信性能的关键[46, 47]，因此中断处理的策略是设计 DMA_NI 驱动的重点。

为互斥访问网络接口，中断处理过程需要申请对 DMA_NI 的使用，请求 DMA_NI 信号量，当任务无法获得信号量时会挂起等待直到获得对 DMA_NI 的使用权。但是中断要求迅速执行完毕，在中断不能因申请资源而将中断处理程序挂起，所以 uC/OS 系统下的信号申请量函数会检查是否处于中断状态，如果是那么立即退出检查并且中断处理程序也不会被阻塞。本文设计了接收端的中断服务启动策略，这种策略使用了 Linux 中断服务前半部和后半部的概念，前半部处于中断状态只完成很少的、必须完成的工作，然后把其他处理放在后半部的普通任务中完成，在普通任务中使用信号量申请对 DMA_NI 的访问。本文在设计过程中使用了两种中断服务启动策略，下面分别予以介绍。

第一种策略是在中断中创建普通任务来完成对应的处理，即前半部的工作是创建任务，后半部完成对应的处理。uC/OS 操作系统下的程序在处于中断状态时不能申请信号量，也不能创建新的任务。本设计在系统启动时建立一个任务 do_Creat_Task，用于创建执行中断处理功能的任务，如图 4-5所示。



do_Creat_Task 循环执行，不停地检查一个队列，根据从队列中取出的值来创建缺失服务或接收服务任务。在创建任务后它再次检查消息队列，直到消息队列空为止。这样为每次 NI 中断创建一个普通任务，在普通任务中进行信号量的申请，并完成所需的处理。这种设计策略下中断的前半部完成的工作是根据中断的种类向队列中写入不同的值，后半部完成任务创建及中断服务。但该种策略的弊端在于从硬件产生中断到开始执行中断服务的延迟很大。这个延迟称为中断响应，包括向队列中写入值、唤醒因申请消息队列而睡眠的 do_Creat_Task、创建中断服务任务等各个部分的开销，期间还包括多次任务调度、任务切换以及现场的保护，这段开销是 Linux 下完成相同中断处理的所有开销的 20 多倍，所以该策略不可取。

第二种策略仍然借鉴了前半部和后半部的设计思路，在系统启动时分别针对缺失服务和接收服务创建了两个普通任务。这两个任务可以进行信号量的申请操作，循环执行并且在每次完成相应的处理后都会自动挂起。此外本文修改了信号量申请函数，使其 uC/OS 操作系统下的程序能够在中断状态下使用信号量，但是不会因无法获取资源而阻塞中断处理程序的执行，而只是通过返回一个非零值来说明无法申请到所需资源。此时 DMA_NI 的中断服务程序就唤醒相应的普通任务，在这些普通任务中再次申请信号量并完成相对应的操作。第二种策略下，如果能够在中断中申请到资源，那么前半部完成了所有操作，没有后半部；如果未申请到资源，那么前半部是唤醒对应的服务任务，后半部是服务任务执行响应的操作。该策略在无法获得所需资源的情况下减少了创建任务以及任务切换和调度的开销，可以大大减少中断响应延迟；在可以获得资

源的情况下能够进一步减小交互开销。该策略处理过程如图 4-6所示。

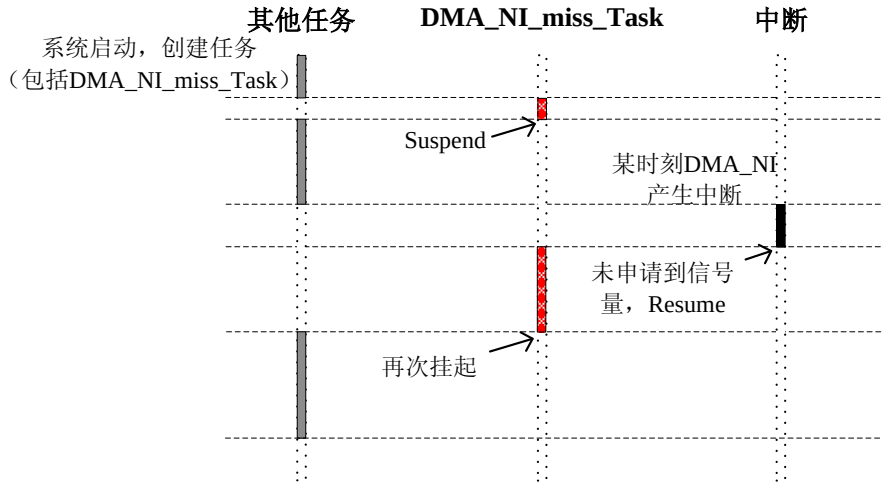


图4-6 DMA_NI 开启中断服务的另一种策略

以上两种设计策略都需要建立有专门目的的任务，它们的优先级设置需要考虑中断处理的特点。例如，策略二中的中断处理可能会在两个普通任务中完成，那么在中断服务程序中完成唤醒操作后，要等到进行任务切换和调度时才能切换到进行中断服务的任务来执行，如果系统中有很多其他的高优先级的任务，那么完成中断服务的任务的执行就会大大往后推迟。这部分处理工作就放在中断处理程序中，但其处理也很急迫，所以将这两个进行 DMA_NI 中断处理的普通任务的优先级别设置的非常高。

4.2.4 内存管理

uC/OS 提供了简单的内存管理，可以在需要时获得一些存储空间，并在使用完毕之后释放获得的临时空间。要动态使用内存空间，需要在内存中划分出可进行动态分配的区域，uC/OS 将这个区域称为内存分区，并使用内存控制块来完成对内存分区的划分和管理。uC/OS 的内存管理有很大的局限性，内存块大小固定，在分配的时候只能分配、回收一个块，因此可动态分配的空间大小是固定的，本文拓展了 uC/OS 内存管理的功能。

uC/OS 通过内存控制块管理内存分区。在创建内存分区时，需要将开辟出来的一段连续空间传递给内存控制块，内存控制块中的 OSMemAddr 就是指向这段空间的起始地址，创建内存分区的工作就是把一段完整的、连续的内存空间拆分成 OSMemNBlks 个内存块，这些内存块的大小完全相同，其中的 OSMemFreeList 是实现内存块分配和回收的关键变量。uC/OS 下创建内存分区所作的工作主要是使用 OSMemFreeList 来将所有内存块的连接成一个链表。

为了实现多个块的分配和回收，需要存储动态分配的空间的信息，包括该空间所属的内存分区以及该空间的大小，这个大小以内存块数目来衡量。这些信息用于内存块回收， uC/OS 已有的内存管理在回收内存块时不需要考虑顺序，将内存块插入到 OSMemFreeList 所指向的队列中即可。修改后的内存管理在分配时需要在整个队列中找到多个连续内存块，由于内存块申请和释放的顺序可能不一致，因此就会出现碎片问题，那么整个内存分区中所需的空间可能足够，但是连续的空间却可能并不满足要求。在申请时就需要注意检查获得的内存块是否连续，实现思路如图 4-7 所示。在分配时检查 OSMemFreeList 指

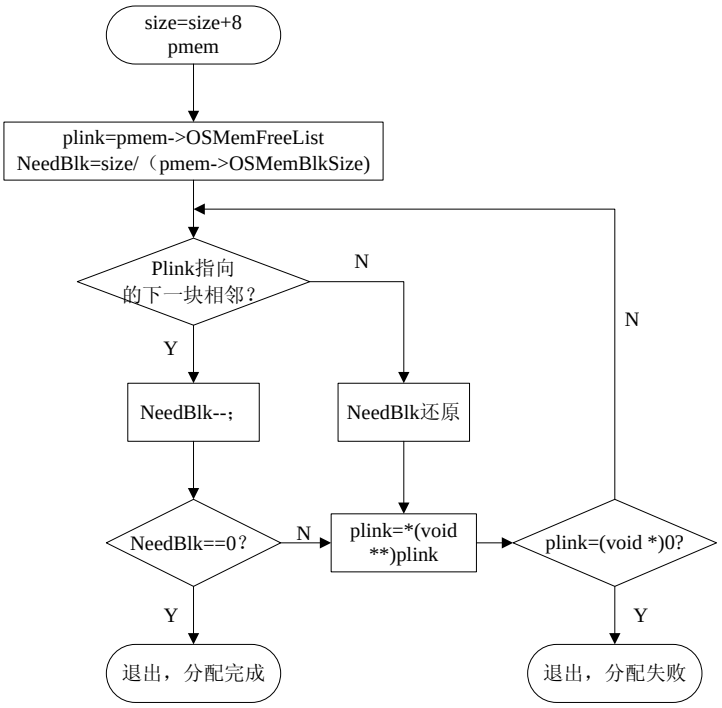


图4-7 多个内存块分配的连续块检测过程

向的链表中两个块是不是相邻的块，如果是则获取内存块并继续检测直到获得足够的内存块，否则重新检测直到完成分配或检查完所有的块，如果没有足够的连续的块，则分配失败。但这种分配思路会导致新的问题，即不再具有确定的执行时间，并且执行时间完全不可估计。当所需的内存块的数目不同时，分配过程中由于查找的块数目不同，因此执行时间不同；即使所需的内存块数目相同，由于可用块的分布情况不同，两次申请所需的时间也可能不同。在多个块的分配过程中，除了检查多个内存块的连续性外，还须在分配结果后使用 OSMemFreeList 将未分配的块连接起来。该问题的描述如图 4-8所示，蓝色块是已分配的内存块，新的内存申请需要分配两个内存块，OSMemFreeList 指向第一个内存块，但分配的是第三个和第四个内存块，此时第一个块所指向的下

个块由第三个块变为第七个块，因此第一个块中的链接地址需要修改。

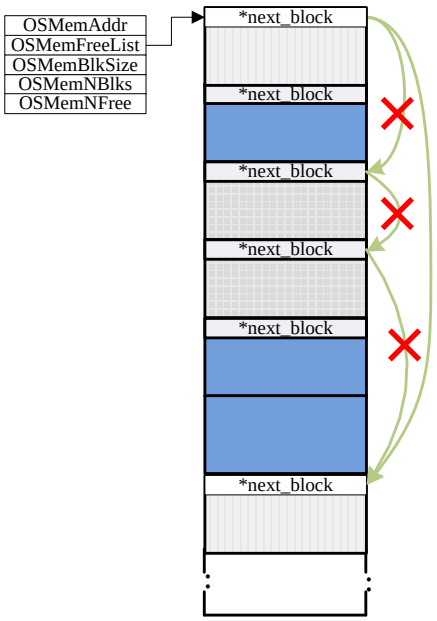


图4-8 多块内存申请时链接的修改

在多个块的释放过程中，需要将作为一个整体的内存空间拆分为多个内存块并把这些内存块插入到所属内存分区使用 OSMemFreeList 连接的队列中，为了确保再次申请时物理上相邻的内存块在内存块队列中也是相邻的，要求将释放的空间插入到队列中合适的位置。基于以上要求，在释放内存空间时需要获知内存空间所属的内存分区以及该块内存空间的大小。本文在 uC/OS 系统中创建了一个用于存储使用动态内存申请得到的内存空间信息的链表。信息中包含管理内存空间的内存控制块地址和分配的内存块数目，通过内存控制块可知该块空间中每个块的大小并可以在释放时获得 OSMemFreeList 的指向，而根据块大小和内存块数目可以计算该块存储空间大小。内存块数目也用于在释放时将多个内存块链接起来，由于在分配时保证了物理地址的连续性，所以只要在内存块队列中找到了内存空间起始块的插入位置，就找到了整个释放空间的插入位置，图 4-9是修改后的内存管理机制下多个内存块的释放过程。

4.3 系统扩展层

uC/OS 系统扩展层提供了虚拟内存分页机制、MMPI 任务加载、资源统计和 MMPI 库等的支持。其中 MMPI 库是节点之间通信的基础，uC/OS 下的消息库实现了基本的点对点通信功能，支持组通信中的广播通信和规约通信。资源统计功能可以根据主控节点发送的资源统计配置收集调度所需的信息并发送

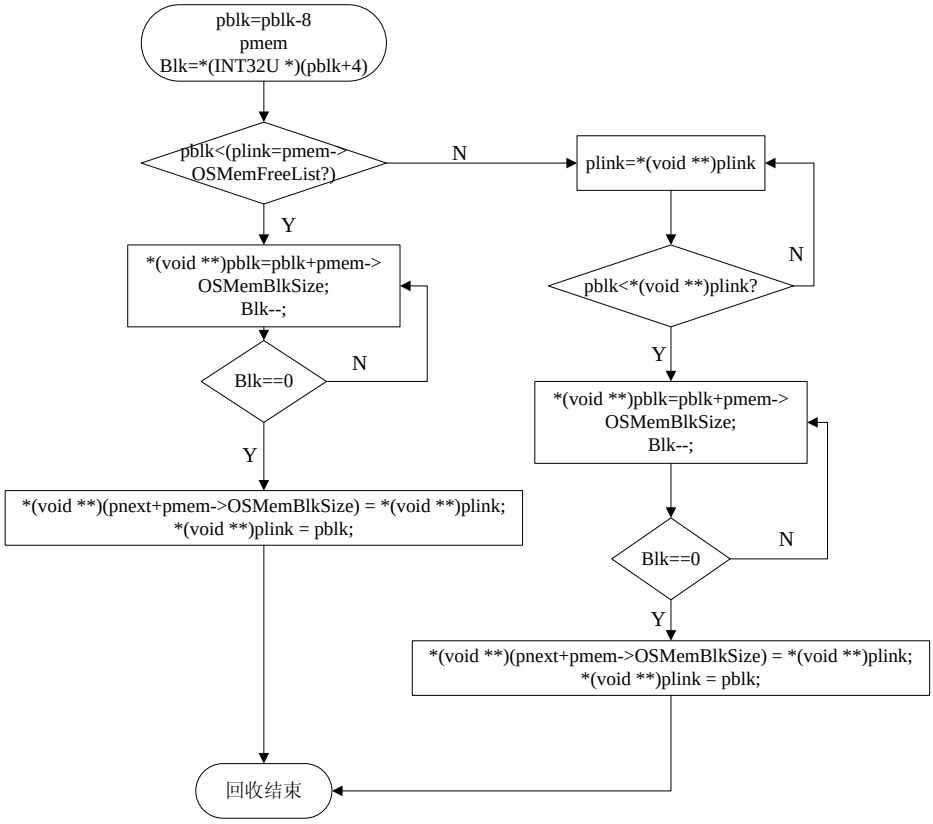


图4-9 多个内存块回收过程

给主控节点。MMPI 任务加载能够在运算节点接收到主控节点发送的并行应用程序后根据调度结果建立一个或多个 MMPI 任务，通过复制程序的二进制机器码来达到复制数据段的目的，分页机制则使得多个 MMPI 任务能够通过相同的虚拟地址访问到对应不同物理地址的数据段，保证了 MMPI 任务的正常执行。uC/OS 系统扩展层为本文多核系统执行 MMPI 并行应用奠定了基础。

4.3.1 资源统计

应用程序不会在使用了某项资源时向操作系统报告资源消耗，因此操作系统不能即时获知各项资源的变化，只能间隔固定时间检查资源信息。当某项资源信息发生变化时，资源统计任务会重新统计所需资源信息并将最新的信息发送给主控节点，主控节点在接收到资源信息后更新资源池。运算节点这种在资源信息发生变化时才发送资源信息的策略称为异步发送策略。资源信息的传输与并行应用间的数据传输均须占用网络带宽，运算节点采用异步策略统计和发送资源信息有助于减少对系统通信资源的占用，提高系统资源利用效率。

资源统计任务在收集资源前需要接收主控节点发送过来的资源统计配置，并根据配置来统计所需的信息。首先需要解决的问题是如何根据配置只统计所

需信息，在程序运行中不能改变数据结构，本文在存储资源信息的结构体中包含了所有的信息项，其中个信息项所占的空间大小相同，均为 4 个字节。根据配置信息，运算节点可以计算存储所需统计资源信息的空间大小，并且不统计的项的空间为后续统计的资源项的数据覆盖。在发送资源信息时，只发送资源信息结构体中需要发送的空间的数据，这样可以实现统计所需信息。

其次是主控节点资源统计模块和运算节点资源统计任务的通信问题，两者使用 MMPI 库进行通信，而 MMPI 程序在执行时需要了解其他任务的映射关系。为了保证系统的可扩展性，主控节点的资源统计模块在读入 NoC 配置文件后才建立相应数目的接收进程，该接收进程与 uC/OS 系统中的资源统计任务之间相互通信。资源统计的模块和任务可以认为是同一个 MMPI 应用中不同的任务，并且在各个运算节点中执行的任务相同。对于资源统计以及程序分发，运算节点之间不需要通信，只有运算节点和主控节点之间的数据传输，所以在统计任务可以认为在这个通信域内只有自身和主控节点之间的通信。3.2.3 节中论述了资源统计模块执行时所需的映射文件的生成，而 uC/OS 操作系统不支持文件系统，运算节点无法通过文件传递映射结果。本文在 uC/OS 系统中显式地初始化通信子。uC/OS 系统内核也相当于一个 MMPI 应用程序，在系统初始化时修改 uC/OS 全局通信子。本文多核系统中主控节点资源统计模块属于 0 号任务，资源统计任务的任务号与运算节点的节点号相同。

在初始化设置映射后，由于资源统计功能在每个运算节点上只需要一个任务，所以也不需要复制通信子。资源统计模块执行 MPI_Bcast 即可接收主控节点对资源统计的配置，获得这个配置信息后资源统计模块计算统计的资源信息的空间大小并将统计的资源信息写入结构体中。至于资源信息统计，理想的实现是在某项资源信息发生变化时统计任务就能够获知资源变化并发送资源信息，但需要在编程中使用了某项资源时由并行程序主动通知资源统计任务，这将给编程带来极大的不便。本文的实现是资源统计任务每隔一定时间查询资源信息，当某项资源信息变化时，统计任务将资源信息发送给主控节点。

运算节点可以统计以下几项资源信息：CPU 利用率、就绪 MMPI 任务优先级表和 MMPI 程序剩余存储空间。uC/OS 中提供了统计函数 OS_TaskStat，该函数使用全局变量 OSCPUUsage 来保存统计值，并将结果转换成 0 到 100 内的整数值。uC/OS 系统一共具有 64 个优先级，本文将 16 到 47 共 32 个优先级分配给 MMPI 并行任务使用。由于任务优先级数目有限，当运算节点上执行的 MMPI 任务过多时，主控节点不能继续向运算节点分配并行任务。主控节点向运算节点发送并行程序，运算节点会建立一个或多个任务，需要存储一份或

者多份程序，本文为并行程序分配了 4MB 的存储空间，当运算节点上的空间不足时，主控节点不能分配新的任务给运算节点。

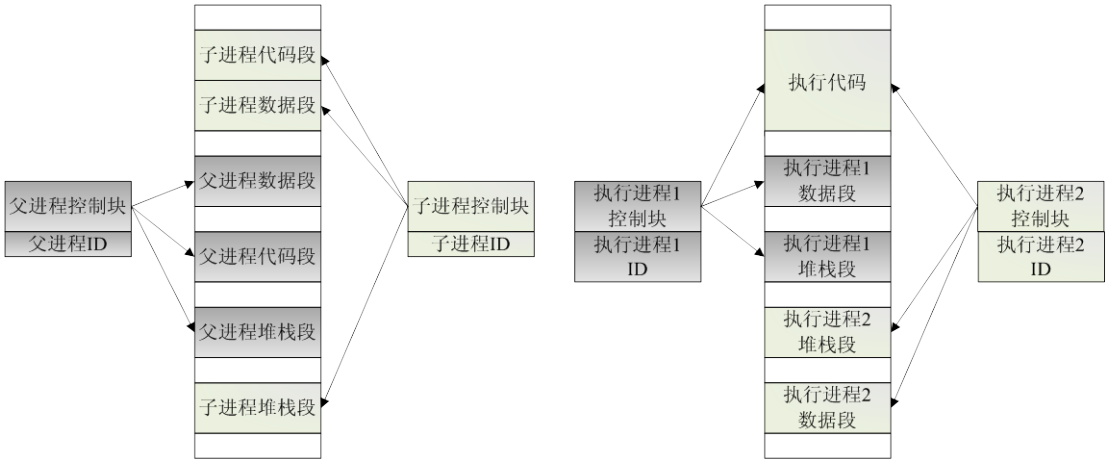
4.3.2 MMPI 任务加载

在 Linux 系统下，每个计算节点的 MMPI 程序执行时会读入映射文件，将应用中所有任务的映射信息保存到全局通信子中，然后为每个执行进程拷贝一份通信子，这份通信子称为执行进程的私有通信子。在 Linux 下通信子的拷贝是通过使用 fork 函数在创建执行进程时由操作系统拷贝数据段来完成，由于 uC/OS 操作系统没有提供 fork 功能，所以必须为 uC/OS 提供系统扩展以实现 MMPI 任务的建立和执行。

3.3.2节论述了主控节点调度和发送并行程序的过程，主控节点发送给所有运算节点的调度结果是相同的，运算节点在接收到调度结果后会处理调度结果，主要是筛出选分配给当前运算节点执行的任务序列。在接收到并行程序后，根据任务序列创建对应数目的并行任务。uC/OS 操作系统分配了 32 个优先级给 MMPI 并行任务使用，每创建一个并行任务必须为之分配对应的优先级。本文对分配的运算节点上的一个应用中的任务按执行顺序的先后从高到低分配优先级。这要求调度结果中除了任务到节点的映射序列外还包括一个执行顺序到任务号的序列，运算节点在处理调度结果时会按照任务执行顺序将分配到当前节点上的任务号提取出来形成一个序列。按照该序列的顺序，加载任务在分配给 MMPI 并行任务的优先级范围中查找到级别最高的优先级数值并将其分配给并行任务。

uC/OS 在创建任务时可以传递参数，加载任务在创建并行任务时通过参数给并行任务传递调度信息、当前任务的任务号以及通信域值。Linux 下每个并行任务的私有通信子中记录了当前进程所要执行的任务的标识号，这是通过在子进程设置私有通信子的任务标识号实现的。运算节点要实现对多 MMPI 任务的支持也必须实现对通信子的复制，而且这个私有子还必须作为全局变量能够被消息原语访问。本文仿照了 Linux 下 fork 函数的思想，fork 函数将父进程的数据段进行了拷贝，由于全局变量是存储在数据段中，因此对数据段的拷贝就实现了对通信子的拷贝，每个并行任务拥有自己的数据段从而就拥有了私有的通信子。图 4-10是在 Linux 和 uC/OS 操作系统下复制数据段的示意图。在 uC/OS 下每个任务在建立时由系统分配私有的堆栈段，必须在 uC/OS 下实现对数据段拷贝功能。

uC/OS 无法解析和执行 Linux 下 ELF 格式或任何其他格式的可执行文件，



a)Linux 下　　b)uC/OS 下

图4-10 Linux 和 uC/OS 下数据段的拷贝

因此 uC/OS 下程序的执行与在裸机上执行相同，即都是二进制机器码的执行。这要求主控节点在发送应用程序时必须以二进制机器码格式发送，而不是具有例如 ELF 格式的文件。gcc 提供了将可执行文件转换成二进制机器码文件的工具，并且 Linux 中可执行文件加载程序也提供了对例如 ELF 格式文件的解析。本文首先在主控节点上对并行应用程序进行解析，分离出其中的代码段及数据段，并分别将映射信息、代码段和数据段发送给运算节点。运算节点接收到映射信息后即可获知当前节点上需要建立的并行任务的数目以及每个进程的任务标识号，接收到任务段和数据段后，由于各个任务的程序代码完全相同，只是执行程序中不同的部分，所以多个并行任务可以共用一个代码段，uC/OS 中固有的每个进程具有私有的堆栈段，在进程建立时为每个进程指定堆栈段，图 4-10 b)是本设计中 uC/OS 系统中建立任务时为任务拷贝数据段的示意图。

数据段则根据需要建立的并行任务的数目拷贝相同的份数，因为每个并行任务需要私有的通信子，即私有的数据段，但是在拷贝数据段后有两个需要解决的问题。首先是所有的并行任务执行的程序相同，只是执行了程序中不同的部分，但是在程序中访问通信子所使用的地址是相同的，为每个并行任务拷贝数据段需要实现每个并行任务使用同一个虚拟地址能够物理内存中的不同单元；其次是 uC/OS 仅为建立的进程分配绑定了堆栈段，而没有为任务指定数据段，需要为并行任务指定其数据段。这两个问题都可以通过虚拟内存的分页机制来解决，虚拟内存使用内存管理单元（Memory Manage Unit，MMU）来实现虚拟地址到物理地址的映射，通过配置映射关系就可以使一个虚拟地址映射到不同的物理地址。虚拟内存分页机制中使用了页表来保存虚拟页面到物理

页面的映射，并将页表中的一部分放到了快表（Translation-Lookside Buffer，TLB）以加速地址转换，本设计中在建立并行任务时在进程控制块中为其建立保存地址转换关系的页表，并行任务通过私有的页表确定访问到对应的数据段，既实现了私有数据段的访问，也能实现数据段与对应并行任务的绑定。任务执行时访问全局变量通信子时会由于每个进程的页表设置不同而访问不同的内存区域，这样每个执行进程就拥有私有的通信子。uC/OS 在建立任务的过程中会将接收到的映射信息以及任务标识号作为参数传递给 MMPI 应用程序，应用程序根据接收到的参数来设置私有通信子及任务标识号，然后根据通信子中的任务映射以及任务标识号来执行程序中对应部分，实现和其他节点任务的通信。

在实现上述各个功能后，uC/OS 系统可以完成 MMPI 任务的加载。在完成初始化工作后，运算节点即可接收其他节点发送过来的消息。并行任务的建立由加载任务完成，因此 uC/OS 系统在启动时就会执行 MPI 任务创建模块，加载任务会循环接收从主控节点发送过来的有关 MMPI 任务属性、调度结果的消息和 MPI 程序，分配通信子结构体并根据映射结果创建多个 MMPI 任务。在某个 MMPI 任务执行完毕后，回收分配给该应用的结构体。当系统中没有其他应用执行时，系统执行空闲统计任务并等待主控节点发送新的任务。

4.3.3 虚拟内存的分页机制

运算节点 uC/OS 下执行的程序是二进制机器码流，由主控节点发送过来的指令流中包含了程序的代码段、数据段等，但是在 uC/OS 下不同的段是不可区分的。采用 fork 函数的思路，MMPI 任务建立模块在接收任务映射信息和 MMPI 并行应用程序后根据映射结果复制 MMPI 应用程序，对应用程序的复制也是对数据段和代码段的复制，因此实现了对通信子的复制。随后即可在每个任务中设置私有通信子的当前任务号即可执行并行任务。为任务建立私有通信子的问题通过复制 MMPI 应用程序解决，但是在复制通信子后如何将映射结果写入私有通信子是接下来需要解决的问题。在 Linux 下，MMPI 程序在初始化过程中读入映射文件将映射结果保存在私有通信子中，然后创建子进程复制通信子，此时映射结果已经保存在通信子中，复制后每个子进程即可获得映射信息。但 uC/OS 下并行应用的任务映射结果是由主控节点使用 MMPI 功能函数发送给运算节点，在为并行程序创建任务时映射结果已经保存在内存中，映射结果不能以文件形式读入，并且这个映射结果存放的位置是不缺定的。

MMPI 任务加载时需要为 MMPI 任务复制数据段，数据段的访问和绑定需

要解决通过一个虚拟地址访问对应不同物理地址存储空间的问题。此外，并行应用程序在主控节点或线下编译，并在运算节点上加载执行，应用程序在编译时并不知道会放置到运算节点的本地存储器中什么位置，在编译链接时程序的地址与在存储到运算节点中本地存储器的地址也是不同的，如果不加处理，程序执行过程的跳转等根据代码地址执行时就会出现地址访问错误，这些问题可以使用虚拟内存分页技术解决。分页机制将物理内存分为大小相同的页，程序在执行时使用逻辑地址访问存储器，使用逻辑地址得到的虚拟页和物理页是对应的。逻辑地址由页号和页内偏移组成，使用逻辑地址进行内存访问都要经历虚拟地址到物理地址的转换，地址转换过程中页内偏移不变，虚拟页号转换为物理页号。分页机制使用页表存储系统中所有虚拟页号到物理页号的转换关系。在进行内存访问时，系统首先访问页表获得虚拟页号对应的物理页号，然后根据物理页号和页内偏移访问对应的存储单元。

ARCA3 CPU 中的 MMU 有两种虚拟地址转换模式，根据 MMU 的控制寄存器 MCR 中 ATE 位段来决定，该位段为 0 表示采用直接地址映射模式，为 1 时表示采用分页系统模式。在直接地址映射模式下，地址的转换完全是通过硬件来完成的。本文使用分页系统，在分页系统模式下，ARCA3 CPU 可以实现很复杂的基于分页系统的虚拟地址映射，支持四种页面大小，分别是 4K、16K、1M 和 16M。本文采用的是 4K 页面，地址的低 12 位是页内偏移，高 20 位为页号，分页机制的实现就是在页表中设置虚拟页号转换为物理页号的对应关系。ARCA3 的分页系统中包含了内存中的页表和加快地址转换的快表，页表中包含了所有的虚拟页到物理页的映射关系，但使用页表访问存储器，在每次地址访问时都需要首先在页表中查逻辑地址对应的物理地址，然后再使用物理地址来进行真正的存储器访问，这意味每次内存访问都需要两次内存访问。为加快地址转换，分页系统中使用了 TLB 来加速内存访问，ARCA3 CPU 中的 TLB 的结构如图 4-11所示。

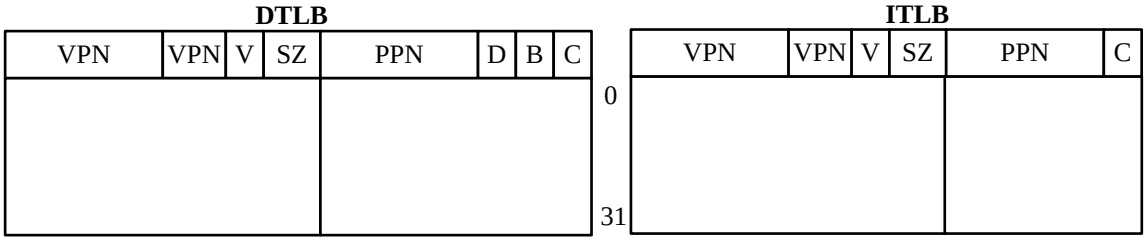


图4-11 ARCA3 中 TLB 结构示意图

在采用分页系统进行内存访问时，ARCA3 CPU 首先根据虚拟页号查找

TLB，如果在 TLB 中无法找到对应项就会触发一次地址 fault 异常，在 fault 异常处理程序中进行 TLB 的配置工作，在完成 TLB 的配置后会再次执行内存访问指令，系统从 TLB 中获取物理页号，并将物理页号和页内偏移拼接为物理地址，使用物理地址进行内存访问。本文要实现分页系统包括两部分工作，在 MMPI 任务创建时的页表的生成和 fault 异常处理程序中对 TLB 的配置。对于页表的建立，uC/OS 系统为每一个 MMPI 并行任务建立了一个私有的页表，这个页表中只存储该任务所访问的页地址转换，因此在该任务执行时如果 fault 异常就只须查找该任务的私有页表，可以减少搜索时间。对于 fault 异常处理程序，TLB 中两项主要内容就是虚拟页号和物理页号，虚拟页号由发生存储器访问异常时的虚拟地址获得，而物理页号在异常处理程序中通过访问任务的私有页表获得，这样就完成了 TLB 的修改，继而可以完成程序的正常执行。

4.3.4 并行编程

前文论述了在 uC/OS 下 MMPI 并行任务的加载和执行过程，uC/OS 操作系统提供的功能有限，许多 Linux 系统提供的功能，例如运行库支持、可执行文件的解析和加载等等。本文的多核系统中 MMPI 并行应用程序实际上时执行于运算节点的 uC/OS 操作系统下，在 uC/OS 下编写基于 MMPI 的并行应用程序有诸多限制。

首先是对栈空间的使用，栈是过程调用中参数存储、寄存器保护和任务切换时现场保护使用的存储单元，程序的局部变量的存储单元也开辟在栈中。如果栈发生了溢出，任务的执行会发生错误。其中过程调用和现场保护的过程由操作系统完成，并且在分配栈空间时已经考虑到这两个过程所需的空间，因此在并行编程时要注意局部变量的使用。当需要使用需要较大存储空间的变量时将其放在数据段中，这可以通过初始化的全局变量来实现。

然后是编写并行应用程序的入口参数。一般情况下在编写 C 程序时 main 函数的参数是固定的，但 uC/OS 系统下建立任务时通过参数向任务传递了当前任务的任务标识号、应用中所有任务的调度映射结果和并行应用的通信域，应用程序在执行时通过参数传递获取这些信息，从而保证并行任务能够正常执行并且各个任务执行并行程序中对应的部分。因此编写运行于本文多核系统下的并行应用程序的入口参数是一个结构体指针，并且在进入并行程序后必备的处理是对传递的参数的处理。

其次是本文的多核系统中节点间通信是基于 MMPI 实现的，uC/OS 节点提供了对 MMPI 库的支持，但 uC/OS 下 MMPI 库的实现与 Linux 下 MMPI 库的

实现有较大不同，由于编写的并行程序运行于运算节点上，所以应该使用uC/OS下的MMPI库。除此之外，uC/OS并未提供对其他函数库的支持，因此并行编程中除 MMPI 功能函数外，其他所有函数都需要用户实现；若使用Linux下的库函数，须将其静态编译在并行程序的可执行文件中。

最后，由主控节点发送到运算节点的并行程序时二进制指令流，而不能是具有格式的可执行文件的内容。uC/OS 操作系统无法解析具有格式的可执行文件，并行任务在运算节点上的执行与在裸核上执行的情形是类似的。用户需要在主控节点上将可执行文件中的代码段和数据段等提取出来，然后将提取出来的部分发送给运算节点执行。

4.4 本章小结

本章论述了运算节点操作系统的设计与实现，介绍了运算节点上 uC/OS 系统的基本结构，说明了选用 uC/OS 作为运算节点操作系统原因。本章将uC/OS 操作系统移植到 ARCA3 处理器，实现了 ARCA3 下的任务切换、现场保护等基本功能以及中断处理流程，保证了普通任务切换和中断级任务切换现场保护的一致性。本章在仿照 Linux 下中断前半部和后半部的设计思想为DMA_NI 设计了中断处理策略，实现了中断处理任务中对信号量的使用。

在操作系统扩展层，通过接收主控节点发送的资源配置信息，运算节点可以只统计所需的信息项。通过在系统初始化时设置通信子的映射结果，uC/OS系统可以在不同网络规模下都能够与主控节点资源统计模块完成 MMPI 通信。加载任务能够根据调度结果建立并行任务，并通过参数传递并行任务执行时所需信息，加载任务通过复制数据段的形式拷贝通信子，为每一个并行任务建立私有通信子。通过分页机制，保证了并行任务能够正确访问到私有的通信子并且只能够访问到所拥有的数据段，使得并行任务能够正确执行。

第5章 功能验证与测试

本章对多核操作系统进行了验证和测试，首先介绍了 M5 仿真平台的参数设置，然后验证了多核操作系统执行并行应用的功能的正确性。在验证本文多核操作系统实现支持并行应用的功能后，测试了多核操作系统的性能，分析了并行程序执行的过程中各个节点的开销，并评测了应用规模和网络规模对各个阶段开销的影响。

5.1 仿真平台设置

本文在实验室已有的 M5 多核仿真平台[48]进行多核系统测试，已有平台各个节点的配置相同。本文多核系统中主控节点和运算节点的设置不同，uC/OS 系统需要的资源较 Linux 系统少，因此运算节点的存储器需求较小，并且运算节点不需要挂接外设。多核系统中主要的参数设置如表 5-1所示。

表5-1 仿真平台参数设置

处理器 频率(Hz)	局部总线 频率(Hz)	主控节点 存储器(Byte)	运算节点 存储器(Byte)	网络接口 频率(Hz)	发送 FIFO 个数
2G	1G	96M	5M	1G	1
接收 FIFO 个数	网络接口 FIFO 深度	查找表 表项个数	数据包长度 (flit)	包头 (flit)	标志信息 (flit)
4	8	4	11	1	2
有效数据 (flit)	路由器频率 (Hz)	路由器 FIFO 深度	网络拓扑	路由器输出端 口仲裁策略	
2	1G	11	2D-Mesh	round-robin	

5.2 操作系统基本属性

本文的分布式操作系统包括主控节点 Linux 操作系统和运算节点 uC/OS 操作系统，两种系统的基本属性如表 5-2所示。由于在 uC/OS 操作系统下使用的所有的存储空间，包括使用内存分区表管理的可动态分配和回收的空间，都必须在使用前分配相应的存储区域，因此其内核大小能够真实反映 uC/OS 运行时所需的内存大小，表 5-2中的 uC/OS 内核大小是在 uC/OS 支持 MMPI 通信后测得的，其中包括 64KB 用于实现消息接收队列而分配的静态空间。此外，

为加载和执行并行任务，本文的系统测试中为运算节点额外开辟了 4MB 用于存储主控节点发送的并行程序的空间。消息队列空间和程序存储空间的容量可以根据需要配置，但并非 uC/OS 系统运行所必须的空间。启动时间是通过测量系统启动后第一个用户程序执行的时间获得的。表中数据显示 uC/OS 运行时所需的内存与启动时间均远远小于 Linux 操作系统。

表 5-2 两种操作系统基本属性

比较项	uC/OS	Linux
内核大小	137KB	7.1MB
启动时间	~20ms	~30s

5.3 功能验证

本文的多核操作系统为并行应用利用多核系统全局资源提供支持，并行应用的加载过程与主控节点上的资源统计模块、并行调度模块以及运算节点上的资源统计任务、并行程序的加载任务相关，其执行过程与运算节点的分页机制和 MMPI 库相关。并行应用的正确执行能够证明本文多核操作系统基本功能的正确性。

5.3.1 实验设计

多核系统功能验证主要包括以下几个方面：1)资源统计功能的正确性，主控节点能够接收运算节点发送的资源信息，并行调度模块能够正确访问到节点资源信息；2)运行时调度功能的正确性：调度模块能够将任务分发到节点上执行；3)MMPI 任务加载的正确性：主控节点分发调度信息，运算节点给出应答，主控节点分发应用程序，运算节点建立任务，传递参数；4)MMPI 并行任务执行的正确性。

根据验证要求，硬件系统采用一个具有 4 个节点的多核系统，0 号节点作为主控节点，MMPI 程序在 1~3 号运算节点上执行。测试案例的任务描述文件由任务图自动生成工具 TGFF[49] 产生，每个任务的执行时间与通信关系如图 5-1所示。MMPI 并行程序中包含 4 个任务，任务标识号为 0~3，不考虑数据传递时该并行应用串行执行时间为 31+32+59+45=167ms。

TGFF 只产生任务描述文件，没有对应的并行程序。本文实现了一个测试案例生成模块，该模块根据 TGFF 任务图生成对应的 C 语言测试程序。为实现该模块本文首先测试了两个 64×64 的矩阵相乘完成全部计算、一行元素计算

```
@HYPERPERIOD 1
@COMMUN_QUANT 0 {
# type    quantity
    0    590.423
    1    636.953
    2     1006.2
    3    726.186
}
@EXE_TIME 0 {
# type    exec_time
    0  0.0317387
    1  0.0323113
    2  0.0595543
    3  0.045102
}
```

```
@TASK_GRAPH 0 {
    PERIOD 1

    TASK t0_0   TYPE 1
    TASK t0_1   TYPE 3
    TASK t0_2   TYPE 2
    TASK t0_3   TYPE 0

    ARC a0_0    FROM t0_0  TO  t0_1 TYPE 0
    ARC a0_1    FROM t0_1  TO  t0_3 TYPE 1
    ARC a0_2    FROM t0_0  TO  t0_2 TYPE 2
    ARC a0_3    FROM t0_2  TO  t0_3 TYPE 3

    HARD_DEADLINE d0_0 ON t0_3 AT 1
}
```

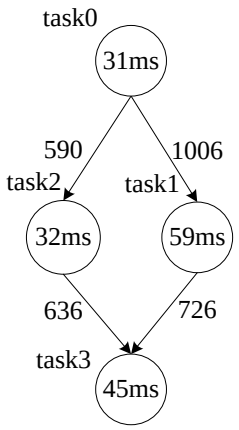


图5-1 测试案例任务描述文件及通信任务图

和一个元素计算的时间，然后将并行任务的计算以多个矩阵运算、多行元素计算和多个元素计算的形式实现。结合任务之间的通信，该模块生成的任务先执行数据接收，之后完成计算，最后发送数据给其他节点。生成测试文件后使用交叉编译器编译测试文件，注意最后调度需要的不是具有一定格式的可执行文件，而是去除段描述信息等的二进制指令文件。在主控节点上调用调度模块，将任务描述文件和由测试文件生成的二进制机器码文件作为参数传递给调度模块。在调度模块以及测试案例中插入获得当前时刻的伪指令记录并行程序执行过程中关键点的执行时间，由获得的时刻值可以计算出各个阶段的开销。

5.3.2 验证结果

并行程序的执行过程可以大概分为并行模块调度、程序分发、运算节点建立任务以及并行任务执行等几个过程，这些过程中程序分发、任务建立以及执行是并行执行的，没有绝对的先后关系。本次测试中调度结果是将 0 号和 1 号任务映射到 1 号节点运行，2 号和 3 号任务映射到 2 号节点运行，3 号节点空闲。测试案例最终的执行结果的正确性能够说明本文分布式操作系统各项功能得到了实现。

由于主控节点是依次将分发调度结果以及并行程序发送到各个运算节点的，所以运算节点接收到并行程序的时刻并不相同。运算节点在接收到程序后就可以建立并行任务，由于加载任务的优先级高于建立的 MMPI 任务，运算节点会在完成任务建立之后才能够开始执行并行任务。而提前接收到程序的任务在建立任务或者开始并行任务执行时某些节点可能仍然在等待主控节点发送并

行程序。表 5-3显示了测试案例执行时各阶段开销，其中任务分发阶段的开销是从主控节点开始发送到所有运算节点接收到应用程序的开销，任务建立开销是所有节点建立并行任务的开销的平均值，任务执行开销是任意一个任务开始执行到所有节点执行完毕的时间开销，总的时间开销是从开始调用并行调度模块到所有任务执行结束的开销。

表 5-3 并行程序执行各阶段开销

阶段	运行时调度	任务分发	任务建立	任务执行	总执行时间
开销（us）	942.217	231.588	21.069	123626	124820.5

本文的动态调度算法的执行开销与网络规模和任务数成正比。由于主控节点只向分发了计算任务的运算节点发送程序，因此分发阶段的开销与节点数成正比，并且随着并行程序的增大而增加。由于本文测试案例的执行时间远远大于其他各个阶段，所以图 5-2只显示了调度、分发和任务建立三个阶段的开销。可以看出运行时调度的开销远远大于任务分发和任务建立，并且本测试案例中只有 4 个任务，当任务数增加时，调度开销会进一步增大。不考虑串行执行过程中的数据传输开销，本文的多核系统将测试案例分配到两个节点上执行，总的执行时间的加速比为 167/124.8205=1.338。

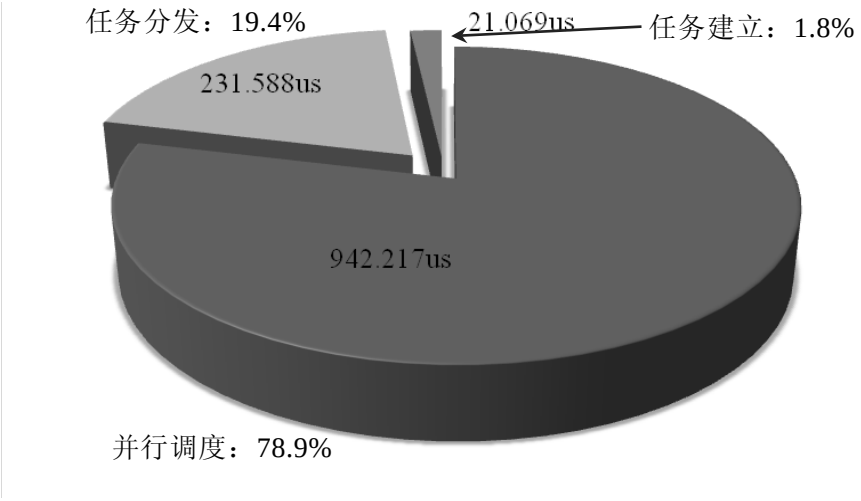


图5-2 并行程序调度的各阶段开销

5.4 性能测试

本节首先测试了多核系统中节点间点对点多核通信的性能，分析了不同种类节点间通信性能变化的原因。然后测试了任务量不同的应用在网络规模不同

的多核系统中调度、分发、加载和执行各个阶段的开销，并分析了应用规模和网络规模变化对各个阶段开销变化的影响。

5.4.1 uC/OS 下 MMPI 通信性能

多核系统中包括主控节点和运算节点，节点间通信包括主控节点发送运算节点接收，运算节点发送运算节点接收，以及运算节点发送主控节点接收三种类型，考虑到系统中只有一个主控节点，所以没有主控节点之间互相通信这种测试类型。设计案例测试了这三种通信延迟，图 5-3是通信开销的比较。可以看出当发送端不同时通信延迟相差较大，这是因为产生性能差距的主要原因在于 uC/OS 系统在发送端消除了数据从用户空间到系统空间的拷贝，而减少的拷贝开销与通信量成正比，因此随着消息长度的增加，两者的性能差距逐渐增大。而在接收端，uC/OS 系统下 DMA_NI 与 CPU 的交互策略与 Linux 系统下完全相同，所以虽然 uC/OS 节点之间的通信延迟仍然要小一些，但两者基本相同。这个差距是由于在 Linux 下的一些操作较 uC/OS 系统下操作相对而言更加复杂带来的，所以这个性能差距值是一个常数，随着通信量的增加在通信开销中所占比例逐步下降，在数据量为 0.5KB 时大约占 2.5%。

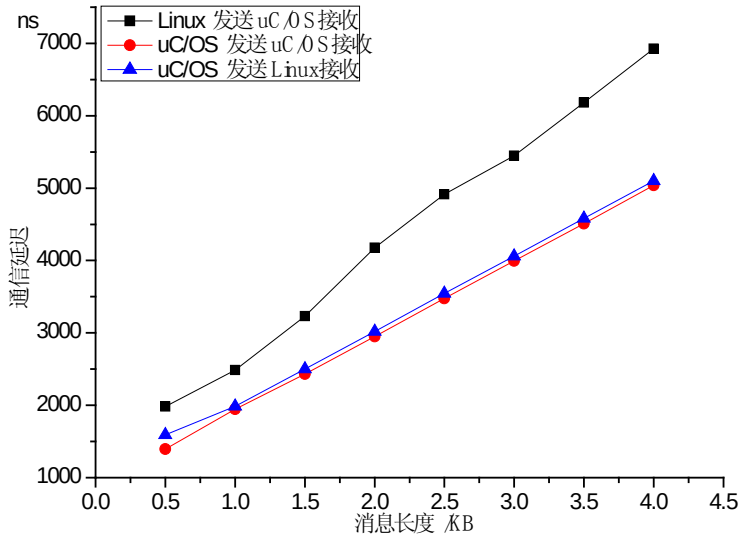


图5-3 节点间通信开销

5.4.2 应用规模和网络规模对各阶段开销影响

本节测试了不同的网络规模和应用规模对并行应用在本文多核系统上执行的各阶段的开销的影响。并行应用的执行分为调度、分发、任务建立和任务执行四个阶段，其中每个阶段的时间开销的含义在5.3.2节中已有阐述。使用的三

个测试案例分别有 6 个、15 个和 20 个任务，网络规模有 2×3 和 3×4 两种，将三个测试案例分别在两种规模下的多核系统执行，统计出各个阶段的开销。

当网络规模确定时，三种应用的各阶段开销随着任务数的增加而增加，图 5-4显示了三种不同应用在 2×3 网络中执行时各阶段开销。运行时调度阶段的开销所占比例一直最大，当网络规模确定时，并行调度的开销与任务数正比，这是由于本文的调度策略只调度新提交的并行应用中的任务，而调度算法是遍历所有任务一一进行分配，每个任务分配过程中所完成的工作基本相同，因此运行时调度的开销与任务数成线性关系。本文实现的主控节点并行调度模块根据调度结果只向分配了并行任务的运算节点发送并行程序，因此任务分发阶段的开销与使用的节点数和并行程序的大小相关，不同规模的应用使用的节点数基本相同，而并行程序则是逐步增大，因此任务分发的开销也是逐步增大。任务建立阶段的开销是各个节点建立任务开销的平均值，因此任务建立开销与节点平均任务数密切相关。三种应用规模下每个节点平均分配的任务数分别为 1.2 个、3 个和 5 个任务，并且任务建立阶段的主要开销在于复制并行程序，当节点上上只有一个任务时，运算节点的加载任务不需要拷贝程序，因此第一种应用的任务建立开销远小于后两种应用的任务建立开销。

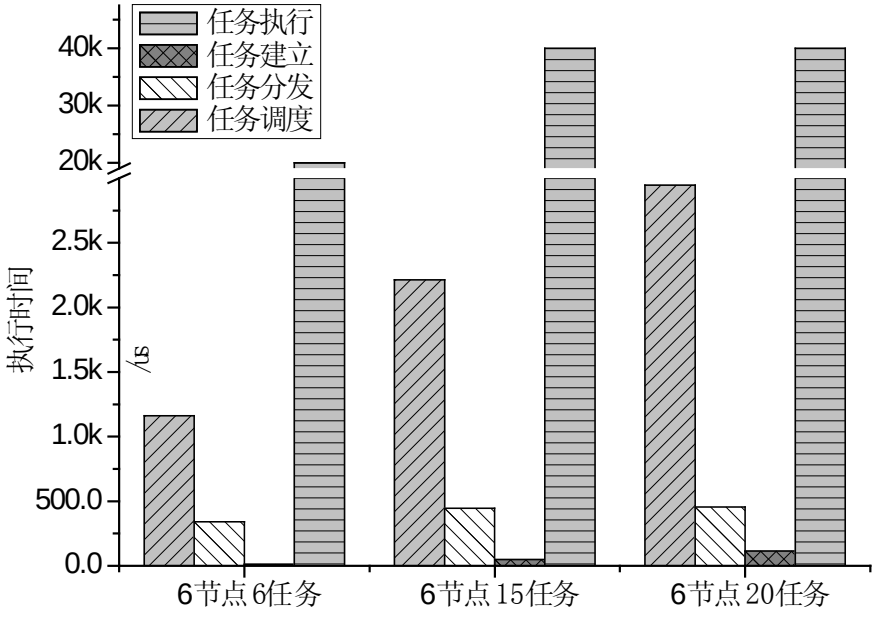


图5-4 不同任务规模应用的各阶段开销

当应用规模确定时，运行时调度的开销随网络规模的增加而增加，任务分发和建立的开销基本保持不变。图 5-5显示了 15 个任务的并行应用分别在 2×3 和 3×4 网络下执行的各个阶段的开销。调度的开销在两种情景下有显著差距，这是由于调度算法执行的过程与网络节点数也是相关的，对每一个任务，

调度算法首先遍历网络中所有节点，找出可以可分配任务的节点集，然后在这个节点集中找出通信开销最小的节点，该节点即为任务分配节点。因此当网络中节点数增加时，调度算法第一步的开销必然增加，但第二步的开销则视系统负载情况可以增加、持平甚至减少。本文的测试案例执行前运算节点上运行的只有资源统计和时钟中断任务，这两种任务的计算量都很小，所以运算节点可以看作是空载的，因此 3×4 网络下的运行时调度的开销明显的高于 2×3 网络下开销。对于任务分发和任务建立，由于两种测试下分配的节点数、任务数和并行程序大小都是相同的，因此理论上任务分发和任务建立的开销应该是相同的。图 5-5中任务建立的开销基本完全相同，而任务分发的开销也由差距，这是由于分配的节点分布在网络中不同地方造成的，其差距小于 10%。

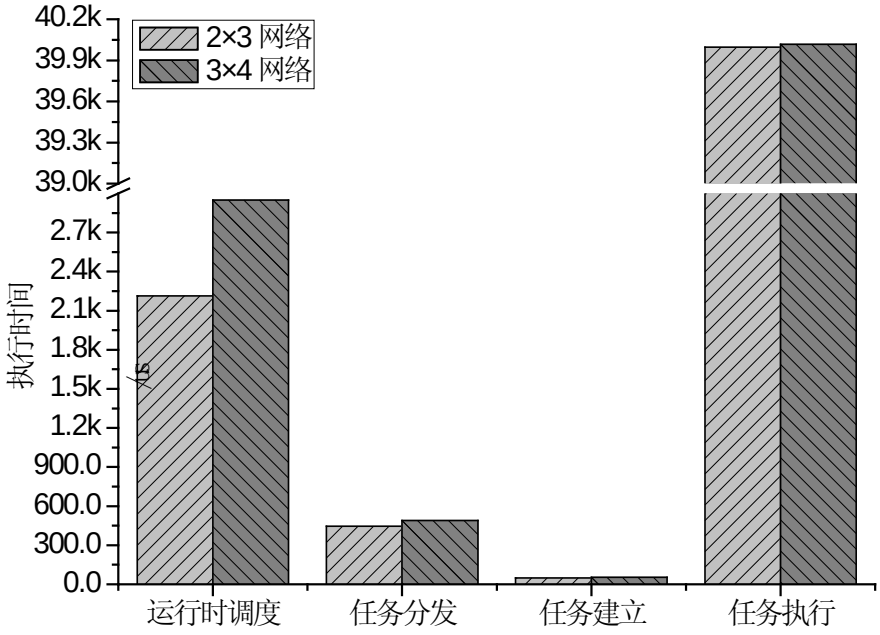


图5-5 不同网络规模下应用的各阶段开销

5.5 本章小结

本章验证了分布式多核操作系统的基本功能，验证结果证明该操作系统实现了并行程序的动态调度功能。根据并行应用调度执行的过程，本章测试了并行应用执行过程中各个阶段的开销，分析了各个阶段开销产生的原因以及影响因素。本章测试了多核系统中主控节点和运算节点不同种类通信的开销，分析了节点间通信开销不同的原因在与发送端的数据拷贝。最后本章测试了网络规模和应用规模对并行应用执行的过程中各个阶段的影响。

结　论

本文在实验室已有的基于 M5 的多核仿真平台上，设计了一个分布式多核操作系统。该操作系统支持基于 MMPI 的并行应用在多核系统上的执行，能够监测全局系统资源的分布和变化，并在并行应用执行时进行运行时调度。本论文主要取得了以下研究成果：

（1）设计和实现了主控节点上的资源统计模块和并行调度模块。资源统计模块读入网络和资源统计配置文件，根据网络规模和统计信息配置使用共享内存建立资源池存储资源信息，使用信号量同步多个接收子进程以及并行调度模块对资源池的访问。资源统计模块根据网络配置生成与运算节点通信所需的映射文件，具有良好的可扩展性。并行调度模块在并行应用执行时使用资源池中运算节点资源信息调度并行应用的执行，运行时调度使用了系统最新的资源信息，在调度完成后将调度结果与并行程序发送到运算节点。

（2）设计和实现了运算节点上 uC/OS 操作系统及其相关功能。本文将 uC/OS 操作系统移植到 ARCA3 处理器，实现了该操作系统下任务管理、异常处理等功能；开发了 DMA_NI 模块的驱动，为多核系统节点间通信提供了支持；拓展了该操作系统已有的内存管理，实现了多个内存块的分配和回收。在操作系统拓展层中，实现了资源统计任务，该任务能够根据主控节点的配置统计所需资源信息并发送给主控节点；实现了 uC/OS 对虚拟内存分页机制和任务加载机制，为 MMPI 并行应用的执行提供了支持。结合 uC/OS 节点下并行任务的加载执行过程，提出在本文多核系统并行编程的要求。

（3）支持并行应用的运行时调度以及基于运行时调度的应用执行，并行程序的执行包括调度、分发和加载执行三部分。并行应用在执行时由主控节点并行调度模块通过运行时调度决定并行任务分配到哪些运算节点上执行，并将调度结果发送给分配了任务的运算节点，在获得应答后并行调度模块将并行程序分发到运算节点上。运算节点根据调度结果和并行程序建立多个并行任务，并将映射结果、通信域的值、任务标识号以参数的形式传递给并行任务，完成并行应用的执行。

本文实现了运行时调度功能，设计了支持基于 MMPI 并行应用加载执行机制。该机制支持基于运行时调度的并行应用执行，实现了对动态调度的支持。本文开发的多核操作系统可用于评估调度算法的执行开销和调度结果，以及嵌入式多核系统的资源管理等。

参考文献

[1] J. Ceng et al. MAPS: An Integrated Framework for MPSoC Application Parallelization[C]. DAC 2008, Anaheim, California, USA, 2008: 754–759.

[2] D. Geer. Chip Makers Turn to Multicore Processors[J]. Computer, 2005, 38(5): 11-13.

[3] Wayne Wolf. Multiprocessor System-on-Chip[J]. Signal Processing Magazine, IEEE, 2009, 26(6): 50-54.

[4] Sudeep Pasricha, Young-Hwan Park. CAPPS: A Framework for Power–Performance Tradeoffs in Bus-Matrix-Based On-Chip Communication Architecture Synthesis[J]. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2010, 18(2):209-221.

[5] R. Ho, K. W. Mai et al. The future of wires[J]. Proc. IEEE, 2001, 89(4): 490–504.

[6] L. Benini, G. De Micheli. Networks on Chips: A New SoC Paradigm[J]. Computer, 2002, 35(1): 70-78.

[7] P. Pande, C. Grecu, A. Ivanov et al. Design, Synthesis and Test of Networks on Chip[J]. IEEE Design and Test of Computers, 2005, 22(5): 404-413.

[8] Yuriy Sheynin, Elena Suvorova. Complexity and Low Power Issues for On-chip Interconnections in MPSoC System Level Design[C]. VLSI Technologies and Architectures, IEEE Computer Society Annual Symposium on, 2006: 6-11.

[9] E. Beigné, P. Vivet. Design of On-chip and Off-chip Interfaces for a GALS NoC Architecture[C]. Asynchronous Circuits and Systems, 12th IEEE International Symposium on, 2006: 174-183.

[10] Radu Marculescu, Umit Y.Ogras, Li-Shiuan Peh et al. Outstanding Research Problems in NoC Design: System, Micro-architecture, and Circuit Perspectives[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2009, 28(1): 3-21.

[11] Tyrone Tai-On Kwok, Yu-Kwong Kwok. On the Design, Control, and Use of a Reconfigurable Heterogeneous Multi-Core System-on-a-Chip[C]. IEEE International Symposium on Parallel and Distributed Processing, Miami, FL, 2008: 1-11.

[12] D. Göhringer, M. Hübner, V. Schatz et al. Runtime Adaptive Multi-Processor System-on-Chip: RAMPSoC[C]. In Proc. of IPDPS 2008, 2008: 1–7.

[13] Jung-Ho Lee, Sung-Rok Yoon, Kwang-Eui Pyun . A Multi-Processor NoC Platform Applied on the 802.11i TKIP Cryptosystem[C]. Design Automation

Conference, Seoul, Korea, 2008: 42-51.

[14] Y. Y. Ye, L. Duan, J. Xu et al. 3D Optical Networks-on-chip(NoC) for Multiprocessor Systems-on-chip(MPSoC)[C]. IEEE International Conference on 3D Systems Integration, San Francisco, CA, 2009: 83-88.

[15] 张庆利. 多核 SoC 中的片上网络关键技术研究[D]. 哈尔滨: 哈尔滨工业大学微电子学与固体电子学专业博士论文, 2008: 1-3.

[16] Jose F. Martınez, Engin Ipek. Dynamic Multicore Resource Management: a Machine Learning Approach[J]. Micro, IEEE, 2009, 29(5): 8-17.

[17] 郑臻伟. NoC 片上分布式通用操作系统设计[D]. 杭州: 浙江大学计算机系统结构专业硕士学位论文, 2008: 26-29.

[18] W. Wolf, A. A. Jerraya and G. Martin. Multiprocessor System-on-Chip (MPSoC) Technology[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2008, 27(10): 1701-1713.

[19] 杜高明. MPSoC—NoC 多核体系结构及原型芯片实现技术研究[D]. 合肥: 合肥工业大学精密仪器及机械专业博士学位论文, 2007: 7-10.

[20] 黄凯. 面向特定应用的 MPSoC 设计流程平台研究[D]. 杭州: 浙江大学电路与系统专业博士学位论文, 2008: 2-5.

[21] L. Rainer et al. Programming MPSoC platforms: Road works ahead![C] Design, Automation & Test in Europe Conference & Exhibition, 2009: 1584-1589

[22] G. Sassatelli et al. Architectural Issues in Homogeneous NoC-Based MPSoC [C]. 18th IEEE/IFIP International Workshop on Rapid System Prototyping, 2007: 139-142

[23] 吴佳骏. 多核多线程处理器上任务调度技术研究[D]. 北京: 中国科学院计算技术研究所计算机系统结构专业博士学位论文, 2006: 4-5.

[24] D. Bernstein, M. Rodeh, and I. Gertner, On the complexity of scheduling problems for parallel/pipelined machines[J]. IEEE Trans. Comput., 1989, 38(9): 1308–1313.

[25] Ayse Kıvılcım Coskun et al. Static and Dynamic Temperature-Aware Scheduling for Multiprocessor SoCs[J]. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2008, 16(9): 1127-1140.

[26] Diana Göhringer, Michael Hübner, Etienne Nguepi Zeutebouo et al. CAP-OS: Operating System for Runtime Scheduling, Task Mapping and Resource Management on Reconfigurable Multiprocessor Architectures[C]. Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, 2010: 1-8.

[27] Rob Knauerhase, Paul Brett et al. Using OS Observations to Improve Performance in Multicore Systems[J]. Micro, IEEE, 2008, 28(3): 54-66

[28] Chandra, A., Shenoy, P. Hierarchical Scheduling for Symmetric Multiprocessors[J]. Parallel and Distributed Systems, IEEE Transactions on, 2008, 19(3): 418 – 431

[29] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson et al. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator[C]. ISCA'09, Austin, Texas, USA, 2009: 140-151.

[30] Nicolas Saint-Jean, Gilles Sassatelli et al. HS-Scale: a Hardware-Software Scalable MP-SOC Architecture for embedded Systems[C]. VLSI, ISVLSI '07, IEEE Computer Society Annual Symposium on，2007: 21-28.

[31] Andrew Baumann, Paul Barhamy, Pierre-Evariste Dagand et al. The Multikernel: A New OS Architecture for Scalable Multicore Systems[C]. SOSP'09, Montana, USA, 2009: 29-43.

[32] Chao WANG, Bin XIE, Jiexiang Kang et al. On-Chip Operating System Design for NoC-Based CMP[C]. Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, 2010: 163-170.

[33] Vincent Nollet, Theodore Marescaux, Diederik Verkest. Operating System Controlled Network on Chip[C]. DAC 2004, San Diego, California, USA, 2004: 256-259.

[34] Wei HU, Jianliang MA, Binbin WU et al. Distributed On-Chip Operating System for Network on Chip[C]. Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, 2010: 2760-2767.

[35] S.Vakili, S.M.Fakhraie, S.Mohammadi. Evolvable multi-processor: a novel MPSoC architecture with evolvable task decomposition and scheduling[J]. Computers and Digital Techniques, IET, 2010, 4(2): 143-156.

[36] J. Barbosa, Belmiro Moreira. Dynamic Job Scheduling on Heterogeneous Clusters[C]. Parallel and Distributed Computing, ISPDC '09, Eighth International Symposium on, 2009: 3-10.

[37] Juan Antonio Clemente, Javier Resano, Carlos González et al. A Hardware Implementation of a Run-Time Scheduler for Reconfigurable Systems[J]. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2010, PP(99): 1–14

[38] Zexin Pan and B. Earl Wells. Hardware Supported Task Scheduling on Dynamically Reconfigurable SoC Architectures[J]. Very Large Scale Integration(VLSI) Systems, IEEE Transactions on, 2008, 16(11): 1465-1474

[39] N. Binkert, Ronald G. Dreslinski, Lisa R. Hsu et al. The M5 Simulator:

Modeling Networked Systems[J]. Micro, IEEE, 2006, 26(4): 52-60

[40] Fangfa Fu, Siyue Sun, Xin'an Hu et al. MMPI: A Flexible and Efficient Multiprocessor Message Passing Interface for NoC-Based MPSoC[C]. The 23rd IEEE International SOC Conference, 2010.

[41] William Stallings. 操作系统, 精髓与设计原理[M]. 陈渝, 译. 北京: 电子工业出版社, 2007: 38-41.

[42] Patrik Strömblad. ENEA Multicore: High Performance Packet Processing Enabled with a Hybrid SMP/AMP OS Technology [EB/OL]. http://www.enea.com.

[43] Christopher Hallinan. 嵌入式 Linux 基础教程[M]. 华清远见嵌入式培训中心, 译. 北京: 人民邮电出版社, 2009: 1-22.

[44] 任哲. 嵌入式实时操作系统 uC/OS-II 原理及应用[M]. 北京: 北京航空航天大学出版社, 2007: 12-13.

[45] FU Fangfa, HU Xin'an, WANG Jinxiang at el. A Novel Communication Strategy between PE and NI in NoC-based MPSoC[C]. Laser Physics and Laser Technologies (RCSLPLT) and 2010 Academic Symposium on Optoelectronics Technology (ASOT), 2010 10th Russian-Chinese Symposium on, 2010: 374-377.

[46] V. Karamcheti, A. Chien. Software Overhead on Messaging Layers: Where Does the Time Go?[C] Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, USA, 1994: 51-60.

[47] Kariniemi, H., Nurmi, J. NoC Interface for Fault-Tolerant Message-Passing Communication on Multi-processor SoC Platform[C]. NORCHIP, 2009: 1-6.

[48] Mingyan Yu, Junjie Song, Fangfa Fu et al. A Fast Timing-Accurate MPSoC HW/SW Co-Simulation Platform based on a Novel Synchronization Scheme[C]. The International MultiConference of Engineers and Computer Scientists 2010, Vol II: 1396-1400.

[49] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task graphs for free[C]. In Proc. 6th Int. Workshop Hardware/Software Codesign (CODES/CASHE), 1998: 97–101.

攻读学位期间发表的学术论文

1 胡新安, 付方发, 孙俊, 喻明艳. 基于 NoC 的多核分布式操作系统设计实现. 计算机工程. (已录用, 待发表. 录用时间: 2011.06)

2 FU Fangfa, HU Xin’an, WANG Jinxiang, Yu Mingyan. A Novel Communication Strategy between PE and NI in NoC-based MPSoC. Laser Physics and Laser Technologies (RCSLPLT) and 2010 Academic Symposium on Optoelectronics Technology (ASOT), 2010 10th Russian-Chinese Symposium on, 2010: 374-377.

3 Fangfa Fu, Yuxin Bai, Xin’an Hu, Jinxiang Wang and Mingyan Yu. Low Latency Clustering &Mapping Algorithm with Task Duplication Technique on Cluster-Based NoC. DATICS-IMECS’10, IMECS 2010, HongKong, Volume II: 1374-1379.

4 FU Fangfa, BAI Yuxin, HU Xin’an, WANG Jinxiang, YU Minyan. An Objective-Flexible Clustering Algorithm for Task Mapping and Scheduling on Cluster-Based NoC. RCSLPLT/ASOT 2010, 2010 10th Russian-Chinese Symposium on, 2010: 369-373.

5 Fangfa Fu, Siyue Sun, Xin'an Hu, Junjie Song, Jinxiang Wang and Minyan Yu. MMPI: A Flexible and Efficient Multiprocessor Message Passing Interface for NoC-Based MPSoC. IEEE SoCC, 2010: 359-362.

哈尔滨工业大学硕士学位论文原创性声明

本人郑重声明：此处所提交的硕士学位论文《支持动态任务调度的多核分布式操作系统设计》，是本人在导师指导下，在哈尔滨工业大学攻读硕士学位期间独立进行研究工作所取得的成果。据本人所知，论文中除已注明部分外不包含他人已发表或撰写过的研究成果。对本文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。本声明的法律结果将完全由本人承担。

作者签名：　　日期：2011 年 6 月 30 日

哈尔滨工业大学硕士学位论文使用授权书

《支持动态任务调度的多核分布式操作系统设计》系本人在哈尔滨工业大学攻读硕士学位期间在导师指导下完成的硕士学位论文。本论文的研究成果归哈尔滨工业大学所有，本论文的研究内容不得以其它单位的名义发表。本人完全了解哈尔滨工业大学关于保存、使用学位论文的规定，同意学校保留并向有关部门送交论文的复印件和电子版本，允许论文被查阅和借阅，同意学校将论文加入《中国优秀博硕士学位论文全文数据库》和编入《中国知识资源总库》。本人授权哈尔滨工业大学，可以采用影印、缩印或其他复制手段保存论文，可以公布论文的全部或部分内容。

本学位论文属于（请在以上相应方框内打“√”）：

保密□，在　　年解密后适用本授权书

不保密☑

作者签名：　　日期：2011 年 6 月 30 日

导师签名：　　日期：2011 年 6 月 30 日

致　谢

值此论文完成之际，谨向在本人攻读硕士学位期间给予过我帮助和关怀的所有老师、同学和亲人致以深深的谢意。

首先衷心感谢我的导师喻明艳教授，他在课题和工作上给予了我很大的帮助。喻老师学识渊博、治学严谨，在专业领域的诸多建树让人十分钦佩，他忘我的敬业精神和对事业执着的追求是对我永远的鞭策和鼓励。其次，感谢王进祥教授，他对我在实验室的学习和生活给了许多有指导性的意见，在我整个硕士期间一直关注着我的学习，督促和鼓励我努力奋进。

特别感谢付方发老师对我的关心和指导。付老师直接指导了我的硕士课题，正是在他的帮助下我才能顺利完成研究工作。他具有敏锐的洞察力，总是及时为我指出研究的方向，他在学习和生活上如同兄长一样给予我耐心的帮助和无私的关怀。

感谢微电子中心的肖立伊老师、来逢昌老师、李晓明老师、桑胜田老师，他们帮助和支持我度过硕士生涯。感谢杨兵师兄和苟鹏飞师兄，在我的课题遇到困难时，他们耐心帮我研究并悉心指导，给了我很多指导性意见。他们对很多问题有着新颖且深刻的见解，使我受益匪浅。

感谢已经毕业的宋俊杰师兄和孙思月师姐，是他们带我走入 NoC 这个研究领域，在实验室学习过程中，很多问题是他们手把手教会我的，他们解决问题的精神和态度深深地感染了我，他们解决问题的方法值得我学习，他们教会了我许多让我终生受益的东西。

感谢同学李清波、王海荣、代洪光和陈达燕，他们在我完成课题的过程中给了我很多的帮助，我们在一起度过了一段快乐的时光！感谢吴子旭师兄、王良师弟，在课题的研究过程中帮我承担了许多工作。感谢杨奕、王浩驰、李丽娟、王晓禹、湛佳、张美英、徐金娜、张玉、王意达等其他所有 NoC 组成员，课题组奋发向上的团队精神将永远激励着我。

感谢研究生期间的所有同学，他们陪伴我一起度过了研究生阶段这令人难忘的时光。

最后，衷心感谢我的家人，他们在我的一生中无论何时都给予了我最充分的支持，是我前进的最坚实基础！

衷心祝福每一位帮助过我的老师、学长、同学和亲人！