非对称多核处理器上的操作系统集成调度

陈锐忠 齐德昱 林伟伟 李 剑
(华南理工大学计算机系统研究所 广州 510006)

摘 要 相对于对称多核处理器,非对称多核处理器具有更高的效能,将成为未来并行操作系统中的主流体系结构.对于非对称多核处理器上操作系统的并行任务调度问题,现有的研究假设所有核心频率恒定,缺乏理论分析,也没有考虑算法的效能和通用性.针对该问题,该文首先建立非线性规划模型,分析得出全面考虑并行任务同步特性、核心非对称性以及核心负载的调度原则.然后,基于调度原则提出一个集成调度算法,该算法通过集成线程调度和动态电压频率调整来提高效能,并通过参数调整机制实现了算法的通用性.提出的算法是第一个在非对称多核处理器上结合线程调度和动态电压频率调整的调度算法.实际平台上的实验表明:该算法可适用于多种环境,且效能比其他同类算法高24%~50%.

关键词 绿色计算;非对称多核处理器;操作系统调度;并行任务调度;动态电压频率调整;负载均衡
中图法分类号 TP316 **DOI号**: 10.3724/SP.J.1016.2012.00616

Integrated Scheduling for Operating Systems on Asymmetric Multi-core Processors

CHEN Rui-Zhong QI De-Yu LIN Wei-Wei LI Jian
(*Institute of Computer Systems, South China University of Technology, Guangzhou* 510006)

Abstract Asymmetric multi-core processors (AMP) are more energy efficient than symmetric multi-core processors (SMP) and will be the mainstream of parallel computing architecture in the future. The existing researches on the problem of parallel task scheduling in operating systems (OS) on AMP assumed all cores have constant frequencies. They haven't analyzed the problem theoretically. These researches took neither the energy efficiency nor the universality of the scheduling into account. To solve this problem, a scheduling model based on nonlinear programming is proposed in this paper. Moreover, scheduling principles of comprehensively considering synchronization characteristics of parallel tasks, asymmetry and load of cores are analyzed and adhered. An integrated scheduling algorithm are also proposed based on the model. The algorithm integrated thread scheduling and dynamic voltage and frequency scaling (DVFS) in OS to improve energy efficiency. In addition, the algorithm achieved universality with a flexible parameter adjustment mechanism. It is the first algorithm to exploit thread scheduling and DVFS on AMP simultaneously. The evaluation on real platform demonstrates that the algorithm is universal for different conditions and it always outperforms other scheduling algorithms on asymmetric multi-core processors (by 24%~50%).

Keywords green computing; asymmetric multi-core processors; OS scheduling; parallel task scheduling; dynamic voltage and frequency scaling; load balancing

---

收稿日期:2011-08-11;最终修改稿收到日期:2012-01-20.本课题得到国家自然科学基金(61070015)、广东省中国科学院全面战略合作项目(2009B091300069)资助.**陈锐忠**,男,1985年生,博士研究生,主要研究方向为计算机体系结构、系统软件.E-mail: chen.rz02@mail.scut.edu.cn.**齐德昱**,男,1959年生,博士,教授,博士生导师,主要研究领域为计算机体系结构、软件体系结构、计算机系统安全.**林伟伟**,男,1980年生,博士,主要研究方向为计算机体系结构、分布式系统.**李 剑**,男,1976年生,博士,讲师,主要研究方向为软件工程、数据挖掘.

1 引　言

IT 行业作为全球增长最快的行业之一，其能耗也随着行业的增长而不断增长. 文献[1]指出：2008 年 IT 设备总共消耗 8680 亿度电，占全球总耗电量的 5.3%；按照目前的增长趋势，到 2025 年，IT 行业平均能耗会达到 2006 年的 5 倍. 能耗问题已成为信息系统持续发展的重大障碍. 如何提高计算机的效能，实现绿色计算，是当今的一个研究热点.

随着芯片集成规模极限的逼近以及能耗和成本等因素，多核处理器逐渐占据了市场[2]. 相对于对称多核处理器（Symmetric Multi-core Processor，SMP），单一指令集非对称多核处理器（Asymmetric Single-ISA Multi-core Processors，AMP）具有更高的效能，更符合绿色计算的要求，将成为未来的主流[3-4]. 现有操作系统调度器从单核处理器发展而来，并为 SMP 做了相应扩展，不能发挥 AMP 的效能优势. 这为操作系统调度带来了新的机遇和挑战.

随着多核技术的发展，并行程序日益普及[2,5]. 由于 AMP 上每个核心支持同一指令集结构，任务可以在不同核心上正确执行；而由于核心间的性能异构性，并行度不同的任务在不同核心上的执行效率却是不同的. 如何利用并行任务的同步特性和 AMP 的非对称性，实现高效能的操作系统调度，是该形势下的一个关键问题. 近年来有一些研究关注这一问题，但都假设所有核心频率恒定，没有建模分析影响调度的各个因素，也没有考虑算法的效能和通用性，不能很好地解决该问题. 其中：文献[6]没有考虑任务的同步特性；文献[7]需要频繁地迁移任务，从而带来巨大开销；文献[8]假设系统运行的线程总数不大于核心总数，但现实中这一假设往往难以满足.

因此，本文以效能和通用性为目标，为 AMP 上操作系统的并行任务调度问题建立了非线性规划模型，分析了任务的同步特性和核心的非对称性，得出调度应遵循 4 个原则：

(1) 同一任务的各个线程在同类核心上运行，但不在同一个核心上运行.

(2) 各核心负载均衡.

(3) 协同调度同一任务的各个线程.

(4) 使参与协同调度的各个核心频率相等.

在此基础上，本文提出一个集成调度算法，其特点如下：

(1) 集成线程调度和核心动态电压频率调整（Dynamic Voltage and Frequency Scaling，DVFS），保证 4 个调度原则，提高系统效能.

(2) 提供参数调整机制，以适应多种机器配置.

(3) 通过状态监控机制和任务集合分解降低调度开销.

据我们所知，还没有研究对该问题进行建模分析，本文是第一个在 AMP 上结合线程调度和 DVFS 的算法. 本文在 Linux 2.6.27 和多种配置的 AMD Opteron 2384 上对算法的效能、通用性和开销进行比较分析，实验证明了该算法的有效性.

本文第 2 节对问题进行描述和建模分析，给出调度的目标和原则；第 3 节详细描述集成调度算法；第 4 节对所提出的算法进行实验和比较分析；第 5 节介绍相关研究；最后是总结以及对未来工作的展望.

2 问题描述与建模

2.1 问题描述

本文研究 AMP 上操作系统的并行任务调度问题，关注的目标如下：

效能：最小化系统的 EDP（Energy Delay Product）[9]. EDP 是系统的能耗与执行效率之比，EDP 越低，系统效能越高.

通用性：适用于核心性能差异程度不同的 AMP 平台.

下面通过一个简单的例子来说明该问题. 示例中的 AMP 包含 8 个核心（如图 1，图 2）：$Core_1 \sim Core_3$ 是快核心，$Core_4 \sim Core_8$ 是慢核心，快核心的计算能力是慢核心的 2 倍. 该平台上运行 4 个任务（P_0，P_1，P_2 和 P_3），每个任务包含 2 个线程：T_1 和

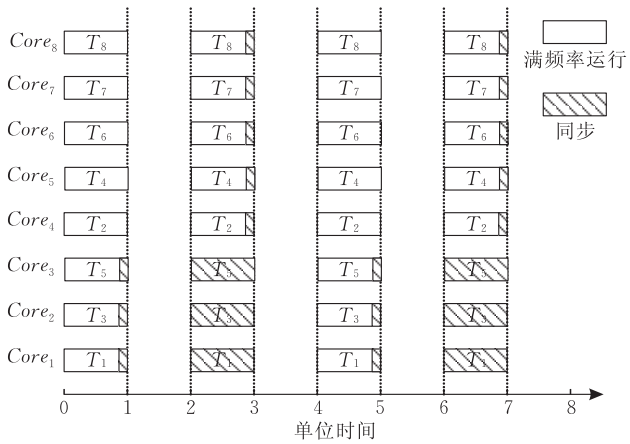


图 1　现有操作系统调度示例

T_2 属于 P_0，T_3 和 T_4 属于 P_1，T_5 和 T_6 属于 P_2，T_7 和 T_8 属于 P_3. 每个线程优先级相等，属于同一任务的 2 个线程每隔 2 个单位时间就要同步一次，然后继续运行. 文献[7]表明：由于 parallel-for、fork-join 等结构在并行编程中的广泛使用，并且程序员倾向于平衡各个并行线程的负载，同一任务的各个线程计算量趋向于相等.

图 1 给出一个现有操作系统中的线程调度示例. 现有操作系统调度器没有考虑核心的非对称性，将线程随机映射到低负载的核心上，这不符合绿色计算的要求：当属于同一任务的几个线程分配到性能不同的核心上执行时，将导致快核心上的线程等待慢核心上的线程同步的情况，此时快核心仍在消耗能量，从而降低效能（如图 2 中 T_1 和 T_2 同属于 P_0，却分别映射到异类核心 $Core_1$ 和 $Core_4$ 上，导致等待；P_1 和 P_2 亦然）.

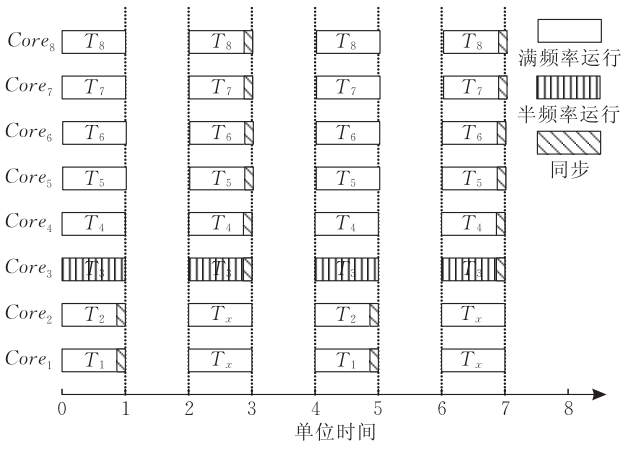


图 2 理想的操作系统调度示例

图 2 给出了理想的操作系统调度示例. (1) 它将属于同一任务的线程调度到类型相同的核心上运行，从而避免了线程空等的情况，并将节省下来的 CPU 时间用于调度其他线程（如图 2 中的 T_x），从而提高了效能. (2) 当无法保证同一任务的所有线程在同类核心上执行时，它将运行这些线程的快核心频率降低到与慢核心相等（如图 2 中的 $Core_3$），在不影响任务完成时间的情况下降低了能耗. 本文提出的调度算法实现了这种调度. 下一节将对 AMP 上操作系统的并行任务调度问题进行建模分析.

2.2 调度模型

我们可以为 AMP 上操作系统的并行任务调度问题建立非线性规划模型：

设 $T=\{T_1,T_2,\cdots,T_n\}$，$n\in N$，表示一个包含 n 个线程的任务，所有线程每隔一定时间间隔需进行同步，线程 T_i 可根据同步间隔分为 N_i 个阶段 $\{T_i^1,T_i^2,\cdots,T_i^{N_i}\}$. 每个阶段可分为计算操作和同步操作，$T_i^k.c$ 表示 T_i^k 计算操作的工作量，$T_i^k.st$ 表示 T_i^k 同步操作所需的时间. 根据文献[7]，本文假设同一任务的各个线程具有相等的计算量. $C=\{C_1,C_2,\cdots,C_n\}$，$n\in N$，表示 n 个核心的集合，第 i 个核心的单位时间计算能力为 C_i. 为了避免频繁上下文切换带来的巨大开销，我们假设式(1)成立：一个任务包含的线程数，不大于物理核心总数.

T 的调度可抽象为一个时空映射 $M=(s,t)$，其中 s 是一个空间映射，表示将 T 的各个阶段映射到核心上；t 是一个时间映射，表示将 T 的各个阶段映射到核心的时间片上. 设 $t(T_k^m)$ 表示 T_k^m 的开始时间，即 T_k^m 分配到的时间片，由任务执行的时序关系，只有所有前驱阶段都完成了，一个新阶段才能开始，即式(2)成立，其中 $\frac{T_k^m.c}{C_i}$ 为 T_k^m 的计算时间，与执行 T_k^m 的核心的性能 C_i 相关. $Time_M(T)$ 表示映射 M 下 T 的完成时间，它等于 T 最后阶段中运行最慢线程的完成时间，满足式(3). $EDP_M(T)$ 定义为系统的能耗与执行效率之比[9]，即式(4)，其中 $Energy\_Consumed$ 表示系统的总能耗，$Instructions\_per\_Second$ 表示单位时间执行的指令数，$Averge\_Power$ 表示系统的平均功率，$Total\_Instructions(T)$ 表示任务 T 指令总数. 因此我们可得该问题的非线性规划模型如下：

$$\text{Minimize } EDP_M(T)$$

$$\text{s.t.}\begin{cases} |T|\le|C| & (1)\\ \max\limits_k\left\{t(T_k^m)+\dfrac{T_k^m.c}{C_i}+T_k^m.st\right\}\le t(T_l^n) & (2)\\ \quad \text{for } 0<k,l<|T| \text{ and } m<n & \\ Time_M(T)=\max\limits_k\left\{t(T_k^{N_k})+\dfrac{T_k^{N_k}.c}{C_i}+T_k^{N_k}.st\right\} & (3)\\ EDP_M(T)=\dfrac{Energy\_Consumed}{Instructions\_per\_Second} & \\ \quad =\dfrac{Averge\_Power\times Time_M^2(T)}{Total\_Instructions(T)} & (4) \end{cases}$$

但在现实中由于缺乏 $T_i^m.c$ 和 $T_i^m.st$ 的先验知识，加上求解该问题带来的开销，无法求得该问题的最优解. 因此我们采用启发式算法来求问题的近优解. 由于式(4)中的 $Total\_Instructions(T)$ 是定值，故最小化 $EDP_M(T)$ 等价于最小化 $Averge\_Power$ 和 $Time_M(T)$. $Averge\_Power$ 可通过 DVFS 技术[10]动态调节，文献[4]预测未来的 AMP 将由少量复杂

核心(快核心)和大量简单核心(慢核心)组成，而出于成本、芯片面积等方面的考虑，简单核心很可能不具备 DVFS 功能. 故本文假设简单核心的功率固定，$Averge\_Power$ 的变化来源于复杂核心功率的变化(通过 DVFS). 下面我们研究在操作系统中通过线程调度和快核心的 DVFS 来降低 $EDP_M(T)$. 由式(2)，(3)可推出等式(5).

$$Time_M(T)=\sum_{j=1}^{p}\max_k\left\{\frac{T_k^j.c}{C_i}+T_k^j.st\right\} \quad (5)$$

其中 p 是任务 T 的阶段数目，由 T 本身属性决定，无法通过调度优化. 结合式(4)，可见 AMP 上操作系统的并行任务调度应遵循的原则如下：

(1) 同一任务的各个线程在同类核心上运行，但不在同一个核心上运行.

同类核心指性能相等的核心. 由式(2)，(3)和等式(5)可知任务完成时间取决于运行最慢的线程，如果将同一任务的各个线程放到性能不同类的核心上运行，即$\frac{T_k^m.c}{C_i}$不同时，将出现快核心上的线程等待慢核心上的线程同步的情况(如图 1 的示例). 而当同一任务的各个线程放在同一核心上运行时，任务变成串行执行，完成时间 $Time'_M(T)=\sum_{j=1}^{p}\sum_{k=1}^{|T|}\left\{\frac{T_k^j.c}{C_i}+T_k^j.st\right\}$，远大于 $Time_M(T)$. 因此调度需遵循该原则.

(2) 各核心负载均衡.

由式(2)可知任务每一阶段的完成时间取决于运行最慢的线程. 当负载不均衡时，轻负载核心将空转，而重负载核心的调度周期将延长，这降低了系统的效能，并使本阶段的 $T_k^m.st$ 和下一阶段的 $t(T_k^{m+1})$ 增大，进一步增加了运行最慢的线程的完成时间. 因此各核心应保持负载均衡.

(3) 协同调度同一任务的各个线程.

协同调度(co-schedule)指保证属于同一任务的所有线程同时运行；独立调度指调度时把线程看成一个独立实体单独运行，不考虑该任务的其他线程. 协同调度在减少线程同步时间的同时，将带来优先级反转、处理器碎片等问题[11]，增加开销. 同一任务的各个线程计算量接近[7]，并且同步较多，需要协同调度来减少 $T_k^m.st$. 串行任务可看成是单线程任务，此时协同调度等同于独立调度. 该原则在减少同步时间的同时，使没参与协同调度的核心可运行其他任务，这避免了协同调度的碎片问题[11]，提高了系统效能.

(4) 使参与协同调度的各个核心频率相等.

随着多核技术的发展，并行程序日益普及[2,5]，由于系统包括少量快核心和大量慢核心[4]，容易出现多数任务的线程数超过快核心个数的情况，此时原则(1)和(2)将无法兼顾，这无法通过纯粹的线程调度来解决. 由式(2)可知任务每一阶段的完成时间取决于运行最慢的线程. 因此当异类核心同时参与协同调度时，我们通过 DVFS 将快核心频率降低到与慢核心相等，以避免快核心线程的等待，在保证 $Time_M(T)$ 不变的同时降低 $Averge\_Power$，从而降低 $EDP_M(T)$.

3　集成调度算法

文献[4]表明：由少量快核心和大量慢核心组成的 AMP 具有很好的效能. 因此本文假设 AMP 上有两类核心：少量快核心和大量慢核心. 如何推广到 n 种核心是我们下一步研究的内容.

该算法由 4 个模块组成：状态监控机制、重调度、负载均衡、任务执行，执行模型如图 3 所示. 其中状态监控机制观察每个任务的线程数，当任务线程数发生变化时，调用负载均衡模块；当某个任务的类型(详见 3.1 节)发生变化时，调用重调度模块调整各个核心的任务队列. 任务执行模块则负责同步执行同一任务的所有线程.

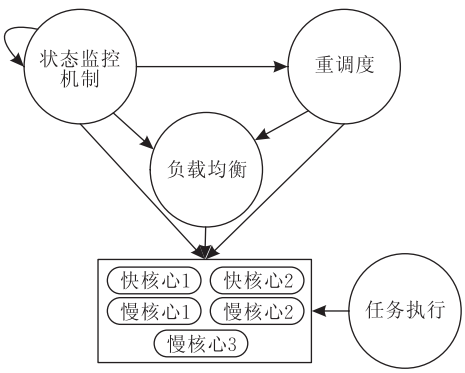


图 3　算法执行模型

3.1　状态监控机制

在运行过程中，任务的线程数将动态变化[7-8]. 本文根据任务当前运行的线程数把它们分成 3 类，定义如下：

$$\begin{aligned}FCT&=\{P\mid TLP(P)\le N(FC)\}\\SCT&=\{P\mid N(FC)<TLP(P)\le N(SC)\}\\ACT&=\{P\mid TLP(P)>N(SC)\}\end{aligned} \quad (6)$$

其中：P 表示一个任务，$TLP(P)$表示 P 当前运行的线程数；$N(S)$表示核心集合 S 包含的核心数目；FC 表示快核心集合；SC 表示慢核心集合. 根据调度原则 1)、2)和 3)，属于 FCT 的任务可以把所有线程映射到快核心上，因此适合在快核心上运行；属于 SCT 的任务线程数介于快核心数目和慢核心数目之间，没法把所有线程映射到快核心上，只好映射到慢核心上；属于 ACT 的任务线程数较大，只能同时映射到快核心和慢核心上，因此可在低负载的核心(快核心和慢核心都可以)上运行.

任务迁移需要一定的开销[6]，应尽量避免. 状态监控机制是控制任务迁移的有效手段，它在线监测各个任务当前包含的可运行线程数，当任务线程数发生变化时，它调用负载均衡模块调整核心的运行队列；当任务的线程数变化触发了类型变化时(例如从 FCT 变成 SCT)，它调用重调度模块对该任务的线程进行重新映射. 算法伪代码描述如下.

算法 1. 状态监控机制.

```
输入：任务 P
输出：无
if (任务 P 的线程数发生变化) {
  if (PreType(P)!=Type(P)) {//P 的类型发生变化
    reschedule(P);//重调度
  } else{
    balance();//负载均衡
  }
}
```

3.2 负载均衡

对于调度原则 2)，由于同类核心属于对称多核处理器，可使用现有操作系统的负载均衡算法，只需要加上调度原则 1)作为限制；本文主要关注异类核心间的负载均衡.

为了实现非对称多核处理器上的负载均衡——核心的负载与其计算能力成正比[6]，本文定义 $SF(C_i)$为核心 C_i的比例系数(*Scaled Factor*，SF)，即核心 C_i当前频率与平台最低核心频率之比. 设 $Load(C_j)$表示核心 C_j的负载(在 Linux 中可直接读取队列的成员变量 *load* 得到)，核心集合 S 的平均负载 $AvgLoad(S)$定义如式(7). 当式(8)成立时，调度原则 2)成立. 其中 α 用于调整负载均衡，α 趋向于 0 时，算法对负载均衡要求严格；α 趋向于 1 时，算法可容许负载不均衡.

$$AvgLoad(S)=\frac{\sum_{C_j\in S}Load(C_j)}{N(S)\times SF(C_i)},\ C_i\in S \quad (7)$$

$$AvgLoad(FC)\in\left[(1-\alpha)\times AvgLoad(SC),\ \frac{1}{(1-\alpha)}\times AvgLoad(SC)\right],\ \alpha\in[0,1) \quad (8)$$

负载均衡模块主要在核心负载发生变化时调用，基本思想是通过快、慢核心间的线程迁移使式(8)成立. 为了提高效率，算法对任务集合进行分解——为每种类型的核心 S(本文为 FC 和 SC)维护 3 个任务集合：$set1(S)$，$set2(S)$和 $set3(S)$. $set2(FC)=set2(SC)$，存放属于 ACT 的任务. $set1(S)$存放有线程在核心集合 S 上运行的 SCT 类的任务，$set3(S)$存放有线程在核心集合 S 上运行的 FCT 类的任务. 当快核心负载过高时，算法按 $set1(FC)\rightarrow set2(FC)\rightarrow set3(FC)$的次序选择任务迁移到慢核心上；当慢核心负载过高时，算法按 $set3(SC)\rightarrow set2(SC)\rightarrow set1(SC)$的次序选择任务迁移到快核心上. 算法伪代码描述如下.

算法 2. 负载均衡.

```
输入：无
输出：无
if (AvgLoad(FC)>AvgLoad(SC)/(1-α)) {
    //快核心负载过高
  while (AvgLoad(FC) >AvgLoad(SC)/(1-α)) {
    if (set1(FC)非空) {
      P=set1(FC)中的某个任务；
      AC=SC 中 TLP(P)个负载最轻的核心；
    } else if (set2(FC)非空) {
      P=set2(FC)中的某个任务；
      AC=所有核心中 TLP(P)个负载最轻的核心；
    } else if (set3(FC)非空) {
      P=set3(FC)中的某个任务；
      AC=SC 中 TLP(P)个负载最轻的核心；
    }
    将P 包含的可运行线程分别迁移到 AC 的每个核心上；
    更新 AvgLoad(FC)和 AvgLoad(SC)；
  }
} else if (AvgLoad(FC) <AvgLoad(SC) * (1-α)) {
    //慢核心负载过高
  while (AvgLoad(FC) <AvgLoad(SC) * (1-α)) {
    if (set3(SC)非空) {
      P=set3(SC)中的某个任务；
      AC=FC 中 TLP(P)个负载最轻的核心；
    } else if (set2(SC)非空) {
      P=set2(SC)中的某个任务；
      AC=所有核心中 TLP(P)个负载最轻的核心；
    } else if (set1(SC)非空) {
      P=set1(SC)中的某个任务；
      AC=所有核心中 TLP(P)个负载最轻的核心；
    }
```

```
    将P 包含的可运行线程分别迁移到 AC 的每个核心上；
    更新 AvgLoad(FC)和 AvgLoad(SC)；
  }
}
```

3.3 重调度

根据调度原则 1)和 2)，重调度模块的基本思想是在兼顾负载均衡的情况下，将属于 FCT 的任务的各个线程映射到快核心上，将属于 SCT 的任务的各个线程映射到慢核心上，将属于 ACT 的任务的各个线程映射到低负载的核心上. 算法伪代码描述如下.

算法 3. 重调度.

```
输入：任务 P
输出：无
将 P 的线程移出各核心的运行队列，并更新AvgLoad(FC)
      和 AvgLoad(SC)；
if (Type(P) =FCT) {
   AC=FC 中 TLP(P)个负载最轻的核心；
} elseif (Type(P) =SCT) {
   AC=SC 中 TLP(P)个负载最轻的核心；
} else {
   AC=所有核心中 TLP(P)个负载最轻的核心；
}
将P 包含的可运行线程分别映射到 AC 的每个核心上；
更新 AvgLoad(FC)和 AvgLoad(SC)；
balance();//负载均衡
```

3.4 任务执行

我们将参与协同调度的核心分成发起者和协作者. 设 $CT(C_i)$ 表示核心 C_i 正在运行的线程，算法操作如下：若 $CT(C_i)$ 属于一个多线程程序，C_i 成为发起者，发送核心间中断到其他核心，收到中断的核心(成为协作者)将执行与 $CT(C_i)$ 同属一个任务的其他线程；否则 $CT(C_i)$ 单独运行. 根据调度原则 4)，当快、慢核心同时参与协同调度时，算法将快核心频率降低到与慢核心相等. 当 $TLP(CT(C_i))$ 小于核心总数时，没参与协同调度的核心可运行其他任务，这避免了协同调度的碎片问题[11]，提高了系统效能. 算法伪代码描述如下.

算法 4. 协同调度的发起者及独立调度.

```
输入：核心 Ci
输出：无
if (CT(Ci)属于多线程程序) {
 if (Ci∈FC) {//根据调度原则 4)调整核心频率
  if (Type(CT(Ci)) !=FCT) {
    将 Ci的频率降低到与慢核心相等；
  } elseif (Ci频率与慢核心相等) {
    将 Ci的频率重置回与快核心相等；
  }
 }
 Threads=与 CT(Ci)同属一个任务的线程集合；
 cores=Threads 所在的核心集合；
 for(cores 中的每个核心 C) {
   //发送核心间中断到 cores，以同步执行 Threads；
    coordinate(C, Threads);//详见算法 5
 }
} else {
  CT(Ci)在 Ci上执行；
}
```

算法 5. 协作者.

```
输入：核心 C，线程集 Threads
输出：无
T=C 上属于 Threads 的线程；
if (C ∈FC) {//根据调度原则 4)调整核心频率
 if (Type(T) ! =FCT) {
   将 C 的频率降低到与慢核心相等；
 } elseif (C 频率与慢核心相等) {
   将 C 的频率重置回与快核心相等；
 }
}
在 C 上执行 T；
```

3.5 参数调整机制

如前所述，AMP 上的操作系统调度需要综合考虑核心性能、核心负载、任务并行性等因素，其中核心性能和负载在不同架构的处理器上优先级不同. 为了使算法在各种核心性能差异程度不同的平台上都获得高效能，需提供优先级调整的机制. 算法提供了参数 α 用于调整优先级(见式(8)). 参数应根据实际情况设置，比如对于核心性能差异很大的 AMP，核心性能优先级较高，α 应取接近 1 的数；对于核心性能差异较小的 AMP，α 应取接近 0 的数，以通过负载均衡提高效能.

3.6 算法运行开销

设 M 表示任务总数，N 表示核心总数，算法需要为每类核心维护任务集合，空间复杂度为 $O(M)$. 算法的时间开销主要来自以下 3 个方面：

(1) 负载均衡. 该模块时间复杂度为 $O(MN)$，但现实中基本只需要迁移一两个任务即可实现负载均衡，并且任务集合分解提高了线程迁移效率，因此该模块的平均时间复杂度接近 $O(N)$.

(2) 重调度. 该模块时间复杂度为 $O(N)$.

(3) 任务执行. 协同调度在减少线程同步时间的同时，将带来额外的上下文切换. 不过算法采用了协同调度和独立调度相结合的方法，使没参加协同调度的核心可以运行其他任务，这避免了碎片问题.

此外，状态监控机制有效避免了不必要的负载均衡和重调度，特别是重调度只在任务类型发生变化时使用，因此算法的开销并不大. 这将在 4.2.2 节的实验中得到进一步验证.

4 实验与分析

4.1 实验平台与方法

本章将集成调度算法(Integrated Algorithm, IA)与 Age based[7]、PA[8]、FF[6]和 Linux 自带的调度器[12](Completely Fair Scheduler, CFS)进行比较.其中 CFS 没有对处理器的非对称性做处理.

本文采用 Linux 2.6.27 来实现和运行上述各个算法.

本文的实验平台是一台 2 路 AMD Opteron 2384 服务器.AMD Opteron 2384 是对称多核处理器,包含 4 个 2.7GHz 的核心.本文用 Linux 提供的 cpufreq governors 调整各个核心的频率,以体现非对称性.本文使用了 3 种配置,如表 1 所示.其中 Conf1 和 Conf2 是 AMP,用于测试算法的效能、通用性和参数灵敏度;Conf3 是 SMP,用于测试算法的开销.α 的取值结合测试平台配置和经验确定,α 取其它值时算法的表现详见 4.2.3 节.

测试程序选自 PARSEC[13].由于测试平台包含 2 个快核心和 6 个慢核心,为了更加全面地比较各个算法在不同平台配置下的效能,我们设计了多种类型的并行程序组成的测试程序集,如表 2 所示.为了使负载均衡机制生效,我们使每个程序集的线程总数大于平台的核心总数.测试程序后括号中的数字 n 表示该程序包含 n 个线程.根据研究文献[6,8]的实验方法,我们把每个程序集在每个调度算法下分别运行 3 次,取相应指标的平均值作为度量;实验中我们发现每次运行结果差异不大,每个算法都体现较好的稳定性,因此我们没有对稳定性进行单独分析.

表 1 测试平台配置

平台	α	描述
Conf1	0.3	2 个核心运行在 2.7GHz 下,另外 6 个核心运行在 1.5GHz 下.
Conf2	0.1	2 个核心运行在 2.7GHz 下,另外 6 个核心运行在 2.0GHz 下.
Conf3	0	8 个核心都运行在 2.7GHz 下.

表 2 测试程序集

程序集	测试程序
W1	canneal(2), freqmine(2), blackscholes(6), dedup(8)
W2	x264(2), bodytrack(3), blackscholes(5), dedup(8)
W3	fulidanimate(4), facesim(6), swaptions(8)
W4	canneal(2), fulidanimate(3), streamcluster(6), ferret(7)
W5	freqmine(2), blackscholes(8), swaptions(8)

4.2 实验结果与分析

4.2.1 效能与通用性分析

每个程序集在不同算法上的 *EDP* 比较如图 4、图 5 所示.为了便于比较实验结果,本文以 CFS 的 *EDP* 为基准对数据进行归一化处理,小于 1 表示 *EDP* 小于 CFS,否则反之.

图 4 给出了 Conf1 配置下各算法的相对 *EDP* 比较.由于快、慢核心的频率相差近 1 倍,核心性能差异对效能影响较大,核心负载次之,我们设定 $\alpha=0.3$.结果显示,IA 的 *EDP* 比 CFS 低 43%～50%,也比 Age based 低 24%～31%;*EDP* 上 Age based 比 CFS 低 22%～34%,PA 比 CFS 低 17%～26%,FF 比 CFS 低 17%～23%.

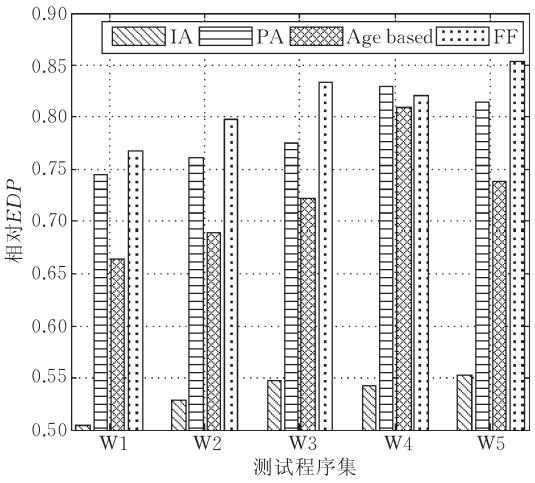


图 4 Conf1 上各程序集的相对 *EDP*

图 5 给出了 Conf2 配置下各算法的相对 *EDP* 比较.由于快、慢核心的频率相差 0.7GHz,核心负载对效能的影响大于 Conf1,我们设定 $\alpha=0.1$.结果显示,IA 的 *EDP* 比 CFS 低 43%～48%,也比 Age based 低 27%～33%;相对于 CFS,Age based 能把 *EDP* 降低 16%～23%,PA 能降低 10%～17%,FF 能降低 10%～16%.

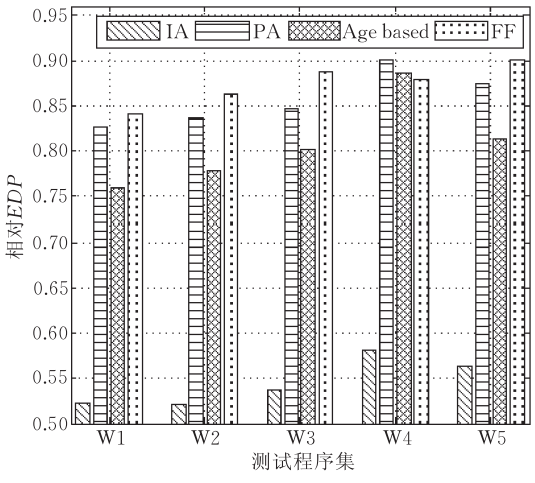


图 5 Conf2 上各程序集的相对 *EDP*

相对于没有针对 AMP 特性做处理的 CFS,IA、

PA、Age based、FF 的效能都有不同程度的提高. 这说明高效能的操作系统调度必须利用核心的特性. 各算法按效能从高到低排序为：IA，Age based，PA，FF，CFS. 其中 Age based 需要频繁地把进度最慢的线程迁移到快核心上，当处理存储密集型任务（如 W4）时，将带来巨大的开销，使其效能低于 PA. 相对于其他算法，IA 在每个程序集上都有明显的效能优势，这是因为它保证了 4 个调度原则，综合考虑核心非对称性、核心负载和任务同步特性进行线程映射和核心频率调整.

当核心性能差异减小（如 Conf2）时，PA、Age based、FF 相对于 CFS 的效能优势有所下降，而 IA 仍保持着 43%～48%的 *EDP* 降幅. 这得益于 IA 的参数调整机制，使其能根据不同环境调节核心性能差异和负载的优先级，从而在每个环境下都有效能提高. 这说明了 IA 的通用性.

4.2.2　开销分析

该实验运行在 Conf3 平台上. Conf3 是 SMP，我们使各个算法假设仍在有 2 个快核心和 6 个慢核心的平台上运行，以此来分析各个 AMP 上的调度算法相对于 CFS 带来的额外开销. 测试程序在每个调度算法下分别运行 3 次，取完成时间的平均值作为度量. 当其中某个程序提前完成时，我们让其重新运行，以保持测试环境的稳定性. 为了便于比较实验结果，本文以 CFS 的完成时间为基准对数据进行归一化处理，超过 1 表示算法的开销.

实验结果如图 6 所示，包括每个程序的相对完成时间以及每个测试集的平均相对完成时间（图 6 中的 geo-mean 列）. 各算法按完成时间从小到大排序为：IA，CFS，FF，PA，Age based. 其中 IA 的开销很小，因为状态监控机制有效限制了线程迁移，任务集合分解提高了线程迁移效率，并且协同调度带来的收益大于其开销——即使在 SMP 上，完成时间相对于 CFS 的降幅可达到 17%. Age based、PA 和 FF 没有协同调度，因此在 SMP 上完成时间都大于 CFS. FF 的开销主要来源于当快核心空闲时，将线程从慢核心迁移到快核心上，在各核心负载均衡的情况下，这种迁移并不多，因此开销略大于 CFS. PA 则由任务线程数的改变触发任务迁移，开销处于 FF 和 Age based 之间. 由于 Age based 需要频繁地将剩余时间最长的线程迁移到快核心上运行，它的开销最大，特别是对于存储密集型任务（如 W4）. 这验证了 3.6 节的算法开销分析，也进一步解释了各算法在 4.2.1 节中的效能表现.

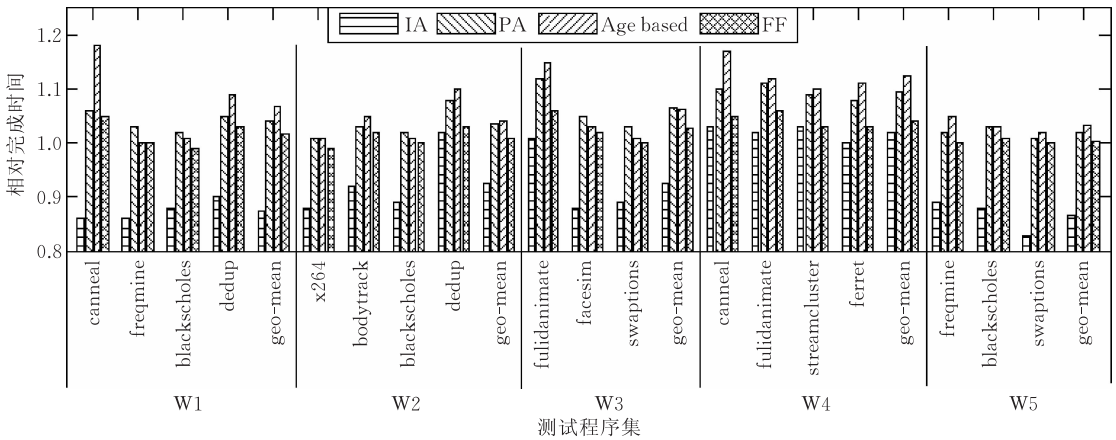


图 6　Conf3 上各程序的相对完成时间

4.2.3　参数灵敏度分析

该实验分析 α 的取值对 IA 效能的影响. 图 7，8 分别给出 Conf1 和 Conf2 配置下各程序集实际 *EDP* 随 α 取值的变化. 可见 Conf1 配置下 IA 在 $\alpha=0.3$ 时取得最优效能，Conf2 配置下 IA 在 $\alpha=0.1$ 时取得最优效能. 不同平台上，核心性能和负载的优先级不同，要取得好的效能需要平衡两者；在 Conf1（Conf2）平台上，$\alpha=0.3$（0.1）正好平衡了这两个因素. 如图 7、8 所示，当 α 取值过大或过小时，效能都会降低. 由此可见，IA 提供的参数调整机制有效提高了算法的通用性和灵活性.

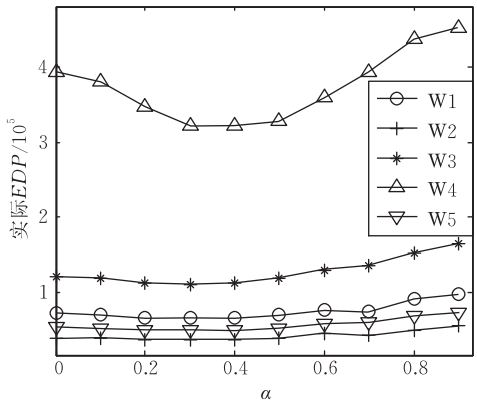


图 7　Conf1 上各程序集的实际 *EDP* 变化

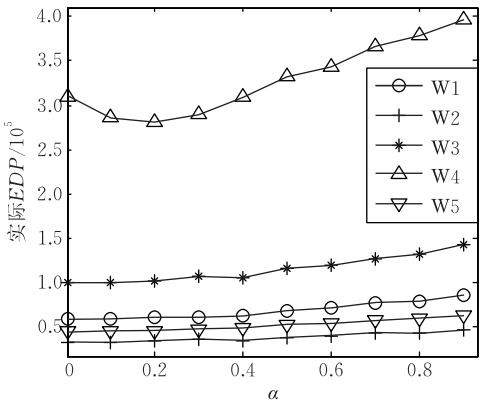


图 8 Conf2 上各程序集的实际 *EDP* 变化

4.3 实验小结

基于以上实验，我们可以得出以下结论：

(1) 效能. 相对于其他调度算法，集成调度算法具有明显的优势，能把系统的 *EDP* 降低 24%～50%.

(2) 通用性. 当核心性能差异减小时，其他 AMP 上的调度算法的效能优势有所下降，而集成调度算法仍保持着 43%～48%的 *EDP* 降幅，这说明该算法具有很好的通用性.

(3) 开销. 集成调度算法的开销优于其它算法，而协同调度的开销远小于其收益，即使在 SMP 上仍有良好的性能表现.

5 相关工作

对于操作系统调度问题，之前的研究主要集中于 SMP，这类平台上的调度算法主要是处理公平性、负载均衡[14]等，没有对 AMP 的非对称性做处理，无法发挥其优势. 对于 AMP 上的调度问题，之前的方法[15-16]都依赖于任务执行时间等参数已知，但在操作系统中这些参数无法直接获得. 因此，这些方法无法解决操作系统调度问题.

本文讨论的 AMP 上操作系统的并行任务调度问题，近年来也有一些研究工作. 其中文献[6]在保证核心负载与其计算能力成正比的同时，优先使用快核心，并对 NUMA 节点间的线程迁移做限制. 该方法利用了 AMP 的特性，但没有考虑并行任务的同步特性，从而影响效能. 文献[7]假设同一任务的所有线程运行时间相等，把剩余时间长的线程优先调度到快核心上运行. 该方法利用了 AMP 的非对称性和并行任务的同步特性，但容易造成频繁的线程迁移，从而带来巨大开销. 而本文的状态监控机制和任务集合分解有效控制了调度开销.

跟本文比较相关的是文献[8]，其提出的 PA 算法按照线程级并行度将任务分成 3 类：MP 为并行度不大于快核心数目的任务；HP 为并行度大于快核心数目的任务；SP 为 HP 类任务的串行阶段. 任务按使用快核心的优先级从高到底排列为：SP，MP，HP. 该算法利用了 AMP 和并行任务的特性，开销也比较小. 但本文和 PA 算法存在以下不同：

(1) PA 假设系统运行的线程总数不大于核心总数；本文则假设一个任务的线程总数不大于核心总数，这大大放宽了 PA 的限制.

(2) PA 没有同步执行同一任务的线程，这将导致线程间的互相等待；本文则协同调度同一任务的所有线程.

并且文献[6-8]都是针对系统性能，假设所有核心频率固定，没有考虑算法的通用性，也没有对 AMP 上操作系统的并行任务调度问题进行建模分析. 本文以效能和通用性为目标为该问题建立了非线性规划模型，结合线程调度和 DVFS 有效降低了系统的 *EDP*.

另外，有一些研究从不同于本文的角度研究 AMP 上的操作系统调度问题，文献[17]将计算密集型的任务调度到快核心上运行，将存储密集型的任务调度到慢核心上运行. 这些方法针对串行任务，没有考虑并行任务的特性；而随着多核技术的发展，并行程序日益普及[2,5]，并行任务的特性是操作系统调度必须考虑的. 本文的方法和这些研究并不冲突，可以互相结合，文献[18]朝这个方向做了尝试，这也是我们下一步研究的内容.

6 结论与展望

本文对 AMP 上操作系统的并行任务调度问题进行研究，建立了非线性规划模型，分析得出 4 个调度原则，并基于调度原则提出了集成调度算法. 据我们所知，本文是第一个对该问题进行建模分析的研究，提出的算法是第一个在 AMP 上结合线程调度和 DVFS 的调度算法. 实际平台上全面的对比实验表明：集成调度算法的效能、通用性和开销都优于其它同类算法. 集成调度算法的效能优势来源于它综合利用了核心非对称性、核心负载和任务同步特性以及对线程调度和 DVFS 的有效集成，这也验证了调度原则的有效性；通用性来源于参数调整机制，它可灵活调节核心负载和非对称性的优先级，从而适

用于核心性能差异不同的多种机器配置；开销优势是因为状态监控机制控制了线程迁移，任务集合分解提高了迁移效率，并且协同调度带来的收益大于其开销.

如何控制系统温度，以降低制冷设备的能耗，也是绿色计算的一个关键问题. 研究操作系统调度对AMP温度的影响，将是我们下一步的工作重点.

参 考 文 献

[1] Lin Chuang, Tian Yuan, Yao Min. Green network and green evaluation: mechanism, modeling and evaluation. Chinese Journal of Computers, 2011, 34(4): 593-612 (in Chinese)
(林闯，田源，姚敏. 绿色网络和绿色评价：节能机制、模型和评价. 计算机学报，2011，34(4)：593-612)

[2] Chen Guo-Liang, Sun Guang-Zhong, Xu Yun et al. Integrated research of parallel computing: status and future. Chinese Science Bulletin, 2009, 54(8): 1043-1049(in Chinese)
(陈国良，孙广中，徐云，等. 并行计算的一体化研究现状与发展趋势. 科学通报，2009，54(8)：1043-1049)

[3] Hill M D, Marty M R. Amdahl's law in the multicore era. IEEE Computer, 2008, 41(7): 33-38

[4] Fedorova A, Saez J C, Shelepov D et al. Maximizing power efficiency with asymmetric multicore systems. Communications of the ACM, 2009, 52(12): 48-57

[5] Manferdelli J L, Govindaraju N K, Crall C. Challenges and opportunities in many-core computing. Proceedings of the IEEE, 2008, 96(5): 808-815

[6] Li T, Baumberger D, Koufaty D A et al. Efficient operating system scheduling for performance-asymmetric multi-core architectures//Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. Reno, USA, 2007: 1-11

[7] Lakshminarayana N B, Jaekyu L, Kim H. Age based scheduling for asymmetric multiprocessors//Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. Portland, USA, 2009: 1-12

[8] Saez J C, Fedorova A, Prieto M et al. Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors//Proceedings of the 7th ACM International Conference on Computing Frontiers. Bertinoro, Italy, 2010: 31-40

[9] Gonzalez R, Horowitz M. Energy dissipation in general purpose microprocessors. IEEE Journal of Solid-State Circuits, 1996, 31(9): 1277-1284

[10] Rangan K K, Wei G Y, Brooks D. Thread motion: fine-grained power management for multi-core systems//Proceedings of the 36st Annual International Symposium on Computer Architecture. Austin, USA, 2009: 302-313

[11] Lee W, Frank M, Lee V et al. Implications of I/O for gang scheduled workloads//Proceedings of the Job Scheduling Strategies for Parallel Processing. Geneva, Switzerland, 1997: 215-237

[12] Mauerer W. Professional linux kernel architecture. Hoboken: John Wiley & Sons, 2008

[13] Bienia C. Benchmarking modern multiprocessors [D]. Princeton: Princeton University, 2011

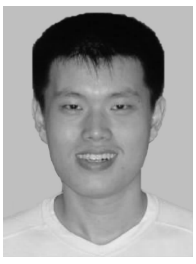
[14] Hofmeyr S, Iancu C, Blagojevic F. Load balancing on speed//Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming. Bangalore, India, 2010: 147-157

[15] Chen J, John L K. Efficient program scheduling for heterogeneous multi-core processors//Proceedings of the 46th Annual Design Automation Conference. San Francisco, USA, 2009: 927-930

[16] Chen J, John L K. Energy-aware application scheduling on a heterogeneous multi-core system//Proceedings of the 2008 IEEE International Symposium on Workload Characterization. Seattle, USA, 2008: 5-13

[17] Saez J C, Shelepov D, Fedorova A et al. Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. Journal of Parallel and Distributed Computing, 2011, 71(1): 114-131

[18] Saez J C, Prieto M, Fedorova A et al. A comprehensive scheduler for asymmetric multicore systems//Proceedings of the 5th European Conference on Computer Systems. New York, USA, 2010: 139-152



CHEN Rui-Zhong, born in 1985, Ph. D. candidate. His research interests focus on computer architecture and system software.

QI De-Yu, born in 1959, Ph. D., professor, Ph. D. supervisor. His research interests include computer architecture, software architecture and computer system security.

LIN Wei-Wei, born in 1980, Ph. D.. His research interests include computer architecture and distributed system.

LI Jian, born in 1976, Ph. D., lecturer. His research interests include software engineering and data mining.

Background

Energy efficiency is increasingly important for future information and communication technologies. Employing asymmetry in multi-core processor design is demonstrated to be an effective approach toward green computing. By integrating

different types of cores in a single chip, the asymmetric multi-core processors (AMP) provide the architectural capability to accommodate the diverse computational requirements of the applications. With the increasing popularity of parallel programming and AMP, how to schedule the parallel tasks in operating systems (OS) on AMP efficiently is a key problem. The widely used OS, such as Linux and Windows, are designed for symmetric multi-core processors (SMP) and can't exploit the potential of AMP. The existing researches on the problem of parallel task scheduling in OS on AMP assume constant frequency for all cores and haven't analyzed the problem theoretically. Neither have they taken the energy efficiency or universality of scheduling algorithm into account.

Therefore, the paper analyzes this problem theoretically and concludes some scheduling principles comprehensively considering synchronization characteristics of parallel tasks, asymmetry and load of cores. Based on the scheduling principles, an integrated scheduling algorithm is also proposed. It is the first algorithm to simultaneously exploit thread scheduling and DVFS on AMP. The evaluation on real platforms demonstrates that the algorithm achieves universal scheduling and it always outperforms other scheduling algorithms on AMP (by 24%～50%).

This work researches on the problem of parallel task scheduling in OS on AMP. It is a part of our projects on cloud computing and computer architecture. These projects are supported by the National Science Foundation of China under Grant No. 61070015 and the Comprehensive Strategic Cooperation Fund of Guangdong Province and Chinese Academy of Science under Grant No. 2009B091300069.

Our past work includes dynamic resource providing and task scheduling in cloud computing environment. The works have been published in Journal of Software, Journal of Computer Research and Development and so on. We will continue our work on OS scheduling on AMP.