计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220700
60(5): 976−991, 2023

CPU-GPU MPSoC 中使用寿命驱动的 OpenCL 应用调度方法

曹 坤[1] 龙赛琴[1] 李哲涛[1,2]

[1](暨南大学信息科学技术学院 广州 510632)

[2](网络安全检测与防护国家地方联合工程中心(暨南大学) 广州 510632)

(kuncao@jnu.edu.cn)

Lifetime-Driven OpenCL Application Scheduling on CPU-GPU MPSoC

Cao Kun[1], Long Saiqin[1], and Li Zhetao[1,2]

[1](*College of Information Science and Technology*, *Jinan University*, *Guangzhou* 510632)

[2](*National Joint Engineering Research Center of Network Security Detection and Protection Technology*(*Jinan University*), *Guangzhou* 510632)

Abstract In recent years, multiprocessor system-on-chips (MPSoC) integrating CPU and GPU have been widely deployed in the fields of industrial control, automotive electronics, smart medical, etc. The open computing language (OpenCL) is regarded as a popular application programming standard for CPU-GPU MPSoC due to the power of fully exploiting the parallel computing power of GPU cores and the general-purpose computing power of CPU cores. However, during deploying OpenCL applications to CPU-GPU MPSoC, most of the existing research works have neglected the management of chip temperature and lifetime, resulting in the elevated peak temperature and the early occurrence of permanent failures. In this paper, we explore the lifetime-driven OpenCL application scheduling for latency minimization on CPU-GPU MPSoC under timing, temperature, energy consumption, and lifetime constraints. We propose a method composed of static and dynamic application scheduling techniques. The static application scheduling technique is built on the improved cross-entropy strategy with consideration of the OpenCL application characteristics in searching for optimal OpenCL application design points. The dynamic application scheduling technique is developed on the feedback control strategy capable of processing the new arrival applications for latency optimization at runtime. Experimental results show that our proposed method reduces the average delay of OpenCL applications by 34.58% while satisfying all design constraints.

Key words CPU-GPU MPSoC; latency; lifetime; OpenCL application; scheduling; temperature

摘 要 近年来,集成 CPU 和 GPU 的多处理器片上系统(multiprocessor system-on-chips, MPSoC),凭借兼顾 GPU 核心的并行计算能力和 CPU 核心的通用计算能力,已经广泛应用于工业控制、汽车电子、智慧医疗等领域.为了充分发挥 CPU-GPU MPSoC 的性能,开放计算语言(open computing language, OpenCL)逐渐成为一种主流的应用程序编写标准.然而,在将 OpenCL 应用部署到 CPU-GPU MPSoC 的过程中,现有研究工作大多忽略了对芯片温度和使用寿命的管理,导致处理器核心在执行应用时超过了峰值温度,甚至

收稿日期: 2022-08-08; **修回日期**: 2023-04-04

基金项目: 国家自然科学基金项目(62102164, 62172350, 62032020); 国家重点研发计划项目(2021YFB3101201); 中国博士后科学基金项目(2021T140272, 2021M691240); 广州市科技计划项目(202201010573); 中央高校基本科研业务费专项资金(21621025)

This work was supported by the National Natural Science Foundation of China (62102164, 62172350, 62032020), the National Key Research and Development Program of China (2021YFB3101201), the China Postdoctoral Science Foundation (2021T140272, 2021M691240), the Science and Technology Projects in Guangzhou (202201010573), and the Fundamental Research Funds for the Central Universities (21621025).

通信作者: 龙赛琴(saiqinlong@jnu.edu.cn)

永久性故障的提前发生，无法保证 OpenCL 应用的长久稳定运行. 为了弥补上述缺点，提出了一种包含静态和动态应用调度技术的方法. 静态应用调度技术是基于改进交叉熵策略，将 OpenCL 应用的特性充分考虑在内，有效提高了 OpenCL 应用设计点的寻优效率. 动态应用调度技术是基于反馈控制策略，克服了传统方案中无法有效应对系统运行时新到应用的缺陷，能够最小化新到应用的平均延迟. 实验表明，所提方法可以将应用的平均延迟降低 34.58%，同时满足温度、能耗、使用寿命的约束.

关键词 CPU-GPU 多处理器片上系统；延迟；寿命；OpenCL 应用；调度；温度

中图法分类号 TP391

近年来，随着半导体技术的快速进步和对应用性能需求的不断增加，多处理器已经取代单处理器，成为当前和未来处理器的主流设计范式. 在多处理器设计方法中，集成 CPU 和 GPU 的多处理器片上系统（multiprocessor system-on-chips, MPSoC）受到广泛关注[1]. 例如，三星 Exynos 5422 MPSoC [2] 集成了 4 个 ARM Cortex A15 CPU 核心、4 个 ARM Cortex A7 CPU 核心、1 个 ARM Mali-T628 MP6 GPU. 这类 MPSoC 可以充分发挥 GPU 核心的并行计算能力和 CPU 核心的通用计算能力，从而满足各种新兴应用的低延迟、低开销、节能等需求.

为了充分发挥 CPU-GPU MPSoC 的性能，开放计算语言[3-5]（open computing language, OpenCL）逐渐成为一种极具吸引力的应用程序编写标准. OpenCL 支持多级别的线程并行化，可以将应用高效地映射到同构或异构、单个或多个 CPU 和 GPU 核心. 然而，在执行 OpenCL 应用时，如何确定 CPU 和 GPU 的工作负载，充分发挥线程级和数据级并行化优势，一直以来是一个研究难点和重点. 图 1 为 OpenCL 应用的延迟和能耗随 CPU 负载变化的示意图. 如图 1 所示，在执行特定的 OpenCL 应用时，存在使该应用获得最优性能的 CPU 负载，当分配更多的负载给 CPU 核心时，应用的性能不会进一步提升，反而会产生额外的能

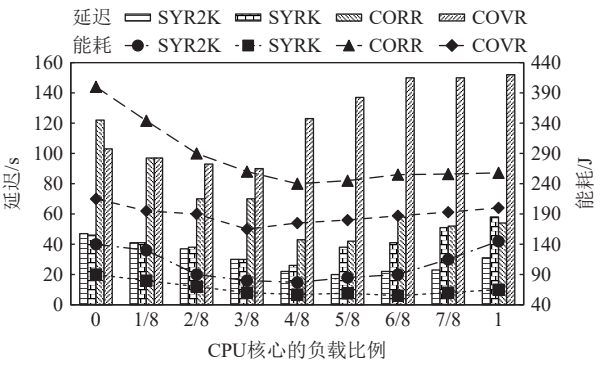


Fig. 1 Performance of OpenCL applications under varied CPU workload fractions [4]

图 1 OpenCL 应用在不同 CPU 负载比例下的性能[4]

量和延迟开销. 因此，在设计 OpenCL 应用调度算法时，需要考虑如何将 OpenCL 应用的负载合理地分配给 CPU 和 GPU 核心.

现有的 OpenCL 应用调度技术[6-14] 大多面向系统静态阶段给定的初始应用，旨在搜索最佳的负载划分点和负载到处理器核心的映射，以平衡 OpenCL 应用的延迟和系统的能耗. 对于此类技术，由于应用的指令条数、截止期限、周期等关键参数均在系统静态阶段已知，因此可设计复杂的离线算法，实现降低延迟和优化能耗的目的. 基于静态调度方案，也有部分工作[10-14] 根据系统运行时的资源利用情况，对初始应用的静态调度表进行微调，以提高应用调度算法的灵活性. 然而，这些技术[6-14] 均无法有效应对在系统运行时的新到应用. 这主要是因为在真实场景中存在事件触发型应用，例如视频监控场景中的人体追踪应用，只有捕捉到人体活动时才会触发后继子任务的执行. 考虑到事件的猝发性，事件触发型应用的到达时间在系统静态阶段是未知的. 此外，由于事件持续时间的随机性导致采集到与该事件相关的数据总量只有在系统运行时才能确定，而应用需要执行的指令总条数通常依赖于输入数据总量，输入数据总量越多，执行的指令总条数也越多. 显然，事件触发型应用执行的指令总条数只有在系统运行时才能确定. 如果直接采用现有技术[6-14] 处理在系统运行时的新到应用，将不可避免地带来极大的调度决策时间开销，无法确保满足应用的时序要求. 尽管文献 [15] 探究了如何在系统运行时处理新到应用，然而却忽视了对应用时序、系统温度、系统使用寿命的管理.

随着芯片制造技术的快速发展，芯片功率密度呈指数级增长，进而导致芯片温度不断升高.如果芯片工作温度过高，系统将出现功能错误、可靠性低、永久损坏等问题[16-17].因此，温度管理一直是异构计算系统中一个重要而紧迫的研究课题，特别是对于配备有限冷却技术的 CPU-GPU MPSoC，迫切需要使用有效的热量管理技术，以实现高性能计算，同时将芯

片的峰值温度保持在指定的温度范围内.然而，当前将 OpenCL 应用部署到 CPU-GPU MPSoC 的研究工作，往往忽略了对芯片温度和使用寿命的管理，导致处理器核心在执行应用时超过了峰值温度，甚至永久性故障的提前发生，无法保证 OpenCL 应用在 CPU-GPU MPSoC 的长久运行.为了弥补现有技术的缺陷，本文将电迁移(electromigration, EM)、经时击穿(time dependent dielectric breakdown, TDDB)、热循环(thermal cycling, TC)三种引起多核处理器系统失效的主要机制考虑在内，旨在降低 OpenCL 应用延迟的同时，有效降低芯片温度和延长系统使用寿命.

本文针对 CPU-GPU MPSoC，研究了如何在满足时序、温度、能耗、使用寿命约束的前提下，优化初始和新到 OpenCL 应用的平均延迟. 本文的主要贡献包括 3 个方面：

1)针对系统静态阶段的初始应用，提出了一种基于改进交叉熵策略的调度技术，该技术将 OpenCL 应用的特性充分考虑在内，有效提高了 OpenCL 应用的设计点寻优效率；

2)针对系统动态运行阶段的新到应用，设计了一种基于反馈控制策略的调度技术，该技术能够有效降低制定动态调度决策的时间开销，同时最小化新到应用的平均延迟；

3)在真实硬件平台上进行了大量实验验证了所提调度方法的有效性，实验结果表明本文提出的方法可以将应用的平均延迟降低 34.58%，同时满足所有的设计约束.

1 相关工作

近年来，如何优化 OpenCL 应用在 CPU-GPU MPSoC 的性能逐渐成为一个研究热点. 文献 [6] 提出了一种名为 Troodon 的新型负载平衡调度启发式算法，该算法融合了基于机器学习的设备适用性模型，能够根据 OpenCL 应用的特征对其分类. 此外，Troodon 还包括一个加速预测器，该算法用来预测 OpenCL 应用在设备上执行时获得的加速比. 通过 E-OSched 调度机制，Troodon 以负载平衡的方式将应用分配给 CPU 和 GPU 核心，从而降低应用的延迟. 文献 [7] 提出了一种面向 OpenCL 应用和 CPU-GPU 集群的计算架构 FlinkCL，该架构使用 4 种技术：异构分布式抽象模型、即时编译模式、分层部分缩减策略、异构任务管理策略.

文献 [8] 探索了如何使用主动模糊学习将 OpenCL 应用映射到异构 CPU-GPU 多核架构. 在学习阶段，通过开发一个基于机器学习的设备适用性分类器来创建子样本，以准确预测哪些处理器核心被分配了过多的负载. 文献 [9] 提出了一种基于预测运行时间的 OpenCL 应用调度策略. 该策略利用了机器学习方法，可以在调度前估计单个应用程序的运行时间. 然后，根据预测结果，选择最合适的处理器核心来处理该应用，从而降低应用的运行时间. 文献 [10] 展示了一个用于优化 CPU-GPU 协同计算的框架 OPTiC，该框架通过利用一系列建模技术，可以准确估计 CPU 和 GPU 的功率、不同频率点的性能、温度和内存争用对性能的影响. 文献 [11] 介绍了一种编程模型调度器，它允许异构系统中的所有设备协同执行单个 OpenCL 应用. 该调度器引入了一种新型负载均衡算法，能够实现最优的系统资源配置.

进一步地，文献 [12] 使用线性回归技术，对 OpenCL 应用的特征和性能进行建模，在该模型基础上，利用动态电压频率调节技术，确定 CPU 和 GPU 最佳的工作负载和频率. 文献 [13] 提出了一个用于在异构多核平台上对实时 OpenCL 应用进行热感知调度的框架. 该框架将任务迁移、频率调整、空闲槽插入等多个控制动作视为任务映射参数，通过调整任务映射参数来降低违背温度约束的概率. 文献 [14] 提出了一种针对运行在 CPU-GPU 集成平台上的应用划分策略，它由探索阶段、稳定阶段、最终决策阶段构成，可以动态地将应用负载分配给 CPU 和 GPU，从而降低应用的执行时间和能耗. 文献 [15] 设计了一种跨设备控制变量自动更新策略，解决了如何使用细粒度共享虚拟内存在具有共享缓存的 CPU-GPU 架构上执行 OpenCL 应用.

表 1 对比了本文提出的方法和现有技术的特点. 由表 1 可知，本文综合考虑了应用时序、能耗、延迟、温度、系统寿命，而相关工作[6-15]未能将这些因素充分考虑在内. 本文弥补了现有工作的不足，针对系统静态阶段的初始应用，提出了一种基于改进交叉熵策略的调度方法，针对系统动态阶段的新到应用，设计了一种基于反馈控制策略的调度技术. 本文提出的调度方法能够在满足时序、温度、能耗、使用寿命约束的前提下，最小化初始和新到 OpenCL 应用的平均延迟.

2 系统模型

2.1 架构模型

考虑由 M 个 CPU 大核组成的集群 $C_{\text{big}}=\{C_{\text{big}}^{1},$

Table 1 Comparison of Our Proposed Method with Existing Techniques

表 1 本文提出的方法和现有技术的对比

方法	能耗	延迟	寿命	温度	时序	处理新到应用	主要技术
文献 [6]	×	√	×	×	×	×	机器学习、E-OSched 调度机制
文献 [7]	×	√	×	×	×	×	集成 4 种技术
文献 [8]	√	√	×	×	×	×	主动模糊学习
文献 [9]	×	√	×	×	×	×	机器学习
文献 [10]	√	√	×	√	×	×	对处理器的温度、功率、内存竞争建模
文献 [11]	√	√	×	×	×	×	负载平衡技术
文献 [12]	√	√	×	√	×	×	线性回归、动态电压频率调节技术
文献 [13]	√	√	×	√	√	×	任务迁移、频率调整、空闲槽插入
文献 [14]	√	√	×	×	×	×	启发式技术
文献 [15]	×	√	×	×	×	√	在线分析、跨设备控制变量自动更新
本文	√	√	√	√	√	√	改进的交叉熵策略、反馈控制策略

注: “√”表示将该指标纳入优化; “×”表示未将该指标纳入优化.

$C^2_{\text{big}}, \cdots, C^M_{\text{big}}\}$、$N$ 个 CPU 小核组成的集群 $C_{\text{little}} = \{C^1_{\text{little}}, C^2_{\text{little}}, \cdots, C^N_{\text{little}}\}$、1 个 GPU 核心集群构成的 MPSoC. 在该系统中, 同一个集群中的所有核心是同构的, 不同集群中的核心是异构的. 此外, 尽管目前主流的 CPU-GPU MPSoC (例如三星 Exynos 5422 芯片)配备了多个 GPU 核心, 但无法指定某个 GPU 核心来完成特定的任务. 因此, 本文采用文献 [6−15] 的假设, 将所有的 GPU 处理单元抽象为 1 个核心 C^0_{GPU}. 进一步地, 该系统中的所有核心都配备了动态电压频率调节技术, 支持一组离散的工作电压和频率组合. 但是, 对于同一个集群中的核心, 它们必须以相同的频率执行负载, 不允许仅提高或降低单个核心的工作频率. 当单个核心没有继续执行的负载时, 会自动由运行状态转换到睡眠状态以节约能耗. 对于 CPU 大核集群 $C_{\text{big}} = \{C^1_{\text{big}}, C^2_{\text{big}}, \cdots, C^M_{\text{big}}\}$、CPU 小核集群 $C_{\text{little}} = \{C^1_{\text{little}}, C^2_{\text{little}}, \cdots, C^N_{\text{little}}\}$、GPU 核心 C^0_{GPU}, 分别用 $\Omega_{\text{big}} = \{(V^j_{\text{big}}, F^j_{\text{big}}) | 1 \leqslant j \leqslant J\}$, $\Omega_{\text{little}} = \{(V^k_{\text{little}}, F^k_{\text{little}}) | 1 \leqslant k \leqslant K\}$, $\Omega_{\text{GPU}} = \{(V^h_{\text{GPU}}, F^h_{\text{GPU}}) | 1 \leqslant h \leqslant H\}$表示它们各自可选的电压和频率组合的集合.

2.2 应用模型

假设实时的 OpenCL 应用集合 $\mathcal{A} = \{A_1, A_2, \cdots, A_S, A_{S+1}, A_{S+2}, \cdots, A_Q\}$在给定的硬件平台上运行. 其中, $\mathcal{A}_{\text{offline}} = \{A_1, A_2, \cdots, A_S\}$表示系统静态阶段的初始应用集合, $\mathcal{A}_{\text{online}} = \{A_{S+1}, A_{S+2}, \cdots, A_Q\}$表示系统动态阶段的新到应用集合. 对于一个初始应用 $A_s (1 \leqslant s \leqslant S)$, 它需要执行的指令条数 W_s和截止期限 D_s是已知的. 假设初始应用的执行顺序已经通过现有算法确定, 例如采用文献 [18] 的最短截止期限优先(earliest deadline first, EDF)算法. 但是, 对于一个新到应用 $A_r (S+1 \leqslant r \leqslant Q)$, 它的达到时间 T_r、需要执行的指令总数 W_r、截止期限 D_r在静态阶段是未知的, 只有在系统运行时的动态阶段才能获得.

此外, 无论是初始应用 A_s和新到应用 A_r, 均支持数据并行化操作, 能够根据系统运行状态调节 CPU 和 GPU 的负载、负载到 CPU 大核心和小核心的映射. 为了便于表述, 用符号 $A_q (1 \leqslant q \leqslant Q)$表示集合 $\mathcal{A} = \{A_1, A_2, \cdots, A_Q\}$中的任意一个应用. 用符号 P_q表示应用 A_q的划分点, 取值为区间[0, 1]的离散值. 例如, 应用划分点集合可以设置为{0, 0.2, 0.4, 0.6, 0.8, 1}, 当 P_q=0.4 时, 表示 40% 的指令条数(或工作组)是由 CPU 核心完成的, 而剩余 60% 的指令条数(或工作组)是由 GPU 核心完成的.

进一步地, 用元组 $\mathcal{Z}_q = (P_q, \Pi_q, F^q_{\text{big}}, F^q_{\text{little}}, F^q_{\text{GPU}})$表示执行应用 A_q时的系统资源配置, 其中 Π_q为调用 CPU 的大核心和小核心的编号集合, F^q_{big}为存储 CPU 大核心的工作频率, F^q_{little}为存储 CPU 小核心的工作频率, F^q_{GPU}为记录 GPU 核心 C^0_{GPU}的工作频率. 例如, 当设置 $\Pi_q = \{C^1_{\text{big}}, C^2_{\text{big}}, C^1_{\text{little}}, C^2_{\text{little}}\}$, $F^q_{\text{big}} = 2.3$ GHz, $F^q_{\text{little}} = 1.8$ GHz, $F^q_{\text{GPU}} = 2.0$ GHz时, 表示 CPU 大核心 C^1_{big}和 C^2_{big}以 2.3 GHz 的频率运行, CPU 小核心 C^1_{little}和 C^2_{little}以 1.8 GHz的频率运行, GPU 核心 C^0_{GPU}以 2.0 GHz的频率运行. 为了便于理解, 采用文献 [6−15] 的表述, 将 A_q的划分点、A_q到 CPU 大核心和小核心的映射称为 A_q的设计点.

2.3 处理器寿命模型

文献 [19−20] 指出, 电迁移、经时击穿、热循环是引起多核处理器系统失效的主要机制. 电迁移指的是在电流和温度的共同作用下, 金属线出现断裂从

而导致芯片无法正常工作. 用符号λ_{EM}表示由电迁移引起的久性故障速率, 计算公式[19]为

$$\lambda_{\mathrm{EM}}=\hbar\times(\psi-\psi_0)^{\omega}\times\exp\left\{\frac{-k_1}{k_2\times T_{\mathrm{tem}}}\right\},\tag{1}$$

其中$\hbar$, ω, k_1, k_2均为非负常量, ψ是电流密度, ψ_0是电流密度常量, T_{tem}为核心的温度. 经时击穿指的是在栅极上施加一个电场低于栅氧的本征击穿场强的恒定电压, 经过一段时间的运行后, 氧化膜被击穿, 造成核心的永久性损坏. 用符号λ_{TDDB}表示由经时击穿引起的久性故障速率, 计算公式[20]为

$$\lambda_{\mathrm{TDDB}}=\hbar\times\exp\left\{\alpha\times\beta-\frac{k_1}{\omega\times T_{\mathrm{tem}}}\right\},\tag{2}$$

其中α和β分别是场加速因子和跨电介质的电场.与电迁移和经时击穿不同, 热循环是指由于相邻材料层的热膨胀系数不匹配产生的热应力造成的磨损, 运行时的温度变化导致非弹性变形, 最终导致元器件损坏.由文献[19]可知, 由热循环引起的永久性故障速率为

$$\lambda_{\mathrm{TC}}=\varpi\times\left(T_{\mathrm{tem}}^{\mathrm{all}}-T_{\mathrm{tem}}^{\mathrm{part}}\right)^{-\eta}\times f_{\mathrm{TC}},\tag{3}$$

其中ϖ是常数, $T_{\mathrm{tem}}^{\mathrm{all}}$和$T_{\mathrm{tem}}^{\mathrm{part}}$分别表示整个和部分温度循环范围, η是科芬-曼森指数, f_{TC}是热循环频率. 利用速率累加法则[19-20], 单个核心的使用寿命可以估算为累加的永久性故障速率对时间t的积分, 即

$$R_{\mathrm{core}}=\int_0^{+\infty}(\lambda_{\mathrm{TDDB}}+\lambda_{\mathrm{EM}}+\lambda_{\mathrm{TC}})\,\mathrm{d}t.\tag{4}$$

3 问题定义和方法概述

3.1 问题定义

给定 CPU-GPU MPSoC 和实时应用集合$\mathcal{A}=\{A_1, A_2, \cdots, A_Q\}$, 在时序、能耗、峰值温度、使用寿命的约束下, 为每个应用找到最佳的设计点(包括划分点以及该应用到 CPU 大核心和小核心的映射), 从而最小化应用的平均延迟. 用符号$\mathcal{L}_q$表示应用A_q的延迟, 则$\mathcal{L}_q$可以估算为[14]

$$\mathcal{L}_q=\max\left\{\mathcal{L}_{\mathrm{CPU}}^q\times P_q, (1-P_q)\times\mathcal{L}_{\mathrm{GPU}}^q\right\},\tag{5}$$

其中$\mathcal{L}_{\mathrm{CPU}}^q$表示应用$A_q$的指令全部在 CPU 核心执行时的延迟, $\mathcal{L}_{\mathrm{GPU}}^q$表示应用$A_q$的指令全部在 GPU 核心执行时的延迟. 上述研究问题可以形式化为式(6)~(10):

$$\min\frac{1}{Q}\sum_{q=1}^{Q}\mathcal{L}_q,\tag{6}$$

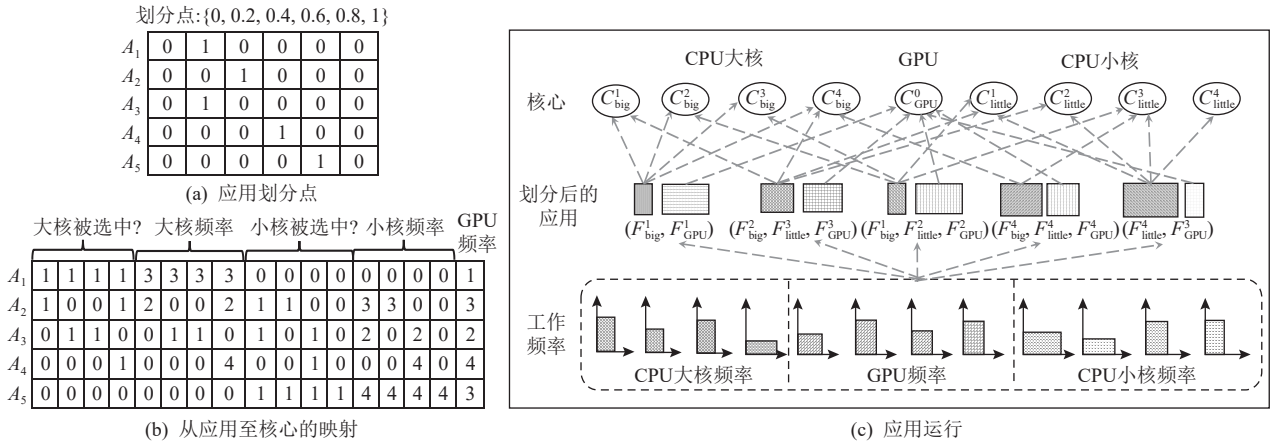
$$\text{s.t.}\sum_{q=1}^{Q}E_q\leqslant E_{\mathrm{bgt}},\tag{7}$$

$$\left\{T_{\mathrm{core}}^{\mathrm{big}}, T_{\mathrm{core}}^{\mathrm{little}}, T_{\mathrm{core}}^{0}\right\}\leqslant T_{\mathrm{peak}},\tag{8}$$

$$\left\{R_{\mathrm{core}}^{\mathrm{big}}, R_{\mathrm{core}}^{\mathrm{little}}, R_{\mathrm{core}}^{0}\right\}\geqslant R_{\mathrm{life}},\tag{9}$$

$$\mathcal{L}_q\leqslant D_q,\ \forall q=1,2,\cdots,Q.\tag{10}$$

式(6)为优化目标, 表示最小化所有应用的平均延迟; 式(7)为能耗约束, E_q表示处理应用A_q的能耗, 应用的总能耗不能超过预先设定的能量预算E_{bgt}; 式(8)为峰值温度约束, $T_{\mathrm{core}}^{\mathrm{big}}$, $T_{\mathrm{core}}^{\mathrm{little}}$, T_{core}^{0}分别表示 CPU 大核集群、CPU 小核集群、GPU 核心C_{GPU}^0的峰值温度, T_{peak}为给定的峰值温度阈值; 式(9)为使用寿命约束, $R_{\mathrm{core}}^{\mathrm{big}}$, $R_{\mathrm{core}}^{\mathrm{little}}$, R_{core}^{0}分别表示 CPU 大核集群、CPU 小核集群、GPU 核心C_{GPU}^0的使用寿命, R_{life}为给定的使用寿命阈值; 式(10)为时序约束, 即任意一个应用的延迟不可以超过给定的截止期限D_q. 为了便于理解, 图 2 展示了本文研究问题示意图.



(a) 应用划分点

(b) 从应用至核心的映射

(c) 应用运行

Fig. 2 Illustration of our studied optimization problem

图 2 本文研究问题示意图

3.2 方法概述

本文针对系统静态阶段的初始应用集合 $\mathcal{A}_{\text{offline}}=\{A_1,A_2,\cdots,A_S\}$和系统动态阶段的新到应用集合 $\mathcal{A}_{\text{online}}=\{A_{S+1},A_{S+2},\cdots,A_Q\}$，分别提出了相应的设计点优化技术. 对于初始应用集合 $\mathcal{A}_{\text{offline}}=\{A_1,A_2,\cdots,A_S\}$，本文基于交叉熵优化策略，首先设计了一种设计点样本表示方法，然后提出利用拉丁超立方采样和样本微调技术来提高设计点寻优效率. 对于新到应用集合 $\mathcal{A}_{\text{online}}=\{A_{S+1},A_{S+2},\cdots,A_Q\}$，本文设计了基于反馈控制的设计点动态优化技术，主要包括比例-积分-微分（proportional-integral-derivative, PID）控制器、应用准入控制器、应用映射控制器、主控制器，能够有效降低制定动态调度决策的时间开销，同时最小化新到应用的平均延迟. 通过上述优化过程，本文提出的方案能够在满足所有系统约束的前提下，最小化 OpenCL 应用的平均延迟.

4 基于交叉熵策略的初始应用设计点寻优技术

本节设计了一种基于交叉熵策略的初始应用设计点寻优技术，首先介绍交叉熵策略的理论基础，然后描述应用特性驱动的交叉熵策略改进机制，最后给出基于改进机制的设计点静态寻优算法.

4.1 交叉熵策略的理论基础

给定任意一个关于多维变量 x 的函数 $\Phi(x)$，目标是在搜索空间 $\mathcal{S}$ 内找到最优映射，使得函数 $\Phi(x)$ 在 $x=x^*$ 处取最小值，即

$$\Phi(x^*)=\gamma^*=\min_{x\in\mathcal{S}}\Phi(x). \tag{11}$$

显然，式(11)所示的优化问题是一个确定性优化问题. 与直接求解该确定性问题的传统优化策略不同，交叉熵策略采用了问题转化的思想，将原问题转化成一个随机优化问题[21]. 本质上，交叉熵策略是一种基于重要性抽样的通用蒙特卡洛方法，用于解决稀有事件的概率估计. 在交叉熵优化策略中，式(11)可转化为

$$\mathcal{T}_u(\Phi(X)\leqslant\gamma)=\Theta_u(I_{\{\Phi(X)\leqslant\gamma\}}). \tag{12}$$

在式(12)中，根据当前的第 u 个概率 $\mathcal{T}(x,u)$，生成了包含 Z 个随机样本的样本集合 $X=\{X_1,\cdots,X_z,\cdots,X_Z\}$，其中每个样本代表原问题的一个解. γ 为阈值，$\mathcal{T}_u(\Phi(X)\leqslant\gamma)$ 是样本函数值 $\Phi(X)$ 不大于阈值 γ 的概率. $I_{\Phi(X)\leqslant\gamma}$ 是指示函数，表达式为

$$I_{\Phi(X)\leqslant\gamma}=\begin{cases}1, & \Phi(X)\leqslant\gamma,\\ 0, & \text{其他}.\end{cases} \tag{13}$$

$\Theta_u(I_{\Phi(X)\leqslant\gamma})$ 为指示函数 $I_{\Phi(X)\leqslant\gamma}$ 的期望，计算式为

$$\Theta_u(I_{\Phi(X)\leqslant\gamma})=\sum_{z=1}^{Z}\mathcal{T}_u(\Phi(X_z)\leqslant\gamma)\times I_{\Phi(X_z)\leqslant\gamma}. \tag{14}$$

交叉熵策略采用了迭代抽样方法，试图逐步改变随机搜索的抽样分布，使稀有事件的概率不断增加. 采样算法的每次迭代会生成代表原始问题解的多个随机样本，这些随机样本以一定概率收敛于最优解. 概括地说，利用交叉熵策略求解优化问题的主要步骤包括：

1）初始化迭代计数器 $g\leftarrow 1$ 和概率向量 $\mathcal{T}_0$；

2）根据概率向量 $\mathcal{T}_{g-1}$ 随机产生 Z 个样本集合 $X=\{X_1,\cdots,X_z,\cdots,X_Z\}$；

3）计算样本的性能，按照性能递减的顺序对样本排序，然后选取性能最好的 B^{elite} 个精英样本；

4）利用式(15)计算第 g 次迭代的阈值

$$\gamma_g\leftarrow\frac{1}{B^{\text{elite}}}\times\sum_{v\in\vartheta}\Phi(X_v)\ , \tag{15}$$

其中 ϑ 为性能最好的 B^{elite} 个精英样本的下标集合；

5）利用式(16)更新第 g 次迭代的概率

$$\mathcal{T}_g^{a,b}=\sum_{z=1}^{Z}(I_{\Phi(X_z)\leqslant\gamma_g}\times I_{x_{z,a}=b})/\sum_{z=1}^{Z}I_{\Phi(X_z)\leqslant\gamma_g}, \tag{16}$$

其中 $x_{z,a}$ 为样本 X_z 的第 $a(1\leqslant a\leqslant U)$ 个元素，$\mathcal{T}_g^{a,b}$ 为在第 g 次迭代过程中元素 $x_{z,a}$ 映射到 $b(1\leqslant b\leqslant U^*)$ 的概率，且 $\mathcal{T}_g^{a,b}$ 为 $\mathcal{T}_g$ 的元素；

6）如果满足终止迭代条件，输出性能最好的样本，退出，否则更新 $g\leftarrow g+1$，返回步骤 2）.

4.2 应用特性驱动的交叉熵策略改进机制

传统的交叉熵策略作为一种启发式的迭代优化策略，在每次迭代中都会产生大量的蒙特卡洛样本，带来巨大的运行时间开销. 本文提出了一种交叉熵策略改进机制来探索静态的设计点寻优方案. 与标准的交叉熵优化过程相比，改进后的交叉熵策略在第 2）步的样本生成阶段进行了优化. 首先设计了一种设计点样本表示方法，然后提出利用拉丁超立方采样和样本微调技术来提高设计点寻优效率.

具体地说，我们采用了 2 个二维数组来表示应用设计点样本，1 个数组用来存储应用的划分点，另一个数组用来存储应用到核心的映射. 图 2(a)展示了应用划分点数组，该数组中的行号指代应用编号，数组中的每列表示给定的划分点是否被选中. 图 2(a)中有 5 个应用 $\{A_1,A_2,A_3,A_4,A_5\}$，应用划分点以步长 0.2 进行设置，因此可供每个应用选择的划分点为 $\{0,0.2,0.4,0.6,0.8,1\}$. 应用 A_1 选择 0.2 作为划分点，则

该应用 20% 的指令条数是在 CPU 上完成的，剩余的 80% 的指令条数是在 GPU 上完成的. 需要特别指出的是，划分点的步长是可以调节的，当步长较小时，表明系统对应用进行了细粒度划分，最终输出解的精度也就越高. 图 2(b)为应用到核心的映射数组示意图，在图 2(b)中，假设有 4 个同构的 CPU 大核、4 个同构的 CPU 小核、1 个 GPU 核心，且 CPU 和 GPU 都支持 4 个离散的频率. 应用到核心映射点数组的第 1~4 列指示了某个 CPU 大核是否被选中执行特定的应用，如果被选中，则将对应的单元置为 1；第 5~8 列指示 CPU 大核的频率等级. 类似地，第 9~12 列指示了某个 CPU 小核是否被选中执行特定的应用，第 13~16 列指示 CPU 小核的频率等级，第 17 列指示 GPU 核心的频率等级. 图 2(c)展示了根据生成的划分点和应用到核心的映射，5 个应用在核心上运行的示意图.

接下来，利用文献 [22] 提出的拉丁超立方采样技术来生成样本点. 拉丁超立方采样是用于多元参数分布的近似随机分层抽样技术. 对于一维变量抽样，拉丁超立方采样技术首先使用概率分布来估计变量的不确定性，然后将变量的范围划分为概率相等的区间，并在每个区间产生变量的样本值. 对于二维变量抽样，拉丁超立方采样技术首先将样本空间划分为若干个具有相同采样概率的二维网格，然后随机选择 1 个网格来生成样本，同时与该网格相同行或相同列的网格不能作为未来样本生成的候选网格. 以对二维变量(x_1, x_2)进行抽样为例，图 3 展示了采用随机采样技术和拉丁超立方采样技术的采样结果. 从图 3 可以看出，与随机采样技术生成的样本点相比，拉丁超立方采样技术可以用更少的样本数来较好地覆盖采样空间，极大地缩短了采样的时间开销. 类似地，对于多维变量抽样，拉丁超立方采样技术需要将每一维分成互不重叠的若干区间，并确保每个区间有相同的采样概率，紧接着在区间内随机抽取 1 个点，并将它们组成 1 个样本. 考虑到拉丁超立方采样技术的优势，在设计点采样阶段，本文利用该技术来生成样本点.

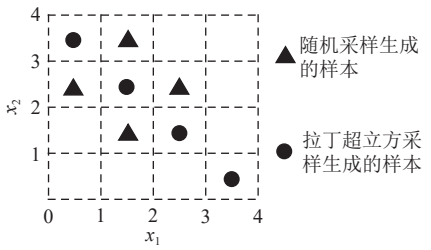


Fig. 3 Comparison of random sampling and Latin hypercube sampling

图 3 随机采样和拉丁超立方采样对比

在产生样本集合$X = \{X_1, \cdots, X_z, \cdots, X_Z\}$后，进一步对每个样本$X_z(1 \leqslant z \leqslant Z)$进行微调操作以提高单个样本的性能. 具体地说，样本微调测策略是受到文献 [4] 的实验观察启发：在执行特定的应用程序时，存在一个使该应用获得最优性能的 CPU 大核和 CPU 小核的最佳组合，当更多的 CPU 核心参与到应用程序的执行时，应用的性能不会进一步提升，反而会产生额外的能量和延迟开销. 根据这一观察，可以对样本进行微调操作，微调后的样本更接近最优解，进而显著地提高了交叉熵算法的迭代效率. 以图 2(b)展示的从应用到核心的映射为例，应用A_2在 CPU 大核C^1_{big}和C^4_{big}的第 2 个频率等级、小核C^1_{little}和C^2_{little}的第 3 个频率等级上运行. 首先，如图 4(a)所示，尝试增加执行应用A_2的 CPU 大核数目，选择交换应用A_2和应用A_1的 CPU 大核映射点，此时 4 个 CPU 大核全部参与应用A_2的执行，所有的 CPU 小核都不参与应用A_2的执行. 接下来，尝试减少执行应用A_2的 CPU 大核和 CPU 小核的数目. 如图 4(b)所示，可以将应用A_2和应用A_4的 CPU 大核和 CPU 小核映射点进行交换，此时应用A_2在 CPU 大核C^1_{big}的第 4 个工作频率、小核C^3_{little}的第 4 个工作频率上执行. 对于图 4(a)和图 4(b)生成的 2 个微调样本，分别计算出它们对应的性能，然后比较它们的性能是否优于微调前的样本性能. 如果图 4(a)中的微调样本性能优于微调前的样本性能，则在下次样本微调操作时，应尝试继续增加执行应

	大核被选中?				大核频率				小核被选中?				小核频率				GPU 频率
A_1	1	0	0	1	2	0	0	2	0	0	0	0	0	0	0	0	1
A_2	1	1	1	1	3	3	3	3	1	1	0	0	1	3	0	0	3
A_3	0	1	1	0	0	1	3	0	1	0	1	0	2	0	4	0	2
A_4	0	0	0	1	0	0	0	4	0	0	1	0	0	0	4	0	4
A_5	0	0	0	0	0	0	0	0	1	1	1	1	4	4	4	4	3

(a) 交换A_2和A_1的CPU大核映射点

	大核被选中?				大核频率				小核被选中?				小核频率				GPU 频率
A_1	1	1	1	1	3	3	3	3	0	0	0	0	0	0	0	0	1
A_2	0	0	0	1	0	0	0	4	0	0	1	0	0	0	4	0	3
A_3	0	1	1	0	0	1	3	0	1	0	1	0	2	0	4	0	2
A_4	1	0	0	1	2	0	0	2	1	1	0	0	3	3	0	0	4
A_5	0	0	0	0	0	0	0	0	1	1	1	1	4	4	4	4	3

(b) 交换A_2和A_4的CPU大核和CPU小核映射点

Fig. 4 Example of fine tuning the samples of design points

图 4 设计点样本微调示例

用A_2的 CPU 资源，比如提高 CPU 大核的工作频率、增加执行应用A_2的 CPU 小核数目等；否则，在下次样本微调操作时，应尝试减少执行应用A_2的 CPU 资源，比如降低 CPU 大核和 CPU 小核的工作频率.

4.3 基于改进机制的初始应用设计点寻优算法

根据应用特性驱动的交叉熵策略改进机制，本文提出了一种应用设计点静态寻优技术，如算法 1 所示.

算法 1. 基于改进交叉熵策略的静态寻优算法.

输入：$\mathcal{A}_{\text{offline}}$, $\{C_{\text{big}}, C_{\text{little}}, C^0_{\text{GPU}}\}$;

输出：性能最优的单个样本.

① 初始化概率向量$\boldsymbol{\mathcal{T}}_0$, 设置计数器$g \leftarrow 1$;

② while $g \leqslant g^{\max}$ do

③ $X \leftarrow ProduceSample(\boldsymbol{\mathcal{T}}_{g-1}, Z)$;

④ $\Gamma \leftarrow SelectApp(\mathcal{A}_{\text{offline}}, Y)$;

⑤ for $A_y \in \Gamma$ do

⑥ $X_{\text{tune}} \leftarrow TuneSample(X, A_y)$;

⑦ for $X_u \in X_{\text{tune}}$ do

⑧ if $\Phi(X_u) > \Phi(X^0_u)$ then

⑨ $X^0_u \leftarrow X_u$, $\Delta^u_{g+1} \leftarrow \Delta^u_g$;

⑩ else

⑪ $\Delta^u_{g+1} \leftarrow -\Delta^u_g$;

⑫ end if

⑬ end for

⑭ end for

⑮ 更新阈值γ_g和概率向量$\boldsymbol{\mathcal{T}}_g$;

⑯ $g \leftarrow g+1$;

⑰ end while

⑱ return 性能最优的单个样本.

算法 1 的输入为应用集合$\mathcal{A}_{\text{offline}} = \{A_1, A_2, \cdots, A_S\}$和核心集合$\{C_{\text{big}}, C_{\text{little}}, C^0_{\text{GPU}}\}$. 算法 1 首先对概率向量$\boldsymbol{\mathcal{T}}_0$和迭代计数器$g$进行初始化操作，然后以迭代的方式寻找最优的静态设计点. 在迭代过程中，根据概率向量$\boldsymbol{\mathcal{T}}_{g-1}$, 利用样本生成函数$ProduceSample(\boldsymbol{\mathcal{T}}_{g-1}, Z)$产生总共$Z$个拉丁超立方采样样本. 紧接着，调用应用选择函数$SelectApp(\mathcal{A}_{\text{offline}}, Y)$, 从应用集合$\mathcal{A}_{\text{offline}} = \{A_1, A_2, \cdots, A_S\}$中随机选取$Y$个应用进行样本微调操作. 对于任意一个被选中的应用A_y($1 \leqslant y \leqslant Y$), 使用样本微调函数$TuneSample(X, A_y)$对其样本进行微调操作. 同时，引入标志位$\Delta^u_g$, 如果$\Delta^u_g$=1, 表示更多的 CPU 资源会带来应用$A_y$的性能提升，即需要增加分配给应用$A_y$的 CPU 资源；反之，如果$\Delta^u_g$=−1, 表示更多的 CPU 资源反而会降低应用$A_y$的性能，即需要减少分配给应用$A_y$的 CPU 资源. 因此，可以比较微调后样本$X_u$的性能是否强于微调前样本$X^0_u$的性能. 如果是，用微调后的样本$X_u$替代微调前的样本$X^0_u$, 并将当前的标志位$\Delta^u_g$赋值给下次迭代的标志位$\Delta^u_{g+1}$；如果微调后的样本性能均弱于微调前的样本性能，则表明当前的微调操作不能提升样本的性能，需要将当前标志位Δ^u_g的相反数赋值给下次迭代的标志位Δ^u_{g+1}. 经过总共$g^{\max}$次迭代，算法 1 最终输出性能最优的单个样本.

5 基于反馈控制的新到应用设计点优化技术

本节介绍基于反馈控制的新到应用设计点优化技术，首先概述该技术的基本思想，然后详述 PID 控制器、应用准入控制器、应用映射控制器、主控制器工作过程.

5.1 反馈控制策略的基本思想

图 5 展示了本文提出的基于反馈控制的动态技术示意图. 如图 5 所示，该动态技术包含了准入队列、等待队列、PID 控制器、应用准入控制器、应用映射控制器、EDF 调度器、主控制器. 对于系统静态阶段的初始应用集合$\mathcal{A}_{\text{offline}} = \{A_1, A_2, \cdots, A_S\}$, 可以直接调用静态方案为它们选取最优的设计点. 对于系统运行时的新到应用集合$\mathcal{A}_{\text{online}} = \{A_{S+1}, A_{S+2}, \cdots, A_Q\}$, 设计了 2 个队列：准入队列用来存储被系统允许进入的应用，等待队列用来存储未被系统允许进入的应用. PID 控制器周期性地对当前系统中处理器核心的平均利用率θ进行采样，并将控制动作φ返回给主控制器. 主控制器调用应用映射控制器和应用准入控制器，以使当前系统中处理器核心的平均利用率θ与利用率控制变量φ相匹配. 应用映射控制器负责将空闲的处理器核心分配给新到应用.EDF 调度器根据 EDF 调度策略，负责管理准入队列里的应用在处理器核心上的执行.

5.2 PID 控制器

PID 控制器的运行过程如算法 2 所示. 算法 2 的输入是预先设定的约束条件违背等级的阈值ε.首先判断当前的约束条件违背等级ε_{now}是否大于阈值ε, 如果是，则需要迭代地优化系统的资源利用率. 根据文献 [23], 可利用式(17)更新系统资源利用率控制变量.

$$\varphi = -\ell_1 \times \xi(t) - \ell_2 \times \sum_{t=1}^{\phi_1} \xi(t) - \ell_3 \times \frac{\xi(t) - \xi(t-\phi_2)}{\phi_2}, \tag{17}$$

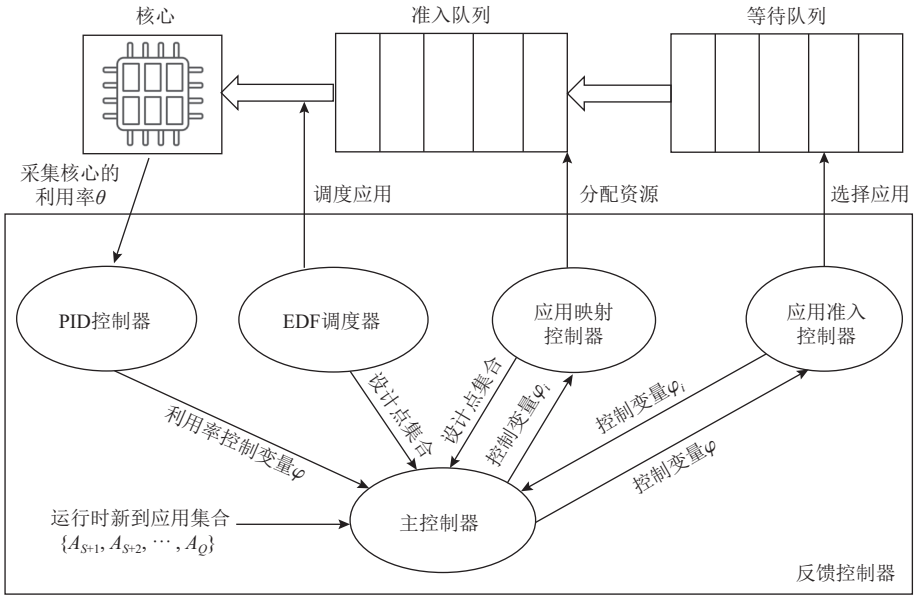


Fig. 5 Overview of our feedback control based dynamic scheduling technique

图 5 基于反馈控制的动态调度技术概览

其中ℓ_1, ℓ_2, ℓ_3分别代表 PID 控制器的比例、积分、微分系数, $\xi(t)$表示约束条件违背等级ε_{now}和阈值ε的差值(即$\xi(t)=\varepsilon-\varepsilon_{\text{now}}$), ϕ_1表示在系统运行时发生积分误差的调度窗口个数, ϕ_2表示系统运行时发生微分误差的调度窗口个数. 算法 2 使用 PID 控制器对应用执行状态采样以更新约束条件违背等级ε_{now}.当满足终止迭代条件时, 输出系统资源利用率控制变量φ. 容易看出, 当$\varphi>0$时, 可以允许更多的新应用进入系统以提高系统的资源利用率; 相反, 当$\varphi<0$时, 需要禁止新应用进入系统执行从而降低系统的资源利用率.

算法 2. PID 控制算法.

输入: 约束条件违背阈值ε;

输出: 系统资源利用率控制变量φ.

① while $\varepsilon_{\text{now}}>\varepsilon$ do

② 利用式(17)计算φ;

③ PID 控制器采样以更新ε_{now};

④ end while

⑤ return 系统资源利用率控制变量φ.

5.3 应用准入控制器

应用准入控制器的运行过程如算法 3 所示. 该算法的输入包括系统资源利用率控制变量φ、准入队列里的应用个数Q_{accept}、等待队列里的应用个数Q_{wait}. 如果系统资源利用率控制变量φ>0, 此时可以允许更多的新应用进入系统执行以提高系统的资源利用率, 因此需要根据 EDF 算法对等待队列里的应用排序. 对于等待队列里的队头应用A_i, 分配给该应用可用于提升系统资源利用率的阈值φ_i. 根据应用的指令条数来确定φ_i, 即

$$\varphi_i=\frac{W_i}{W_1+W_2+\cdots+W_{Q_{\text{wait}}}}. \tag{18}$$

换言之, 一个应用的指令条数越多, 则分配给它的可用于提升系统资源利用率的阈值越大. 如果$\varphi-\varphi_i>0$成立利用$\theta\leftarrow\theta+\varphi_i$以更新系统当前的资源利用率, 利用$\varphi\leftarrow\varphi-\varphi_i$更新系统资源利用率控制变量. 随后, 从等待队列中删除队头应用, 更新准入队列里应用的个数. 当系统资源利用率控制变量φ<0 时, 输出$\{\varphi_i|1\leqslant i\leqslant Q_{\text{accept}}\}$后退出.

算法 3. 应用准入控制算法.

输入: φ, Q_{accept}, Q_{wait};

输出: $\{\varphi_i|1\leqslant i\leqslant Q_{\text{accept}}\}$.

① if $\varphi>0$ then

② 根据 EDF 算法对等待队列里的应用排序;

③ for $i=1$; $i\leqslant Q_{\text{wait}}$; i++ do

④ 根据式(18)分配φ_i;

⑤ if $\varphi-\varphi_i>0$ then

⑥ $\theta\leftarrow\theta+\varphi_i$;

⑦ $\varphi\leftarrow\varphi-\varphi_i$;

⑧ 从等待队列中删除队头应用;

⑨ $Q_{\text{accept}}\leftarrow Q_{\text{accept}}+1$;

⑩ end if

⑪ end for

⑫ end if

⑬ return $\{\varphi_i|1\leqslant i\leqslant Q_{\text{accept}}\}$.

5.4 应用映射控制器

应用映射控制器的运行过程如算法 4 所示.

算法 4. 应用映射算法.

输入: $\{\varphi_i|1\leqslant i\leqslant Q_{\text{accept}}\}$;

输出: 应用设计点集合.

① for $i=1$; $i\leqslant Q_{\text{accept}}$; i++ do

② $\sigma\leftarrow CoreIdleCheck$;

③ if $\sigma\neq 0$ then

⑥ while $\varphi_i>0$ do

⑦ 分配一个空闲核心给 A_i;

⑧ 计算资源利用率增量 ϱ_i;

⑨ $\varphi_i\leftarrow\varphi_i-\varrho_i$;

⑩ end while

⑪ end if

⑫ end for

⑬ return 应用设计点集合.

算法 4 的输入为用于提升系统资源利用率的阈值集合$\{\varphi_i|1\leqslant i\leqslant Q_{\text{accept}}\}$. 对于准入队列里的任意一个应用, 调用函数*CoreIdleCheck*来判断系统中是否有处于空闲状态的核心, 如果系统存在处于空闲的核心, 函数*CoreIdleCheck*返回值为 1, 否则返回值为 0. 如果函数*CoreIdleCheck*返回值为 1, 则进入空闲核心分配过程. 此时, 需要进一步判断φ_i是否大于 0, 如果是, 随机分配一个空闲核心给A_i, 并计算资源利用率增量ϱ_i, 更新可用于提升系统资源利用率的阈值φ_i. 注意到在该过程中, 分配给应用A_i的空闲核心的工作频率是由同一个集群中的处于运行状态的核心确定的. 应用A_i负载的划分采用平均分配原则, 即分配给应用A_i的空闲核心需要执行的指令条数是相等的. 当可用于提升系统资源利用率的阈值φ_i耗尽时, 进行下一次迭代. 整个算法在输出准入队列里的应用设计点集合后退出.

5.5 主控制器

主控制算法整合了 PID 控制器、应用准入控制器、应用映射控制器, 形成一个闭环, 有效提高了控制过程对外部干扰的鲁棒性. 主控制器的运行过程如算法 5 所示.

算法 5. 主控制算法.

输入: $\mathcal{A}_{\text{online}}=\{A_{S+1},A_{S+2},\cdots,A_Q\}$;

输出: 应用调度表.

①调用算法 2 获取资源利用率控制变量φ;

②调用算法 3 获取$\{\varphi_i|1\leqslant i\leqslant Q_{\text{accept}}\}$;

③调用算法 4 生成应用设计点;

④调用 EDF 调度器执行应用;

⑤ return 应用调度表.

算法 5 的输入为在系统动态运行时, 新到来的应用集合$\mathcal{A}_{\text{online}}=\{A_{S+1},A_{S+2},\cdots,A_Q\}$. 算法 5 调用算法 2 以获取系统资源利用率控制变量φ, 调用算法 3 获取可用于提升系统资源利用率的阈值集合$\{\varphi_i|1\leqslant i\leqslant Q_{\text{accept}}\}$. 接下来, 算法 5 调用算法 4 生成应用调度表并采用 EDF 调度器执行应用.

6 实验验证

6.1 硬件平台

本文采用 2 种 CPU-GPU MPSoC 硬件平台. 一种是 Hardkernel Odroid-XU3 硬件平台[2], 集成了三星 Exynos 5422 MPSoC, 包含 4 个 ARM Cortex A15 核心、4 个 ARM Cortex A7 核心、1 个 ARM Mali-T628 MP6 GPU. 4 个 ARM Cortex A15 核心构成了高性能的 CPU 大核集群, 每个核心都支持步长为 100 MHz、200~2 000 MHz 之间的多种离散频率. 4 个 ARM Cortex A7 核心组成了低功耗的 CPU 小核心集群, 每个核心都支持步长为 100 MHz、200~1 400 MHz 之间的不同离散频率. 对于 ARM Mali-T628 MP6 GPU, 它的工作频率从 600 MHz, 543 MHz, 480 MHz, 420 MHz, 350 MHz, 266 MHz, 177 MHz 中选取.

除了 Hardkernel Odroid-XU3 平台, 本文还将三星 Exynos 9810 MPSoC[24-25] 作为测试硬件平台. Exynos 9810 MPSoC 的 CPU 大核集群包含 4 个 M3 核心, 每个核心支持 18 种离散的工作频率, 包括 2652 MHz, 2496 MHz, 2314 MHz, 2106 MHz, 2002 MHz, 1924 MHz, 1794 MHz, 1690 MHz, 1586 MHz, 1469 MHz, 1261 MHz, 1170 MHz, 1066 MHz, 962 MHz, 858 MHz, 741 MHz, 704 MHz, 650 MHz; CPU 小核集群包含 4 个 ARM Cortex A55 核心, 每个核心支持 10 种不同的离散频率, 包括 1690 MHz, 1456 MHz, 1248 MHz, 1053 MHz, 949 MHz, 832 MHz, 794 MHz, 715 MHz, 598 MHz, 455 MHz; GPU 集群由 ARM Mali-G72 MP18 GPU 构成, 支持 6 种离散的工作频率, 包括 572 MHz, 546 MHz, 455 MHz, 338 MHz, 299 MHz, 260 MHz.

对于 Hardkernel Odroid-XU3 和 Exynos 9810 MPSoC, 在系统静态阶段, 我们采用热量仿真工具 HotSpot[26] 获取 GPU 和任意 CPU 核心的温度. 在系统动态阶段, 为了降低获取温度数值的时间开销, 利用片上系统的温度传感器来捕捉核心温度. Hardkernel Odroid-XU3 部署了 5 个温度传感器, 其中 4 个温度传感器分别用来监测 4 个 ARM Cortex A15 核心的温度,

1个温度传感器用来监测 ARM Mali-T628 MP6 GPU 的温度. Exynos 9810 MPSoC 同样部署了 5 个温度传感器, 但只有 1 个温度传感器用来测量 CPU 大核集群的温度, 无法获取 CPU 小核集群和 GPU 的温度. 考虑到无论是 CPU 小核集群还是 GPU 的工作频率, 它们支持的平均工作频率均低于 CPU 大核集群支持的平均工作频率, 导致 CPU 小核集群和 GPU 的温度通常低于 CPU 大核集群的温度. 因此, 我们采用与文献 [24] 和文献 [27] 类似的做法, 在系统动态阶段, 忽略对 Hardkernel Odroid-XU3 的 CPU 小核集群、Exynos 9810 MPSoC 的小核集群和 GPU 的温度管理. 相应地, 在系统动态阶段, 只对 Hardkernel Odroid-XU3 的 CPU 大核集群和 GPU、Exynos 9810 MPSoC 的大核集群进行使用寿命优化.

6.2 基准应用

如表 2 所示, 我们从 Hetero-Mark OpenCL[28], PolyBench[29], PARSEC[30] 三个基准应用库选取了 16 个典型的 OpenCL 应用作为测试应用, 包括 K 均值聚类算法(K-means, KM)、页面排序(page rank, PR)、高级加密标准(advanced encryption standard, AES)、背景提取(background extraction, BE)、颜色直方图制作(color histogramming, CH)、布莱克-舒尔斯模型(Black-Scholes model, BS)、力导的边绑定方法(force directed edge bundling, FDEB)、有限冲激响应滤波器(finite impulse response, FIR)、K 近邻算法(K-nearest neighbors, KNN)、演化规划(evolutionary programming, EP)、二叉查找树插入(binary search tree insertion, BSTI)、基因比对(gene alignment, GA)、二维卷积(convolution-2D, C2D)、对称 rank-2k 更新(symmetric rank-2k update, SYR2K)、人体追踪(body track, BT)、内容相似性搜索(content similarity search, CSS).单个基准应用的指令条数和工作组个数如表 2 所示, 它的截止期限设置为

$$D_q=\frac{W_q\times 2.5}{\max\{F^J_{\text{big}},F^K_{\text{little}},F^H_{\text{GPU}}\}\times(J+K)}. \tag{19}$$

6.3 基准算法

为了验证本文算法的性能, 我们将基于交叉熵策略的静态算法与延迟感知的集成式热量管理算法[13] (latency-aware integrated thermal management, LITM)、延迟感知的粒子群优化算法[31](latency-aware particle swarm optimization, LPSO)进行对比. 此外, 还将基于反馈控制的动态算法与对数曲线拟合算法[14](logarithmic curve fitting, LCF)和高能效映射重新划分算法[4](energy efficient mapping and repartitioning, EEMR)进行对比.

1)LITM. 一个用于在异构多核平台上对实时 OpenCL 应用进行热感知调度的框架, 该框架将任务迁移、频率调整、空闲槽插入等多个控制动作视为任务映射参数, 通过自适应的方式调整任务映射参数来降低违背温度约束的概率.

2)LPSO. 一种旨在降低应用执行时间的静态方法, 在应用实时性、系统能耗、处理器温度、系统使用寿命的约束下, 采用粒子群优化算法完成应用指令条数的划分、CPU 核心和 GPU 核心的频率选择.

3)LCF. 一种针对运行在 CPU-GPU 集成平台上的应用划分策略, 由探索阶段、稳定阶段、最终决策阶段构成, 可以动态地将应用负载分配给 CPU 和 GPU, 以同时降低应用的延迟和能耗. 但是, 它忽视了应用的时序要求和系统的使用寿命需求.

4)EEMR. 一种节能的运行时应用映射和划分方法, 根据应用的实时性要求, 对于每个并发执行的应用程序, 映射过程会找到适当数量的 CPU 核心以及 CPU 核心和 GPU 核心的运行频率, 应用划分过程考虑到了 CPU 核心和 GPU 核心之间的负载平衡.然而, 它忽视了系统的使用寿命和峰值温度约束.

Table 2 Number of Instruction Cycles and Work-Groups of Benchmarking Applications[28-30]

表 2 基准应用的指令条数和工作组个数[28-30]

应用	指令条数	工作组个数	应用	指令条数	工作组个数
FDEB	12 582 912	128	KM	97 458 400	128
FIR	22 949 120	256	CSS	856 987 426	256
KNN	26 843 545	128	BT	774 570 000	512
EP	15 680 000	128	PR	984 107 000	128
BSTI	323 000 000	1 024	AES	5 749 000	256
GA	654 280 000	512	BE	1 728 000 000	1 024
C2D	44 568 000	2 048	CH	1 568 000 000	4 096
SYR2K	784 554 712	512	BS	180 018 000	512

6.4 实验分析

在本文提出的静态算法中，设置迭代次数 g^{max}=50，每次迭代生成的样本数目 Z=50，用于微调的样本数目 Y=30. 在本文提出的动态算法中，PID 控制器的参数 ℓ_1, ℓ_2, ℓ_3 分别设置为 0.5, 0.05, 0.1.

6.4.1 静态算法性能分析

图 6 比较了 FDEB, FIR, KNN, EP, BSTI, GA, C2D, SYR2K, KM, CSS 共 10 个基准应用在 Hardkernel Odroid-XU3 硬件平台上执行的延迟. 从图 6 可以看出，本文提出的静态算法可以有效地降低基准应用的延迟. 例如，在执行基准应用 EP 时，本文提出的静态算法、基准算法 LITM、基准算法 LPSO 取得的应用延迟分别为 89.41 s, 130.30 s, 109.71 s. 此外，与基准算法 LITM 和 LPSO 相比，本文提出的静态算法能够将 10 个基准应用的平均延迟分别降低 29.83% 和 23.95%. 图 7 比较了 10 个基准应用在 Exynos 9810 MPSoC 硬件平台上执行的延迟. 与基准算法 LITM 和 LPSO 相比，本文提出的静态算法将基准应用的平均延迟分别降低了 34.58% 和 25.42%.

图 8 和图 9 分别比较了 10 个基准应用在 Hardkernel Odroid-XU3 硬件平台、Exynos 9810 MPSoC 执行的应用能耗. 在图 8 中，设置应用的能量预算 E_{bgt}=3 000 J. 图 8 的数据表明本文提出的静态算法和 2 种基准算法都可满足应用总能耗的约束. 此外，本文提出的静态算法的应用能耗大于 2 种基准算法的应用能耗. 类似地，从图 9 可知，对于任意应用，本文提出的静态算法的应用能耗大于基准算法的应用能耗. 这主要是因为优化能耗和优化延迟是互斥目标，本文提出的静态算法充分利用了给定的能耗预算，以最小化基准应用的延迟.

图 10 列出了本文提出的静态算法、基准算法 LITM、基准算法 LPSO 取得的处理器核心峰值温度. 在本组实验中，将 Hardkernel Odroid-XU3 和 Exynos 9810 MPSoC 的峰值温度阈值分别设置为 70℃ 和 90℃. 从图 10 可以发现，无论是 Hardkernel Odroid-XU3 还是 Exynos 9810 MPSoC 硬件平台，本文提出的

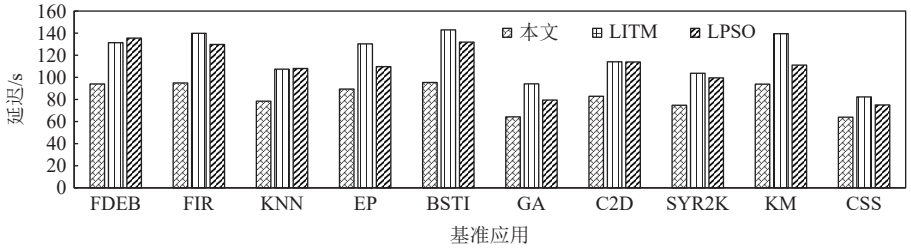


Fig. 6 Latency achieved by static algorithms when running on the Hardkernel Odroid-XU3 platform

图 6 静态算法在 Hardkernel Odroid-XU3 平台上执行实现的延迟

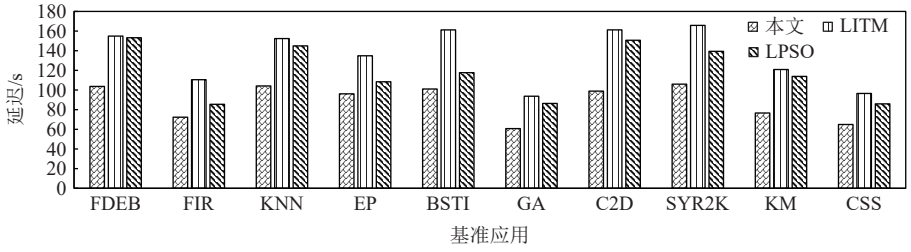


Fig. 7 Latency achieved by static algorithms when running on the Exynos 9810 MPSoC platform

图 7 静态算法在 Exynos 9810 MPSoC 平台上执行实现的延迟

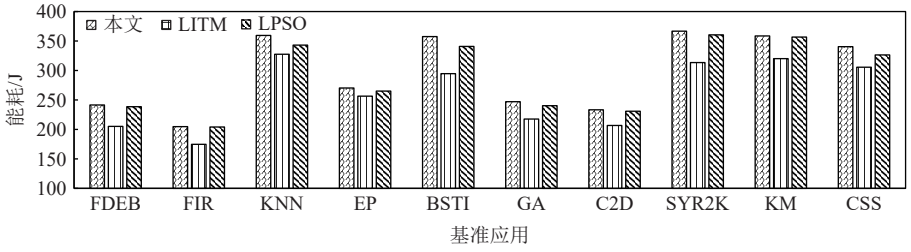


Fig. 8 Energy consumption achieved by static algorithms when running on the Hardkernel Odroid-XU3 platform

图 8 静态算法在 Hardkernel Odroid-XU3 平台上执行实现的能耗

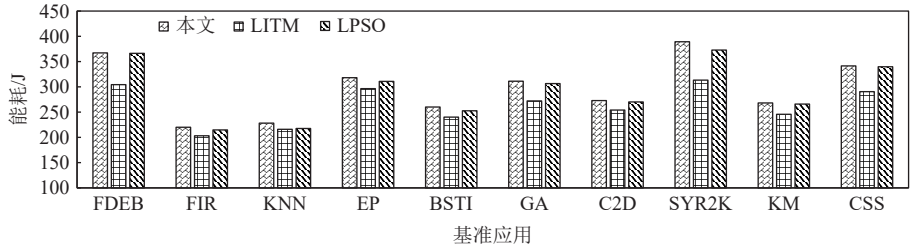


Fig. 9 Energy consumption achieved by static algorithms when running on the Exynos 9810 MPSoC platform

图 9 静态算法在 Exynos 9810 MPSoC 平台上执行实现的能耗

静态算法和 2 种基准算法都可以满足峰值温度的约束. 图 11 给出了本文提出的静态算法、基准算法 LITM、基准算法 LPSO 取得的系统使用寿命. 在这组比较实验中, 将 Hardkernel Odroid-XU3 和 Exynos 9810 MPSoC 使用寿命需求分别设置为 16 年和 18 年. 如图 11 所示, 无论是 Hardkernel Odroid-XU3 还是 Exynos 9810 MPSoC 硬件平台, 本文提出的静态算法和 2 种基准算法都可以满足使用寿命的约束, 并且本文提出的静态算法充分利用了给定的使用寿命阈值, 以优化应用的延迟.

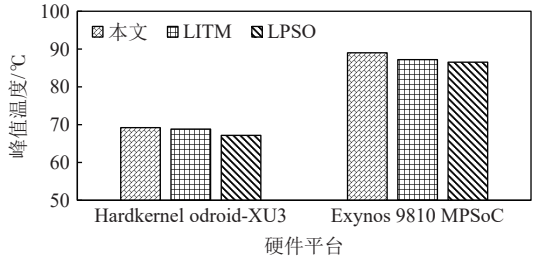


Fig. 10 Peak temperature of processor cores achieved by static algorithms

图 10 静态算法实现的处理器核心峰值温度

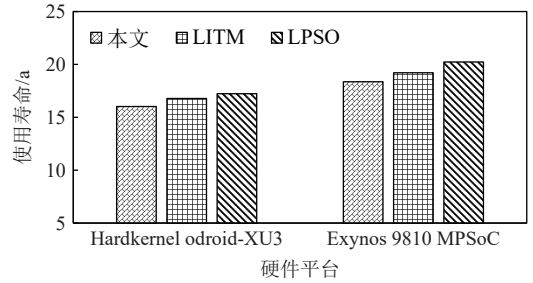


Fig. 11 Lifetime achieved by static algorithms

图 11 静态算法实现的使用寿命

6.4.2 动态算法性能分析

图 12 比较了 3 种动态算法在 Hardkernel Odroid-XU3 硬件平台上执行 6 个基准应用 BT, PR, AES, BE, CH, BS 时的应用延迟. 由图 12 可知, 与基准算法 LCF 和 EEMR 相比, 本文提出的动态算法可以将基准应用的平均延迟分别降低 23.47% 和 24.89%. 进一步地, 图 13 探究了 3 种动态算法在 Exynos 9810 MPSoC 硬件平台上执行 6 个基准应用时的应用延迟. 与图 12 展示的结果类似, 本文提出的动态算法在 Exynos 9810 MPSoC 硬件平台上实现的性能仍然优于基准算法 LCF 和 EEMR.

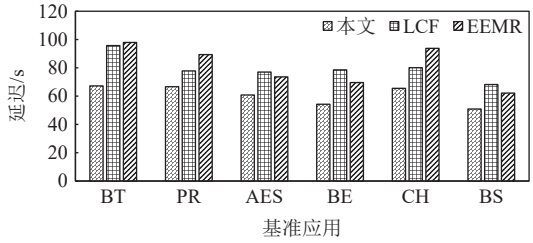


Fig. 12 Latency achieved by dynamic algorithms when running on the Hardkernel Odroid-XU3 platform

图 12 动态算法在 Hardkernel Odroid-XU3 平台上执行实现的延迟

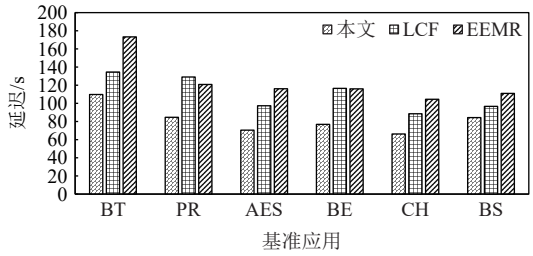


Fig. 13 Latency achieved by dynamic algorithms when running on the Exynos 9810 MPSoC platform

图 13 动态算法在 Exynos 9810 MPSoC 平台上执行实现的延迟

图 14 展示了 3 种动态算法在 Hardkernel Odroid-XU3 硬件平台上执行基准应用 BT, PR, AES, BE, CH, BS 时的能耗. 在这组实验中, 设置应用的能量预算 E_{bgt}=3 000 J. 从图 14 可以看出, 与基准算法 LCF 和 EEMR 相比, 本文提出的动态算法在执行单个应用时消耗了更多的能量, 但是消耗的总能量没有超过给定的能量预算. 进一步地, 图 15 展示了 3 种动态算法在 Exynos 9810 MPSoC 硬件平台上执行基准应用时的能耗. 在这组实验中, 设置应用的能量预算 E_{bgt}=

2 000 J. 由图 15 可知，本文提出的动态算法消耗的总能量依然没有超过给定的能量预算.

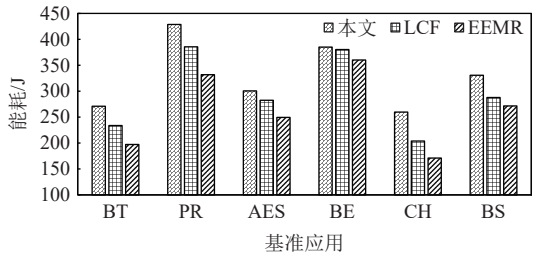


Fig. 14 Energy consumption achieved by dynamic algorithms when running on the Hardkernel Odroid-XU3 platform

图 14 动态算法在 Hardkernel Odroid-XU3 平台上执行实现的能耗

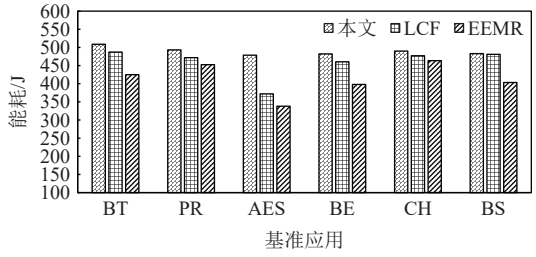


Fig. 15 Energy consumption achieved by dynamic algorithms when running on the Exynos 9810 MPSoC platform

图 15 动态算法在 Exynos 9810 MPSoC 平台上执行实现的能耗

图 16 探究了本文提出的动态算法、基准算法 LCF、基准算法 EEMR 取得的处理器核心峰值温度. 在本组实验中，Hardkernel Odroid-XU3 和 Exynos 9810 MPSoC 的峰值温度阈值仍然分别为 70℃ 和 90℃. 由图 16 可知，本文提出的动态算法执行 6 个基准应用时，均未超过硬件平台设定的峰值温度阈值. 相反，基准算法 LCF 和 EEMR 都超过了硬件平台设定的峰值温度阈值. 图 17 比较了本文提出的动态算法和基准算法 LCF、基准算法 EEMR 取得的系统使用寿命.

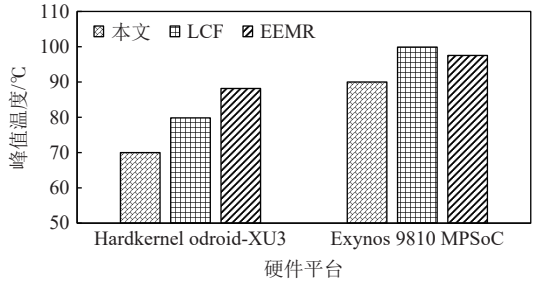


Fig. 16 Peak temperature of processor cores achieved by dynamic algorithms

图 16 动态算法实现的处理器核心峰值温度

在这组对比实验中，Hardkernel Odroid-XU3 和 Exynos 9810 MPSoC 的使用寿命需求仍然分别设置为 16 年和 18 年. 由图 17 可知，本文提出的动态算法在执行 6 个基准应用时，始终没有违背系统使用寿命约束，而基准算法 LCF 和 EEMR 均不能满足使用寿命需求.

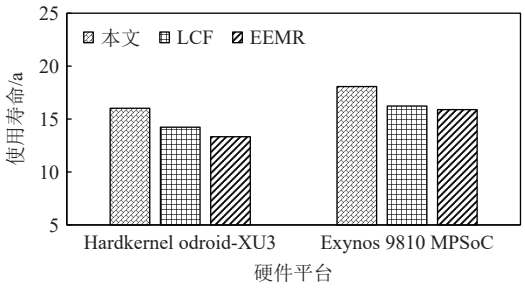


Fig. 17 Lifetime achieved by dynamic algorithms

图 17 动态算法实现的使用寿命

图 18 比较了本文提出的动态算法和基准算法 LCF、基准算法 EEMR 生成任务调度表的运行时间. 从图 18 看出，本文提出的动态算法可以降低生成任务调度表的运行时间. 例如，当运行在 Hardkernel Odroid-XU3 平台时，本文提出的动态算法与基准算法 LCF 和 EEMR 相比，运行时间分别降低了 23.47% 和 30.92%；当运行在 Exynos 9810 MPSoC 时，本文提出的动态算法与 LCF 和 EEMR 相比，运行时间分别降低了 24.71% 和 32.63%.

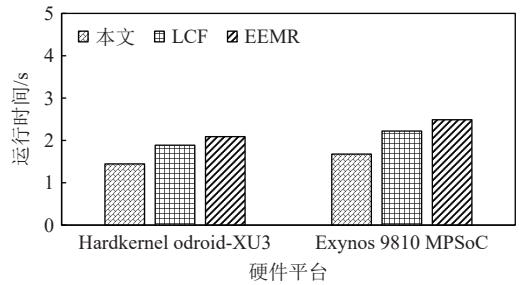


Fig. 18 Runtime overheads of dynamic algorithms

图 18 动态算法的运行时间开销

7 结　论

本文提出了一种使用寿命驱动的调度方法，以最小化 OpenCL 应用在 CPU-GPU MPSoC 执行的延迟. 该方法包括静态调度技术和动态调度技术，静态调度技术利用了交叉熵策略，动态调度技术利用了反馈控制策略. 实验结果表明，本文提出的方法在满足所有约束的前提下，有效地降低了 OpenCL 应用的延迟.

作者贡献声明:曹坤负责研究方案设计、实验验证、初稿撰写;龙赛琴负责算法分析和论文修改;李哲涛提出指导意见,参与方案讨论以及论文审阅.

参 考 文 献

[1] Zhang Feng, Zhai Jidong, Chen Zheng, et al. Survey on performance analysis, optimization, and applications of heterogeneous fusion processors[J]. Journal of Software, 2020, 31(8): 2603−2624 (in Chinese)
(张峰, 翟季冬, 陈政, 等. 面向异构融合处理器的性能分析、优化及应用综述[J]. 软件学报, 2020, 31(8): 2603−2624)

[2] Hardkernel Co, Ltd. Introduction to Odroid-XU3 [EB/OL]. 2019 [2022-05-27].https://www.hardkernel.com/shop/odroid-xu3/

[3] Kaeli D R, Mistry P, Schaa D, et al. Heterogeneous Computing with OpenCL 2.0 [M]. San Francisco, CA: Morgan Kaufmann, 2015

[4] Singh A K, Prakash A, Basireddy K R, et al. Energy-efficient run-time mapping and thread partitioning of concurrent OpenCL applications on CPU-GPU MPSoCs[J]. ACM Transactions on Embedded Computing Systems, 2017, 16(5s): 1−22

[5] Wang Shaochung, Yu Linya, Her Li 'an, et al. Pointer-based divergence analysis for OpenCL 2.0 programs[J]. ACM Transactions on Parallel Computing, 2021, 8(4): 1−23

[6] Khalid Y N, Aleem M, Ahmed U, et al. Troodon: A machine-learning based load-balancing application scheduler for CPU–GPU system[J]. Journal of Parallel and Distributed Computing, 2019, 132: 79−94

[7] Chen Cen, Li Kenli, Ouyang Anjia, et al. FlinkCL: An OpenCL-based in-memory computing architecture on heterogeneous CPU-GPU clusters for big data[J]. IEEE Transactions on Computers, 2018, 67(12): 1765−1779

[8] Ahmed U, Lin J C W, Srivastava G, et al. Fuzzy active learning to detect OpenCL kernel heterogeneous machines in cyber physical systems[J]. IEEE Transactions on Fuzzy Systems, 2022, 30(11): 4618−4629

[9] Harichane I, Makhlouf S A, Belalem G. KubeSC-RTP: Smart scheduler for Kubernetes platform on CPU-GPU heterogeneous systems[J]. Concurrency and Computation: Practice and Experience, 2022, 34(21): 1−19

[10] Wang Siqi, Ananthanarayanan G, Mitra T. OPTiC: Optimizing collaborative CPU–GPU computing on mobile devices with thermal constraints[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018, 38(3): 393−406

[11] Pérez B, Stafford E, Bosque J L, et al. Auto-tuned OpenCL kernel co-execution in OmpSs for heterogeneous systems[J]. Journal of Parallel and Distributed Computing, 2019, 125: 45−57

[12] Isuwa S, Dey S, Singh A K, et al. TEEM: Online thermal-and energy-efficiency management on CPU–GPU MPSoCs [C] //Proc of the 22nd IEEE Design, Automation & Test in Europe Conf & Exhibition. Piscataway, NJ: IEEE, 2019: 438−443

[13] Maity S, Ghose A, Dey S, et al. Thermal-aware adaptive platform management for heterogeneous embedded systems[J]. ACM Transactions on Embedded Computing Systems, 2021, 20(5s): 1−28

[14] Navarro A, Corbera F, Rodriguez A, et al. Heterogeneous parallel for template for CPU–GPU chips[J]. International Journal of Parallel Programming, 2019, 47(2): 213−233

[15] Damschen M, Mueller F, Henkel J. Co-scheduling on fused CPU-GPU architectures with shared last level caches[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018, 37(11): 2337−2347

[16] Li Tiantian, Yu Ge, Song Jie. Optimization research on thermal and power management for real-time systems[J]. Journal of Computer Research and Development, 2016, 53(7): 1478−1492 (in Chinese)
(李甜甜, 于戈, 宋杰. 实时系统温度功耗管理的优化方法研究[J]. 计算机研究与发展, 2016, 53(7): 1478−1492)

[17] Huang Huang, Chaturvedi V, Quan Gang, et al. Throughput maximization for periodic real-time systems under the maximal temperature constraint[J]. ACM Transactions on Embedded Computing Systems, 2014, 13(2s): 1−22

[18] Kalaivani C T, Kalaiarasi N. Earliest deadline first scheduling technique for different networks in network control system[J]. Neural Computing and Applications, 2019, 31(1): 223−232

[19] Rosing T S, Mihic K, De Micheli G. Power and reliability management of SoCs[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2007, 15(4): 391−403

[20] Chantem T, Xiang Yun, Hu Sharon Xiaobo, et al. Enhancing multicore reliability through wear compensation in online assignment and scheduling [C] //Proc of the 16th IEEE Design, Automation & Test in Europe Conf & Exhibition. Piscataway, NJ: IEEE, 2013: 1373−1378

[21] Rubinstein R Y, Kroese D P. The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning [M]. Berlin: Springer, 2004

[22] Helton J C, Davis F J. Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems[J]. Reliability Engineering & System Safety, 2003, 81(1): 23−69

[23] Li Liying, Cong Peijin, Cao Kun, et al. Game theoretic feedback control for reliability enhancement of EtherCAT-based networked systems[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019, 38(9): 1599−1610

[24] Dey S, Singh A K, Wang Xiaohang, et al. User interaction aware reinforcement learning for power and thermal efficiency of CPU-GPU mobile MPSoCs [C] //Proc of the 23rd IEEE Design, Automation & Test in Europe Conf & Exhibition. Piscataway, NJ: IEEE, 2020: 1728−1733

[25] Samsung Electronics Co, Ltd. Introduction to Samsung Exynos 9810 mobile SoC [EB/OL]. 2018 [2022-07-10].https://www.notebookcheck.net/Samsung-Exynos-9810-SoC.276866.0.html

[26] Zhang Runjie, Stan M R, Skadron K. HotSpot 6.0: Validation, acceleration and extension [EB/OL]. 2015 [2022-07-10].https://www.cs.virginia.edu/~skadron /Papers/HotSpot60_TR.pdf

[27] Ma Yue, Zhou Junlong, Chantem T, et al. Improving reliability of soft real-time embedded systems on integrated CPU and GPU platforms[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019, 39(10): 2218−2229

[28] GitHub Inc. Hetero-Mark benchmark suit [EB/OL]. 2021 [2022-07-10].https://github.com/NUCAR-DEV/Hetero-Mark

[29] Grauer-Gray S, Killian W, Cavazos J, et al. PolyBench benchmark suit [EB/OL]. [2022-07-10].http://cavazos-lab.github.io/PolyBench-ACC/

[30] Bienia C, Kumar S, Singh J P, et al. The PARSEC benchmark suite: Characterization and architectural implications [C] //Proc of the 17th Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2008:72-81

[31] Bento M E C. A hybrid particle swarm optimization algorithm for the wide-area damping control design[J]. IEEE Transactions on Industrial Informatics, 2022, 18(1): 592−599

Cao Kun, born in 1991. PhD, associate professor. Member of CCF. His main research interests include Internet of things, edge computing, and cyber-physical systems.

曹　坤, 1991 年生. 博士, 副教授. CCF 会员. 主要研究方向为物联网、边缘计算、信息物理融合系统.

Long Saiqin, born in 1986. PhD, professor. Member of CCF. Her main research interests include cloud computing, edge computing, parallel and distributed systems, and Internet of things.

龙赛琴, 1986 年生. 博士, 教授. CCF 会员. 主要研究方向为云计算、边缘计算、并行和分布式系统、物联网.

Li Zhetao, born in 1980. PhD, professor, PhD supervisor. Member of CCF. His main research interests include cloud computing, intelligent network, and artificial intelligence.

李哲涛, 1980 年生. 博士, 教授, 博士生导师. CCF 会员. 主要研究方向为云计算、智能网络、人工智能.