Architectural Support for Task Dependence Management with Flexible Software Scheduling

灵活软件调度的任务依赖管理的架构支持

Emilio Castillo, Lluc Alvarez, Miquel Moreto, Marc Casas, etc

巴塞罗那超算中心, 西班牙, HPCA-2018

Table of contents

multi-core

- task parallelism: programmability, portability
- OpenMP : data-flow exection model

task granularity

- fine-grained parallelism: load balancing, flexibility, large software overhead

Solution

- pure hardware: limited adaptability
- pure software: low performance

Introduction Proposed Solution

硬件层

- 任务依赖管理器(Task Dependence Management, TDM)，一种软硬件协同设计的机制
- 在软件中允许灵活的任务调度策略，利用专用硬件加速运行时系统最耗时的活动
- 依赖管理单元(Dependency Management Unit, DMU) 通过一组表和列表来维护就绪任务的信息以及它们之间的依赖关系

ISA扩展

- 允许运行时系统通信任务创建、任务依赖关系和任务终结，并请求就绪的任务

软件层

- 准备执行的任务公开给运行时系统，运行时系统可以自由地部署任何软件调度策略

Introduction Related Works

Carbon "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," ISCA-2007

- task scheduler at hardware level
- task dependence management in software

Superscalar "Task superscalar: An out-of-order task pipeline," MICRO-2010

- offload all runtime system activities to architecture
- fixed FIFO policy -> limited flexibility of system

TDM advantages

- mitigate overheads in runtime system phases
- provide flexibility for software layer
- more adaptable, composable, capable for diffent scheduling policies

thread execution

- runtime system activity
- application task

thread model

- master: creation/schedule
- worker: running

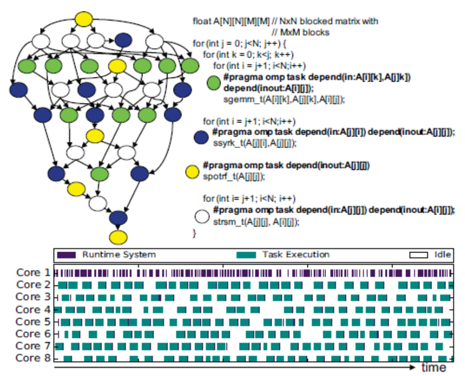


Figure 1: Cholesky task-based annotated code (right), task dependence graph (left), and execution timeline (bottom).

Background Characterizing Runtime System Activity

the cost of dependence management operations during task creation is crucial for performance because it determines the idle time in the whole execution

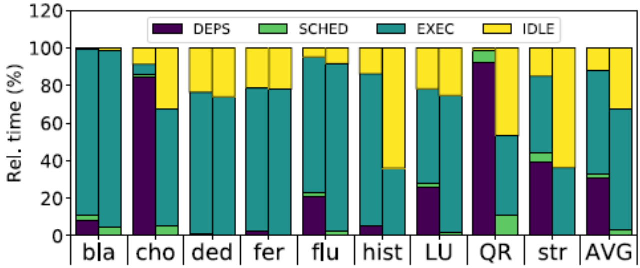


Figure 2: Execution time breakdown of the master and worker threads during the parallel execution. Different states represent dependence management operations during task creation and task finalization (DEPS), scheduling (SCHED), task execution (EXEC), and idle time (IDLE).

inst execution

- issued by runtime system
- in task creation/finalization phases
- to exchange info with DMU

4 new ISA instructions

1. create_task(task_desc)
2. add_dependence(task_desc, dep_addr, size, direction)
3. finish_task(task_desc)
   When a task finishes its execution, the runtime system uses this instruction to notify it to the architecture.
   The DMU wakes up the successors of the task and cleans up the information of the task and its dependences from its internal structures.
4. get_ready_task() -> task_desc, #succ

modules

1. TAT/DAT: Task/Dependence Alias Table
2. TT/DT: Tast/Dependence Table
3. SLA/DLA/RLA: Successor/Dependence/Reader List Array
4. RQ: Ready Queue

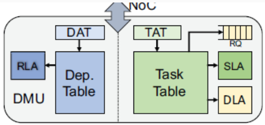


Figure 3: DMU architectural support overview.

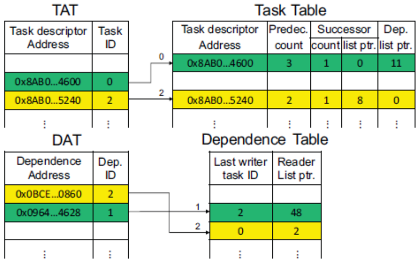


Figure 4: Overview of TAT, DAT, Task and Dependence Table. Two active elements are presented in each table.

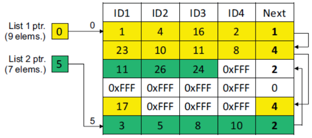
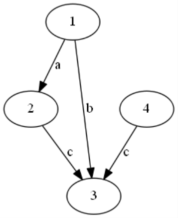


Figure 5: Overview of a generic list array.

TDM Design DMU Hardware Design




```
Data: taskID, depID, dir
Insert depID in dependence list of taskID;
if lastWriterID of depID is valid then
    Insert taskID in successor list of lastWriterID;
    Increment #succ of lastWriterID;
    Increment #pred of taskID;
end
if dir is In then
    Insert taskID in reader list of depID;
end
if dir is Out then
    for readerID in reader list of depID do
        Insert taskID in successor list of readerID;
        Increment #succ of readerID;
        Increment #pred of taskID;
    end
    Flush reader list of depID;
    Set lastWriterID of depID to taskID and mark valid;
end
```

Algorithm 1: Algorithm for *add_dependence* instruction.

Table: Example of DMU

depID	Last Writer Task ID	Reader List Array
a	1	2
b	1	3
c	2->4	3->3

taskID	depList	succList	#succ	#pred
1	a,b	2,3	1+1	0
2	a,c	3,4	1+1	1
3	b,c		0	1+1+1
4	c	3	1	1

Experimental Framework Full-System Simulation Infrastruture

Tools

- gem5 to simulate ARM full-system
- 32-core processor using out-of-order CPU
- McPAT for power consumption, DMU modeled using CACTI 6.0
- Ubuntu 14.04 with Nanos++ 0.10a runtime system

Benchmarks

- 5 from PARSECS:
  - Blackscholes/Streamcluster: fork-join
  - Dedup/Ferret: pipeline parallelism
  - Fluidanimate: 3D stencil
- 4 from HPC
  - Histogram
  - Cholesky/LU/QR

Flexible Scheduling with TDM Solution

Software Scheduler

1. FIFO
2. LIFO
3. Locality
    - data locality and min data movement
4. Successor
    - # successor > threshold with higher priority
    - overlapping I/O
5. Age

Optimization

1. Design Parameters Exploration: vs ideal case
2. inosensitive to DMU access latancy: 1/16 cycle -> 0.2%/0.9%
3. dynamic index-bit-selection: log(size of dependencies)

Flexible Scheduling with TDM Performance

Experiment Result

- vs software + FIFO
  - performance: 12.3%
  - EDP(Energy Delay Product): 20.4%
  - opt: Successor+TDM
- Carton/Superscalar/TDM:
  - performance: 1.9%/8.1%/12.3%
  - EDP: 5.1%/14.1%/20.4%

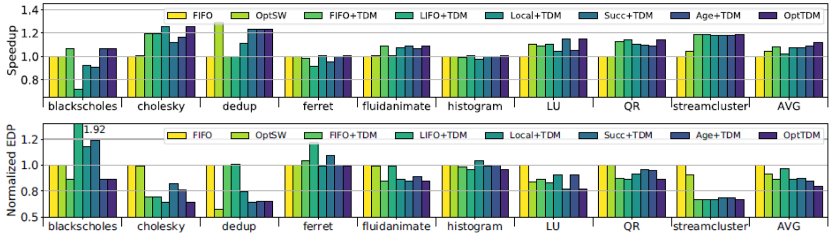


Figure 12: Speedup (top) and EDP reduction (bottom) with FIFO, LIFO, Locality-aware and Criticality-aware schedulers using software runtime system and TDM. Results are normalized to the software runtime system with a FIFO scheduler.

Tip : 比率 -> 几何平均 AVG

Architectural Support for Task Dependence Management with Flexible Software Scheduling

灵活软件调度的任务依赖管理的架构支持

Emilio Castillo, Lluc Alvarez, Miquel Moreto, Marc Casas, etc

巴塞罗那超算中心, 西班牙, HPCA-2018