# Task Scheduling For Multi-Core Asymmetric Systems

Pablo Ruiz
*Dept. of Electrical and Computer Engineering*
*University of Massachusetts*
Lowell, MA 01854, USA

## I. INTRODUCTION

### A. *Why it Matters?*

As processors become larger and more powerful, they reach a power limit at which cooling and maintaining performance starts becoming impractical. Therefore, since the power ceiling is somewhat known, the key factor to maximize performance becomes efficiency or in other words how much compute can a processor provide per watt of energy used. A higher efficiency level will not only mean that the processor can perform more compute with the same power but also less of the input energy is being converted into output heat. Mobile devices were the first to run into this problem due to their form factor, limited cooling, and reliance on batteries. In order to extract the most efficiency per watt, multi-core, asymmetric designs are becoming the norm. Since processors are changing schedulers must follow suit in order to use different cores for different purposes and guarantee that the potential for efficiency of the new hardware is maximized.

### B. *Proposed Task Scheduling Strategies*

This paper will explore several scheduling strategies that could benefit multi-core systems with asymmetric core layouts. The paper will progress from the simplest scheduling strategies with the least amount of overhead to the more complex, more accurate strategies with higher overhead compute costs. The proposed strategies are the following:

- ***Discussed:***
    - Criticality-Aware Scheduler (CATS)
- ***Proposed:***
    - Critical Path Scheduler (CPATH)
    - Hybrid Criticality Scheduler (HYBRID)
    - Dynamic implementation of Heterogeneous Earliest Finish Time (dHEFT)

## II. TASK DEPENDENCY GRAPH (TDG)

### A. *Task Dependency Chains*

A task dependency graph (TDG) is designed to clearly display the dependency chain for all tasks within an application as shown in figure 1. Each node represents a task to be performed.

Additionally, each edge represents a dependency between one task to another, meaning that the first task must be completed before the next task in a dependency chain can start computing. The numbers outside of each node represents its ID. Furthermore, the number within each node represents the task's priority level which can be computed and referred to in different ways by different scheduling strategies.

### B. *Critical and Non Critical Queues*

In figure 1, the filled (dashed) nodes are considered what is referred to as critical tasks. How critical tasks are chosen can vary depending on the scheduling strategy, but generally critical tasks are those that belong to the longest path of the TDG. At runtime, the longest path changes as tasks are completed so therefore whether a task is considered critical or not can change as other tasks and task chains are completed. Generally, in a system with performance cores (big cores) and efficiency cores (little cores), the critical tasks will be queued for the performance cores to compute while the non-critical tasks will be given to the efficiency cores. However, this can vary in some of the scheduling strategies.

## III. Criticality-aware scheduler (CATS)

CATS is the simplest of all the scheduling strategies that will be covered in this paper. In CATS the priority level of a task is referred to as the *bottom level*, or the number of "hops" between itself and a leaf node (node which does not have a child/dependent). Therefore, this means that the criticality of a task is determined by two conditions. The first is if its priority is higher or equal to the priority of the previous critical task. The seconds is if it is the highest priority child of the previous critical task. Additionally, when the application first starts the first critical task is the one with the highest priority which is equal to or greater than one. In this scheduling strategy, the critical tasks are always sent to the performance cores while the non-critical tasks are always sent to the efficiency cores. Ultimately, CATS incurs a relatively small amount of overhead since the priority levels are computed prior to the tasks running. However, CATS is nor the most accurate at finding the true current critical path since it assumes all tasks take an equal amount of time to complete.

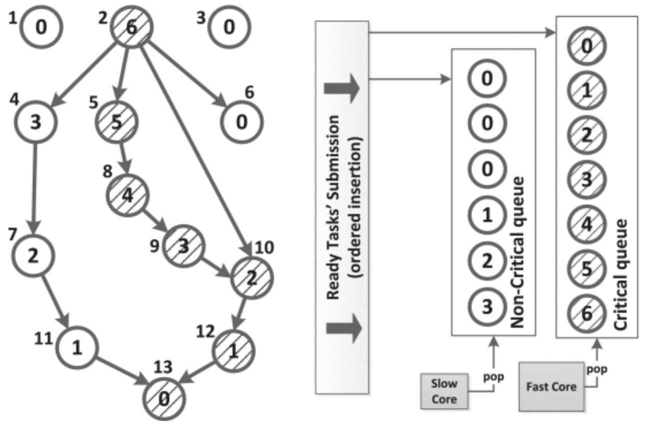
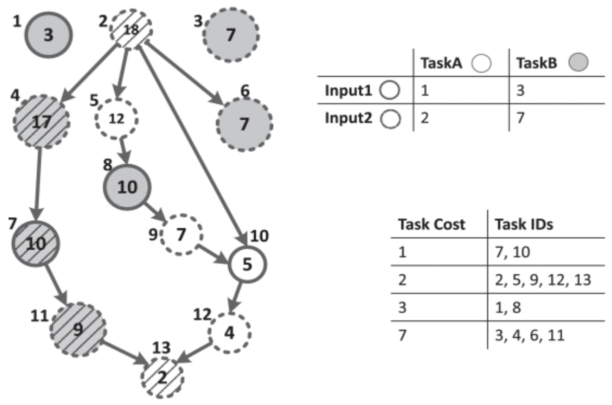


*Figure [1]: CATS Example. Task Dependency Graph.*

## IV. CRITICAL PATH SCHEDULER (CPATH)

CPATH is very similar to CATS with the only difference being how the priority levels of the tasks are computed and when they are computed. In this strategy, the criticality of a task path is determined by its estimated execution time rather than just its length. The execution time cost or priority level is referred to as the *bottom cost*. The execution time of a node is based on its task type and its input size. This method assumes that if these attributes are comparable the execution time will be comparable. CPATH attempts to have the most accurate task cost values always stored in a table as seen in figure 2. For this reason, it recomputes priority levels after each critical task computation. For this reason, it is more accurate than CATS at finding the true critical path.
The downside to this method is that it increases the scheduling overhead costs due to it dynamically calculating the priority levels after every task computation. This means that the next task has to wait until the priority calculations are done before executing which increases the compute overhead.



| | TaskA ○ | TaskB ● |
|---|---|---|
| Input1 ○ | 1 | 3 |
| Input2 ○ | 2 | 7 |

| Task Cost | Task IDs |
|---|---|
| 1 | 7, 10 |
| 2 | 2, 5, 9, 12, 13 |
| 3 | 1, 8 |
| 7 | 3, 4, 6, 11 |

*Figure [2]: CPATH Example. Task Dependency Graph w/ tt-is graph.*

## V. HYBRID CRITICAL SCHEDULER (HYBRID)

HYBRID is essentially a combination of CATS and CPATH strategies. It uses mainly CATS and includes the tasks execution time table to calculate priority levels only when it is available to reduce overhead. In CATS the priority levels are computed before hand when tasks are created. In CPATH after every critical task is completed the performance core has to be devoted to priority calculations before it can begin processing the next critical task. HYBRID attempts to calculate bottom cost like CPATH but it does it at task creation like CATS. This reduces the overhead but is not as accurate as CPATH in selecting the true critical path because it may lack information on the time cost of task type-input size combinations at the start of the application.
If time completion information is not available, it defaults to CATS and assigns that task a cost of one as shown in figure 3.

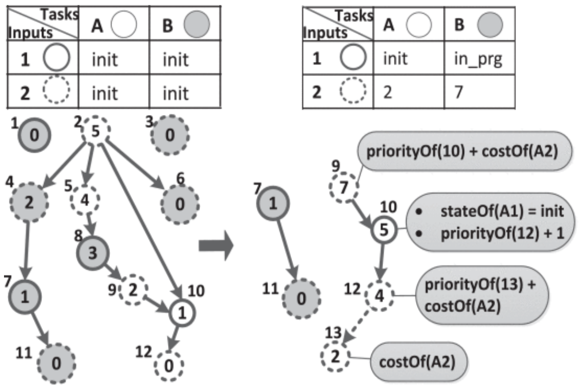


*Figure [3]: HYBRID Example. On the right figure notice how only the path that had an element added to it is recalculated.*

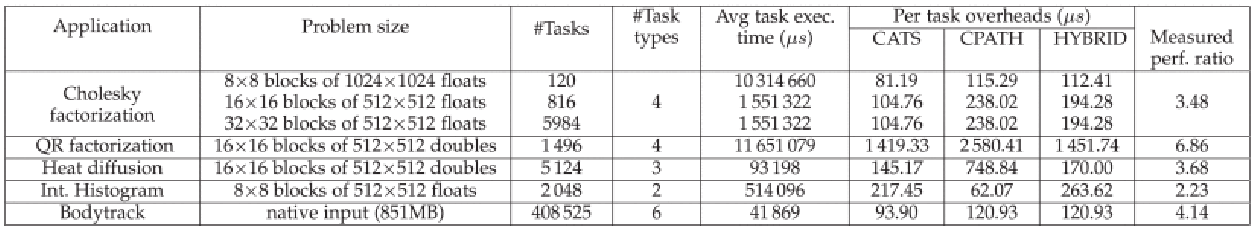
## VI. DYNAMIC HETEROGENEOUS EARLIEST FINISH TIME SCHEDULER (DHEFT)

DHEFT is more complex than any of the strategies discussed previously. Traditional HEFT strategies use the task computation and communication costs to calculate the priority of a task within a path. Additionally, it then chooses the processor core/type that will meet those costs in the shortest amount of time. This means that HEFT works very similarly to CPATH however it maintains a task execution time table for each and an additional task-core table which shows the cost of each task in each core type used in the system. HEFT assumes known task execution and communication times at compile time and thus all scheduling is done before tasks start executing. However, since in real applications execution and communication times are not known at compile time DHEFT was developed. In DHEFT, these tables are updated at runtime as tasks execute.
If no data is present for a given task/core combination, then a task will be sent to the first available core of that type to collect data until at least three data points are collected. Ultimately DHEFT should be the most efficient at finding the true critical path and assigning it to the most efficient core for that task. However, it is also the strategy with the highest scheduling overhead.

## VII. RESULTS

### A. *Testing Tasks*

To test the strategies outlined above a group of test applications with varying numbers of tasks and task types was selected. The average task execution time and calculated per task overhead is shown in table 1 along with all of the information necessary to qualify the tasks for each scheduling strategy. The last column in table 1 represents the performance ratio between performance and efficiency cores. All of the testing conducted for this paper was performed on a 8-core (4 Big, 4 Little) Odroid-XU3 development board.



| Application | Problem size | #Tasks | #Task types | Avg task exec. time ($\mu s$) | Per task overheads ($\mu s$) | | | Measured perf. ratio |
|---|---|---|---|---|---|---|---|---|
| | | | | | CATS | CPATH | HYBRID | |
| Cholesky factorization | 8×8 blocks of 1024×1024 floats | 120 | 4 | 10 314 660 | 81.19 | 115.29 | 112.41 | 3.48 |
| | 16×16 blocks of 512×512 floats | 816 | | 1 551 322 | 104.76 | 238.02 | 194.28 | |
| | 32×32 blocks of 512×512 floats | 5984 | | 1 551 322 | 104.76 | 238.02 | 194.28 | |
| QR factorization | 16×16 blocks of 512×512 doubles | 1 496 | 4 | 11 651 079 | 1 419.33 | 2 580.41 | 1 451.74 | 6.86 |
| Heat diffusion | 16×16 blocks of 512×512 doubles | 5 124 | 3 | 93 198 | 145.17 | 748.84 | 170.00 | 3.68 |
| Int. Histogram | 8×8 blocks of 512×512 floats | 2 048 | 2 | 514 096 | 217.45 | 62.07 | 263.62 | 2.23 |
| Bodytrack | native input (851MB) | 408 525 | 6 | 41 869 | 93.90 | 120.93 | 120.93 | 4.14 |

*Table [1]. Presents the per task data gathered for each application using each of the proposed schedulers.*

Additionally, figure 4 shows a breakdown of the task number and task cost for each of the testing applications that were selected. From these figures it is clear that an application set with a wide variety of tasks and task costs was chosen in order to the test the scheduler thoroughly in most realistic operation conditions.

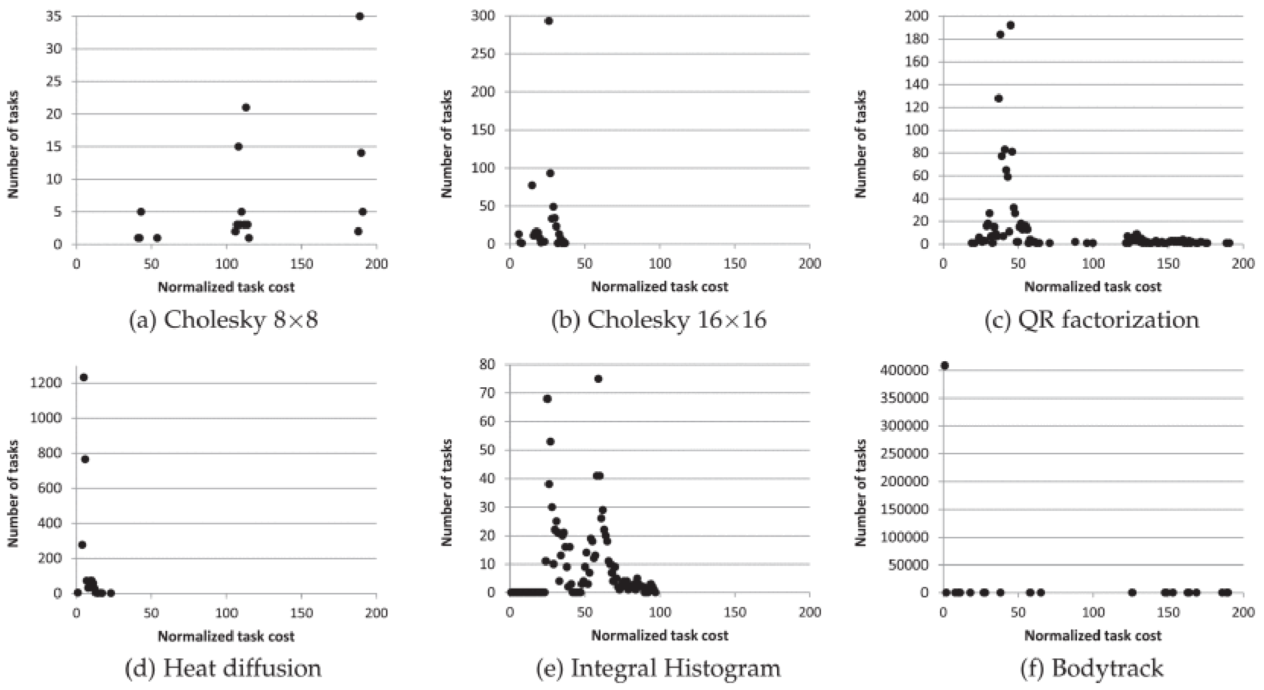


*Figure [4]: Graphs representing the number of tasks and their cost for each application.*

## B. *Speed Results*

As shown in figure 5 the proposed scheduling strategies were tested against a traditional homogenous system scheduler which is referred to as BG and shown in solid black. Additionally, in figure 5 the y-axis represents the execution speed as a multiple of the performance observed in a homogenous system made up of only efficiency cores. As can be observed all of the proposed schedulers performed better than BG in most tasks. The only notable exceptions are CPATH in the Heat application and DHEFT in the Bodytrack applications which both contain a relatively small number of tasks.

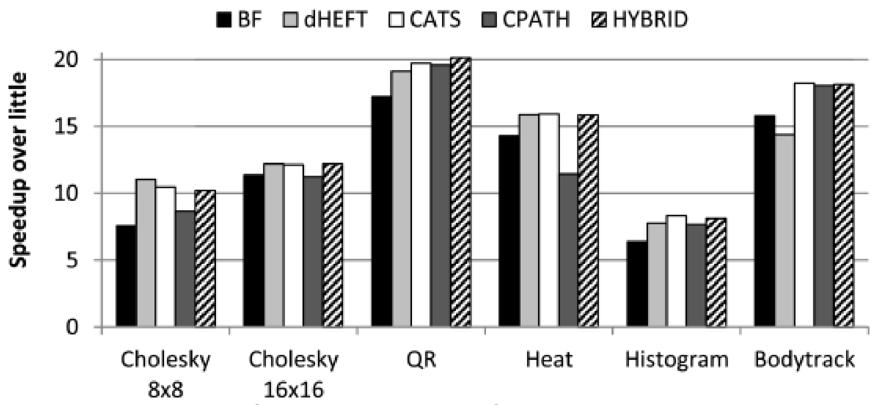


*Figure [5]: Completion speed of each application for each scheduler compared to completing the whole application using a single little core.*

Finally, figure 6 shows the average performance in the selected applications outlined above of each of the scheduling strategies in a variety of core configurations which all add up to a total of 8 cores. In this graph it is again clear that all of the schedulers tend to outperform the traditional scheduler BG in most instances except for a few notable exceptions in which CPATH and DHEFT lag slightly behind. As shown in the previous figures it seems as though dynamic strategies with large scheduling overheads such as CPATH and DHEFT do not perform any better than simpler strategies and that in some types of applications they can perform worse than a homogenous scheduler. Overall however, it is clear that the lead of the proposed schedulers over the traditional scheduler increases at the heterogeneity of the cores increases, proving that they are more efficient in heterogenous processing systems.

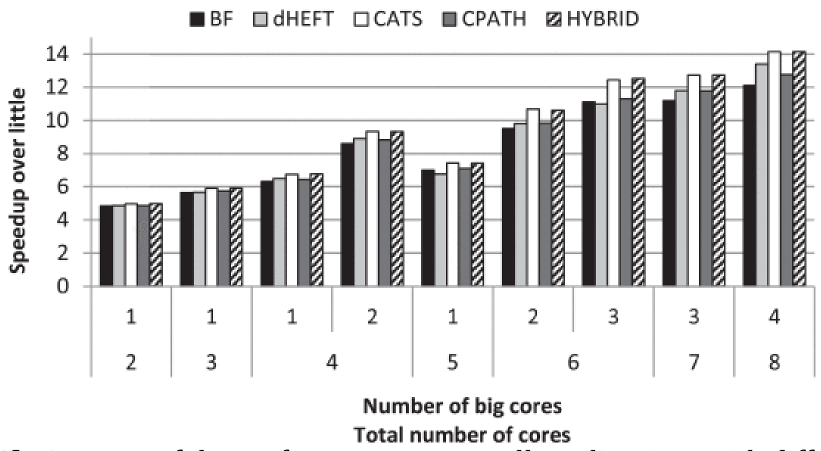


*Figure [6]: Average of the performance across all applications with different core configurations.*

## VIII. CONCLUSION

All of the discussed scheduling strategies improve on the performance of a traditional homogenous scheduler as the number of cores in a heterogenous processor increases. When the heterogeneity is low and a large percentage of the system cores are homogenous, scheduling strategies with high overhead like CPATH and DHEFT can fall behind traditional homogenous schedulers, however. All the scheduling approaches shown in this presentation obtain the information they need to schedule tasks at runtime, making them more suited to use in real systems than other strategies like traditional HEFT. These results make it clear that more complex scheduling approaches are not always beneficial due to their increased overhead, especially when an application has a smaller number of tasks.

## REFERENCES

[1] Chronaki, K., Rico, A., Casas, M., Moreto, M., Badia, R. M., Ayguade, E., Labarta, J., & Valero, M. (2017). Task scheduling techniques for asymmetric multi-core systems. *IEEE Transactions on Parallel and Distributed Systems, 28*(7), 2074–2087. https://doi.org/10.1109/tpds.2016.2633347