CHRT: a Criticality- and Heterogeneity-Aware Runtime System for Task-Parallel Applications

Myeonggyun Han
School of ECE, UNIST
hmg0228@unist.ac.kr

Jinsu Park
School of ECE, UNIST
jinsupark@unist.ac.kr

Woongki Baek
School of ECE, UNIST
wbaek@unist.ac.kr

***Abstract*—Heterogeneous multiprocessing (HMP) is an emerging technology for high-performance and energy-efficient computing. While task parallelism is widely used in various computing domains from the embedded to machine-learning computing domains, relatively little work has been done to investigate the efficient runtime support that effectively utilizes the criticality of the tasks of the target application and the heterogeneity of the underlying HMP system with full resource management.**

To bridge this gap, we propose a criticality- and heterogeneity-aware runtime system for task-parallel applications (CHRT). CHRT dynamically estimates the performance and power consumption of the target task-parallel application and robustly manages the full HMP system resources (i.e., core types, counts, and voltage/frequency levels) to maximize the overall efficiency. Our experimental results show that CHRT achieves significantly higher energy efficiency than the baseline runtime system that employs the breadth-first scheduler and the state-of-the-art criticality-aware runtime system.

I. Introduction

Heterogeneous multiprocessing (HMP) has surfaced as a promising solution for high-performance and energy-efficient computing [8]. HMP processors consist of multiple types of cores with different performance and power characteristics. The HMP system software analyzes the characteristics of the applications and schedules them on the most efficient cores to maximize the overall efficiency.

Task parallelism is an effective technique to dynamically exploit the parallelism available in the application by executing independent tasks in parallel [1, 7, 9]. Task parallelism is widely used in a variety of computing domains across the embedded to machine-learning computing domains [4].

Prior work has extensively investigated the efficient runtime support for task-parallel applications [3, 5, 10, 11]. However, relatively little work has been done to simultaneously exploit the criticality of the tasks of the target application and the heterogeneity of the underlying HMP system with full resource management.

To bridge this gap, this work proposes a criticality- and heterogeneity-aware runtime system for task-parallel applications (CHRT). CHRT dynamically estimates the performance and power consumption of the target task-parallel application and robustly manages the full HMP system resources to significantly improve the efficiency of the target application.

Specifically, this paper makes the following contributions:

- We propose a criticality- and heterogeneity-aware runtime system for task-parallel applications (CHRT).
- We design and implement a prototype of CHRT.
- We quantify the effectiveness of CHRT.

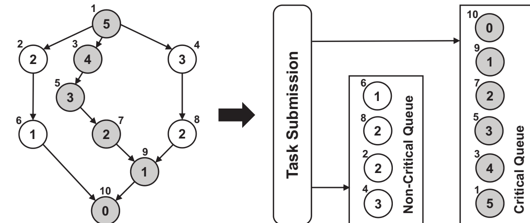

Fig. 1. Task dependency graph and critical path

II. Background

Task Parallelism: Task parallelism decomposes a job into multiple tasks and employs multiple worker threads that execute independent tasks in parallel to improve the overall throughput [1, 7, 9]. When a worker thread encounters a code block marked as a task, the runtime system creates a new task. Tasks may have dependency with other tasks. In such cases, their execution is deferred until all the dependency is resolved. Tasks without any dependency are inserted into the ready queues.

Worker threads fetch and execute tasks from the ready queues based on the task scheduling algorithm. For instance, the breadth-first scheduler [6], which is employed by the baseline runtime system evaluated in this work, gives higher priority to the tasks that have been encountered earlier and executes them first. Task parallelism is supported by widely-used parallel programming frameworks such as OpenMP [1], Cilk [7], and TBB [9].

Task-dependency graphs (TDGs) are commonly used to represent the dependency between tasks. In a TDG, nodes represent tasks and edges denote the dependency between tasks. As with the prior work [5], we define the *critical path* as the longest path in a given TDG and the *critical tasks* as the tasks on the critical path. All the other paths and tasks are defined as the non-critical paths and tasks. In addition, as with the prior work [5], we assume that the runtime system maintains two types of the ready queues – (1) the critical queue, which contains critical ready tasks and (2) the non-critical queue, which contains non-critical ready tasks. Figure 1 shows an example of a task-dependency graph, where the numbers in and next to each node denote the criticality (i.e., depth) and ID of the corresponding task.

Heterogeneous Multiprocessing: A single-ISA heterogeneous multiprocessing (HMP) system comprises cores that implement the same ISA but exhibit different micro-architectural characteristics such as the instruction-issue width and the capability of out-of-order execution [8]. A core cluster is a group of the cores with the same micro-architectural characteristics. In

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

this work, we assume that the underlying HMP system consists of two types of core clusters. The core cluster with higher (or lower) performance and power consumption is referred as the *big* (or *little*) core cluster. We assume that the big and little core clusters comprise N_B and N_L cores and provide N_{f_B} and N_{f_L} different voltage/frequency levels and the underlying HMP system supports per-cluster DVFS.

For the target task-parallel application, we assume that the big and little cores running at their reference frequencies (i.e., $f_{B,ref}$ and $f_{L,ref}$) can process $C_{B,f_{B,ref}}$ and $C_{L,f_{L,ref}}$ per unit time (e.g., 1 second). Then, the *computation capacities* of the big and little cores running at f_B and f_L are defined as follows – $C_{B,f_B} = C_{B,f_{B,ref}} \cdot \frac{f_B}{f_{B,ref}}$ and $C_{L,f_L} = C_{L,f_{L,ref}} \cdot \frac{f_L}{f_{L,ref}}$. CHRT dynamically profiles the target task-parallel application and computes $C_{B,f_{B,ref}}$ and $C_{L,f_{L,ref}}$ without requiring any kind of offline data.

III. Design and Implementation

CHRT mainly comprises two components – the application programming interface (API) and the runtime system.

A. The CHRT Application Programming Interface

The CHRT API consists of the three functions (i.e., `begin_app`, `end_task`, and `end_app`). To exploit the adaptive task parallelism supported by CHRT, the target task-parallel application or library code needs to be instrumented using the API. In this work, we employ the OmpSs library code [6] as the baseline code and instrument it using the API.

The `begin_app` function is used to specify the beginning of the target task-parallel application. It establishes the interprocess communication (IPC) between the CHRT runtime system and the target task-parallel application. It takes an input parameter (i.e., `n`), which is used to control the communication frequency between the CHRT runtime system and the target application. Specifically, each worker thread sends a message to the CHRT runtime system after completing `n` tasks, which is configurable.

The `end_task` function is used to mark the end of processing a task. When each of the worker threads completes the processing of a task, it invokes the `end_task` function. Based on the aforementioned parameter (i.e., `n`), each worker thread periodically sends the runtime data such as the critical and non-critical task counts to the CHRT runtime system. The runtime system uses the runtime data received from the target application to dynamically analyze its characteristics (e.g., the ratio of the non-critical task count to the critical task count, which will be discussed in Section III-B) and estimate its performance. Finally, the `end_app` function is used to specify the end of the target task-parallel application.

B. The CHRT Runtime System

The CHRT runtime system dynamically manages the full HMP system resources to significantly improve the efficiency of the target task-parallel application. Specifically, the HMP system resources controlled by CHRT are the core types, counts, and voltage/frequency levels of each core cluster.

We define the system state space (i.e., $\mathbf{S}$) as all the possible combinations of the system resources. We represent a system state (i.e., $\vec{s}$) as a vector with four elements – $\vec{s} = (n_B, n_L, f_B, f_L)$, where n_B, n_L, f_B, and f_L denote the big core count, little core count, big core cluster frequency, and little core cluster frequency allocated to the target task-parallel application. The CHRT runtime system comprises the performance estimator, power estimator, and runtime manager.

Performance Estimator: For a system state of interest (i.e., $\vec{s} = (n_B, n_L, f_B, f_L)$), the performance estimator of CHRT estimates the performance of the target task-parallel application. CHRT allocates a dedicated big core to execute the tasks on the critical path and utilizes some of the remaining resources to execute the tasks on the non-critical paths. Therefore, the computation capacities allocated for the critical (C_C) and non-critical tasks (C_N) are computed as follows – $C_C = C_{B,f_B}$ and $C_N = (n_B - 1) \cdot C_{B,f_B} + n_L \cdot C_{L,f_L}$.

For the target task-parallel application, let k be the ratio (i.e., $k = \frac{T_N}{T_C}$) of the non-critical task count (T_N) to the critical task count (T_C). To ensure the fair progress of the critical and non-critical paths, CHRT aims to satisfy the capacity constraint as follows – $C_N \geq k \cdot C_C$.

The execution time (t_{f_B}) of each critical task, which determines the overall performance of the target task-parallel application, is inversely proportional to the computation capacity (C_C) allocated for critical tasks. Specifically, the performance estimator estimates the performance of the target application using Equation 1.

$$t_{f_B} = \frac{1}{C_C} \quad (1)$$

Power Estimator: For a system state of interest, the power estimator of CHRT estimates the power consumption of the underlying HMP system. The power estimator employs a linear regression model that assumes that the power consumption of each cluster is proportional to the average per-core utilization of the corresponding cluster.

Equation 2 shows the linear regression model, where α_{B,f_B} and β_{B,f_B} are the regression coefficients for the big core cluster, α_{L,f_L} and β_{L,f_L} are the regression coefficients for the little core cluster, U_B and U_L are the average per-core utilization of the big and little core clusters, and γ denotes the power consumption of the other components (e.g., memory). For every available frequency of each core cluster, the corresponding regression coefficients are determined based on the data collected through the offline experiments with our microbenchmark. The microbenchmark is fully configurable in that it can stress the underlying HMP system in any arbitrary setting (e.g., core type, count, and utilization).

$$P = \alpha_{B,f_B} \cdot U_B + \beta_{B,f_B} + \alpha_{L,f_L} \cdot U_L + \beta_{L,f_L} + \gamma \quad (2)$$

Runtime Manager: The CHRT runtime manager dynamically explores the system state space to find a system state that significantly improves the efficiency of the target task-parallel application. Since the system state space rapidly grows with the system parameters (i.e., core types, counts, and voltage/frequency levels), the runtime manager explores the system state space based on an incremental and greedy algorithm, similarly to the hill-climbing algorithm.

Algorithm 1 shows the main function of the runtime manager. The runtime manager executes in two phases – the *adaptation* and *idle* phases.

During the adaptation phase, the runtime manager checks if an adaptation period is reached (Line 6) for every new heartbeat received from the target task-parallel application. A

Algorithm 1 The global variables and main function

```
1: prevState ← initState; currState ← initState
2: prevEfficiency ← 0; currEfficiency ← 0
3: phase ← adaptation; adaptCount ← 0; f_B ← f_{B,init}
4: procedure MAIN
5:   while true do
6:     if isAdaptPeriod() then
7:       currEfficiency ← measureEfficiency()
8:       if phase = adaptation then                ▷ Adaptation phase
9:         exploreSystemStateSpace()
10:      else                                      ▷ Idle phase
11:        if triggerReadaptation() = true then
12:          phase ← adaptation
13:          resetVariables()
14:      applyState()
```

Algorithm 2 The exploreSystemStateSpace function

```
1: procedure EXPLORESYSTEMSTATESPACE
2:   if currEfficiency − prevEfficiency < δ_e then
3:     currState ← prevState
4:     phase ← idle
5:   else if adaptCount < N_{f_B} then
6:     f_B ← getNext(F_B)                 ▷ F_B: a set of all f_B's
7:     adaptCount ← adaptCount + 1
8:     prevState ← currState
9:     prevEfficiency ← currEfficiency
10:    currState ← getNextState()
11:    if currState = invalidState then
12:      currState ← prevState
13:      phase ← idle
14:  else
15:    phase ← idle
```

new adaptation period is reached when the runtime manager receives H new heartbeats, which is configurable. When an adaptation period is reached, the runtime manager measures the efficiency of the target application of the current period by invoking the `measureEfficiency` function (Line 7), which quantifies the efficiency based on the power consumption data collected from the power sensors equipped in the underlying HMP system (Section IV) and the number of the completed critical tasks.

The runtime manager then explores the system state space by invoking the `exploreSystemStateSpace` function in Algorithm 2. It first checks if the measured efficiency of the current period is lower than that of the previous period (Line 2) by a predefined threshold (i.e., δ_e, which is set to zero in this work). If so, the runtime manager rolls back to the previous state and transitions to the idle phase (Lines 3–4).

Otherwise (i.e., currEfficiency $-$ prevEfficiency $\geq \delta_e$), the runtime manager continues to explore the system state space (Lines 6–13) in an incremental manner. In each adaptation period, the runtime manager first determines the big core frequency (f_B) by iterating each of the available big core frequencies (F_B in Line 6) at a time. The key idea behind determining f_B first among all the HMP system resources is to set the performance of the target task-parallel application (Equation 1) to a certain level. After setting f_B, the runtime manager invokes the `getNextState` function (Line 10) to determine the next system state to transition.

For a given $f_B = f$, the `getNextState` function explores the system state space to find a system state that is expected to maximize the energy efficiency. Algorithm 3 shows the `getNextState` function, which evaluates the efficiency of

Algorithm 3 The getNextState function

```
1: procedure GETNEXTSTATE
2:   bestState ← (N_B, N_L, f_B, f_{L,max})
3:   bestEfficiency ← estimateEfficiency(bestState)
4:   for n_B in {1, ···, N_B} do
5:     for n_L in {0, ···, N_L} do
6:       for f_L in {f_{L,min}, ···, f_{L,max}} do
7:         candidateState ← (n_B, n_L, f_B, f_L)
8:         if checkConstraint(candidateState) = false then
9:           continue
10:        estEfficiency ← estimateEfficiency(candidateState)
11:        if estEfficiency > bestEfficiency then
12:          bestState ← candidateState
13:          bestEfficiency ← estEfficiency
14: label_exit:
15:   estCurrEff ← estimateEfficiency(currState)
16:   if bestEfficiency < estCurrEff then
17:     bestState ← invalidState
18:   return bestState
```

the all the possible system states with $f_B = f$. For a system state of interest (i.e., $\vec{s} = (n_B, n_L, f, f_L)$), the runtime manager first checks if $\vec{s}$ satisfies the capacity constraint (i.e., $C_N \geq k \cdot C_C$) and skips it if it fails to satisfy the constraint.

If $\vec{s}$ satisfies the capacity constraint, the runtime manager invokes the `estimateEfficiency` function (Line 10) to estimate the energy efficiency of $\vec{s}$. Specifically, the runtime manager performs the following by invoking the `estimateEfficiency` function.

The runtime manager first computes the computation capacities allocated to execute the critical (i.e., $C_C = C_{B,f_B}$) and non-critical (i.e., $C_N = (n_B - 1) \cdot C_{B,f_B} + n_L \cdot C_{L,f_L}$) tasks. The runtime manager then computes the system usage effectiveness ($e = \frac{C_N}{k \cdot C_C}$), which is defined as a ratio of the computation capacity (i.e., C_N) allocated for the non-critical tasks to the computation capacity (i.e., $k \cdot C_C$) that is actually required for executing the non-critical tasks.

Based on the system usage effectiveness (e), the average per-core utilization of the big (U_B) and little (U_L) core clusters is estimated as follows – $U_B = \frac{1 + \frac{(n_B - 1)}{e}}{N_B}$ and $U_L = \frac{n_L}{N_L \cdot e}$. Intuitively, with lower e, the utilization of the allocated cores increases because the allocated computation capacity becomes closer to the required computation capacity.

The runtime manager then estimates the power consumption of the target task-parallel application by substituting the estimated utilization of the big and little core clusters into Equation 2. In addition, the performance of the target application can be estimated using Equation 1. Putting it all together, the energy consumption can be estimated using Equation 3, when the system state is set to $\vec{s} = (n_B, n_L, f, f_L)$.

$$E_{\vec{s}} = P_{\vec{s}} t_f = \frac{\alpha_{B,f} U_B + \beta_{B,f} + \alpha_{L,f_L} U_L + \beta_{L,f_L} + \gamma}{C_C} \quad (3)$$

The `getNextState` function evaluates the energy efficiency of all the candidate system states using Equation 3 and determines the best system state when $f_B = f$. If the best system state is worse than the current system state in terms of the efficiency, the `getNextState` function returns the invalid system state (Lines 16–17 in Algorithm 3). In such a case, the runtime manager terminates the adaptation phase and transitions to the idle phase (Lines 11–13 in Algorithm 2). Further, when the runtime manager has evaluated all the system states with all the possible big-core cluster frequencies

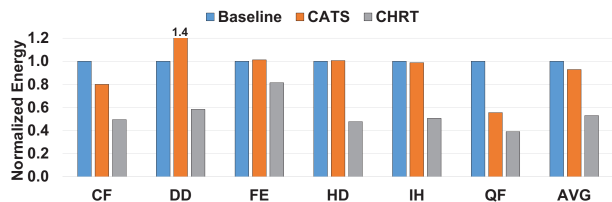

Fig. 2. Normalized energy consumption

(Lines 14–15 in Algorithm 2), it terminates the adaptation phase and transitions to the idle phase.

IV. Evaluation

Methodology: To quantify the effectiveness of CHRT, we use a full heterogeneous multiprocessing (HMP) system, the ODROID-XU3 embedded development board. The board is equipped with the Exynos 5422 processor, which implements ARM's big.LITTLE architecture. The processor comprises four big cores ($N_B = 4$) and four little cores ($N_L = 4$). We use the following dependency-intensive task-parallel benchmarks from the BAR [2] and PARSECSs [4] benchmark suites – Cholesky Factorization (`CF`) [2], Dedup (`DD`) [4], Ferret (`FE`) [4], Heat Diffusion (`HD`) [2], Integral Histogram (`IH`) [2], and QR Factorization (`QF`) [2].

Experimental Results: Through our quantitative evaluation, we aim to investigate the energy-efficiency gains achieved by CHRT. We quantify the efficiency of the following runtime systems – the baseline runtime system that employs the breadth-first scheduler [6], the state-of-the-art criticality-aware runtime system (i.e., CATS) [5], and CHRT. Due to the space limit, we refer to [5] for additional details on CATS.

Figure 2 shows the energy consumption of the aforementioned runtime systems, normalized to that of the baseline version. We observe the following data trends.

First, CHRT significantly outperforms the baseline version in terms of energy efficiency. Specifically, CHRT achieves 47.0% higher energy efficiency than the baseline version on average (i.e., geometric mean) across all the evaluated benchmarks. This is mainly because the baseline version lacks the capability of exploiting the criticality of the tasks of the target application and the heterogeneity of the underlying HMP system. For example, while `QF` is highly sensitive to criticality (i.e., k is relatively small) and heterogeneity (i.e., the ratio of $C_{B,f_{B,ref}}$ to $C_{L,f_{L,ref}}$ is significantly higher than the other benchmarks), the baseline version is unable to exploit such properties. In contrast, CHRT effectively utilizes the criticality and heterogeneity to dynamically control the full HMP system resources, achieving significantly higher energy efficiency across all the evaluated benchmarks (including `QF`).

Second, CHRT achieves significantly higher energy efficiency than CATS. Specifically, CHRT achieves 43.0% higher energy efficiency than CATS on average. This is mainly because CATS lacks the performance and power estimation functionality and controls only a subset (i.e., core types) of the full HMP system resources when executing the target task-parallel application. For an example, since the parallelism available in `DD` is rather low, CATS schedules and executes most of the tasks on the big core cluster, resulting in sub-optimal energy efficiency. In contrast, CHRT robustly manages the full HMP system resources through the guidance of the performance and power estimators, significantly outperforming CATS in terms of energy efficiency.

V. Related Work

Prior work has extensively investigated the efficient runtime support for task-parallel applications [3, 5, 10, 11]. While insightful, most of the prior work lacks the runtime support that simultaneously utilizes the criticality of the tasks of the target application and the heterogeneity of the underlying HMP system.

The prior work close to ours is the work (i.e., criticality-aware dynamic task scheduling (CATS)) presented in [5]. While insightful, CATS lacks the performance and power estimation models for the target task-parallel application, which are highly important for energy-efficiency optimization. Further, CATS only manages a part (i.e., core types) of the full HMP system resources. In contrast, our work significantly differs in that CHRT employs the performance and power estimation models to robustly guide the entire optimization process and effectively manages the full HMP system resources (i.e., core types, counts, and voltage/frequency levels), considerably outperforming CATS in terms of energy efficiency.

Prior work has extensively investigated the architectural and system software techniques to improve the efficiency of conventional applications on HMP systems [8, 12]. Our work differs in that it proposes an energy-efficient runtime system for task-parallel applications in the context of HMP.

VI. Conclusions

This work proposes CHRT, a criticality- and heterogeneity-aware runtime system for task-parallel applications. CHRT dynamically estimates the performance and power consumption of the target task-parallel application and effectively manages the full HMP system resources. Our quantitative evaluation demonstrates the effectiveness of CHRT in that it significantly outperforms the baseline and state-of-the-art runtime systems in terms of the energy efficiency.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2014R1A1A1005969). Woongki Baek is the corresponding author.

References

[1] E. Ayguadé et al. "The Design of OpenMP Tasks". In: *IEEE Trans. Parallel Distrib. Syst.* (2009).
[2] *BSC Application Repository*. https://pm.bsc.es/projects/bar.
[3] Y. Cao et al. "Stable Adaptive Work-Stealing for Concurrent Multi-core Runtime Systems". In: HPCC '11. 2011.
[4] D. Chasapis et al. "PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite". In: *ACM Trans. Archit. Code Optim.* (2015).
[5] K. Chronaki et al. "Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures". In: ICS '15. 2015.
[6] A. Duran et al. "Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures". In: *Parallel Processing Letters* (2011).
[7] M. Frigo et al. "The Implementation of the Cilk-5 Multithreaded Language". In: PLDI '98. 1998.
[8] R. Kumar et al. "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction". In: MICRO 36. 2003.
[9] J. Reinders. *Intel Threading Building Blocks*. 2007.
[10] H. Ribic et al. "Energy-efficient Work-stealing Language Runtimes". In: ASPLOS '14. 2014.
[11] H. Sun et al. "Efficient Adaptive Scheduling of Multiprocessors with Stable Parallelism Feedback". In: *IEEE Trans. Parallel Distrib. Syst.* (2011).
[12] J. Yun et al. "HARS: A Heterogeneity-aware Runtime System for Self-adaptive Multithreaded Applications". In: DAC '15. 2015.