HAP: A Heterogeneity-Conscious Runtime System for Adaptive Pipeline Parallelism

Jinsu Park and Woongki Baek^(✉)

School of ECE, UNIST, Ulsan, South Korea
{jinsupark,wbaek}@unist.ac.kr

Abstract. Heterogeneous multiprocessing (HMP) is a promising solution for energy-efficient computing. While pipeline parallelism is an effective technique to accelerate various workloads (e.g., streaming), relatively little work has been done to investigate efficient runtime support for adaptive pipeline parallelism in the context of HMP. To bridge this gap, we propose a heterogeneity-conscious runtime system for adaptive pipeline parallelism (HAP). HAP dynamically controls the full HMP system resources to improve the energy efficiency of the target pipeline application. We demonstrate that HAP achieves significant energy-efficiency gains over the Linux HMP scheduler and a state-of-the-art runtime system and incurs a low performance overhead.

1 Introduction

Heterogeneous multiprocessing (HMP) is rapidly emerging as a promising solution for energy efficient computing [9]. The key idea of HMP is to provide multiple types of cores that are architecturally designed and optimized for different performance and energy efficiency goals. To maximize the energy efficiency of HMP, its system software must be able to dynamically analyze the characteristics of the applications and schedule them on the most efficient cores. Recent work has demonstrated that application-directed dynamic optimization is effective for improving the energy efficiency of parallel applications (e.g., web browser [17] and DBMS [10]).

Pipeline parallelism is an effective software technique to accelerate the execution of tasks, which are difficult to parallelize using conventional techniques (e.g., data parallelism) due to the internal dependence between their subtasks. Pipeline parallelism decomposes the entire task into multiple subtasks and overlaps the execution of the subtasks for different work items to improve the overall throughput on parallel systems. Recent work has shown that various workloads (e.g., data mining [1] and streaming [4]) can be effectively accelerated through pipelining.

To achieve the best possible energy efficiency of pipeline parallelism on HMP, heterogeneity-conscious runtime support is crucial. While there has been prior work that investigates the runtime support for adaptive pipeline parallelism, they have limitations in that they target symmetric multiprocessing (SMP) without

any support for HMP [14] or they control only a subset of system resources, leaving key HMP system resources unmanaged (e.g., no dynamic voltage and frequency scaling (DVFS) of heterogeneous cores) [8, 15] and achieving suboptimal energy efficiency.

To bridge this gap, we propose a heterogeneity-conscious runtime system for adaptive pipeline parallelism (HAP). Unlike the aforementioned prior approaches, HAP dynamically controls the full system resources (i.e., core types, counts, and voltage/frequency levels) of the underlying HMP system to maximize the energy efficiency of the target pipeline application. In addition, HAP provides a simple and easy-to-use application programming interface (API) that programmers can use to exploit the energy-efficient adaptive pipeline parallelism supported by HAP. Through our quantitative evaluation, we demonstrate the effectiveness of HAP. Specifically, this paper makes the following contributions:

- We propose a heterogeneity-conscious runtime system for adaptive pipeline parallelism. HAP manages the full system resources (i.e., core types, counts, and voltage/frequency levels) of the underlying HMP system for energy-efficient adaptive pipeline parallelism.
- We implement and evaluate HAP based on a full HMP system. Prior work is based on architectural simulators that lack the modeling of the entire system software stack such as the operating system [8, 15]. We investigate the interaction between the Linux HMP scheduler and pipeline applications, demonstrating its performance and energy inefficiency.
- We quantify the effectiveness of HAP using six pipeline applications and a fully-configurable microbenchmark. Our quantitative evaluation shows that HAP significantly outperforms the Linux HMP scheduler and a state-of-the-art runtime system for adaptive pipeline parallelism [14, 15] in terms of energy efficiency. In addition, our experimental results demonstrate that HAP robustly detects and adapts to the phase changes of the target pipeline application and incurs a low performance overhead.

2 Background

Heterogeneous Multiprocessing: A single-ISA heterogeneous multiprocessing system consists of cores that implement the same ISA but exhibit different architectural characteristics such as the instruction issue width [9]. A core cluster is defined as a group of the cores with the same architectural characteristics. In this work, for simplicity, we assume that the HMP system consists of the two types of core clusters, similarly to the ARM’s big.LITTLE processor [7]. The core cluster that consists of the cores with higher (or lower) performance and power consumption is referred as the *big* (or *little*) core cluster. We assume that the big and little clusters consist of N_B and N_L cores, respectively.

In addition, we assume that the big and little clusters provide N_{f_B} and N_{f_L} voltage/frequency levels, which can be dynamically controlled in software. While per-core DVFS is promising, we consider cluster-level DVFS in this work because per-core DVFS is not yet widely supported in commodity processors.

Pipeline Parallelism: Pipeline parallelism decomposes a task into subtasks and overlaps their execution to improve the overall throughput. A pipeline application consists of one or more stages, each of which executes its assigned subtask for processing work items. Adjacent stages communicate through the work queues. Each pipeline stage consists of one or more worker threads. Each stage worker thread retrieves a work item from its input queue, processes it, and inserts the processed work item into its output queue, which is used as the input queue of the next stage.

The throughput of a pipeline stage is defined as the number of work items that can be processed by the stage per unit time. If the average processing time of a work item in stage s is t_s and the number of worker threads is N_s , the throughput of the stage s is computed as $\lambda_s = \frac{N_s}{t_s}$. The *limiter* stage of a pipeline application is defined as the stage whose throughput is the minimum among all the stages. The *non-limiter* stages are defined as all the stages except for the limiter stage. The overall throughput of the pipeline application is limited by the throughput of the limiter stage. This indicates that accelerating the non-limiter stages by allocating excessive hardware resources may significantly degrade energy efficiency without achieving any performance gain.

Heterogeneity-conscious runtime support is crucial to achieve the best possible energy efficiency of pipeline parallelism on HMP systems due to the following reasons. First, since the system state space rapidly grows with various characteristics of the target pipeline application (e.g., stage count and worker count) and the underlying HMP system (e.g., core types, counts, and voltage/frequency levels), it is nearly infeasible to develop a profiling-based static system for energy-efficient pipeline parallelism. Second, since the target pipeline application may exhibit widely different behaviors depending on its input data and program phases, it is critical to dynamically adapt its execution in a heterogeneity-conscious and energy-efficient manner, guided by runtime information.

3 Design and Implementation

HAP mainly consists of two components – the application programming interface (API) and the runtime system. For simplicity, we describe the design and implementation of HAP with an assumption that the underlying HMP system consists of two types of core clusters (i.e., big and little). However, we believe that our proposed techniques can be generalized for various HMP systems (e.g., more core types).

3.1 The HAP API

HAP provides the four API functions summarized in Table 1. To exploit the energy-efficient adaptive pipeline parallelism supported by HAP, programmers need to instrument their applications using the API functions. The `begin_app` function is used to notify the beginning of the target pipeline application. The `begin_app` function establishes the interprocess communication (IPC) between

Table 1. The HAP application programming interface (API)

Function	Description
<code>begin_app(nStage)</code>	Beginning of the target pipeline application
<code>begin_work()</code>	Beginning of the work-item processing
<code>end_work(stageId)</code>	End of the work-item processing
<code>end_app()</code>	End of the target pipeline application

the target pipeline application and the HAP runtime system and sends the information on the target application (e.g., the number of pipeline stages) to the HAP runtime system through IPC.

The `begin_work` function is used to mark the beginning of a work-item processing performed by the calling stage worker thread. The `begin_work` function reads the time (t_B) when the processing of the work item is about to begin. The `end_work` function reads the time (t_E) when the processing of the work item has ended and sends the data such as the work-item processing time ($t_W = t_E - t_B$) and the stage ID of the calling thread to the HAP runtime system through IPC. Finally, the `end_app` function is used to notify the end of the target pipeline application.

The current implementation of the HAP API builds upon the Application Heartbeats framework [6], which provides a well-established interface to communicate messages called *heartbeats* between processes. We extend the Application Heartbeats framework to encode and communicate pipeline-specific information such as the stage ID and work-item processing time (t_W).

3.2 The HAP Runtime System

The HAP runtime system manages the full system resources to significantly enhance the energy efficiency of the target pipeline application. The system resources managed by the HAP runtime system are the core types, counts, and the voltage/frequency levels of each cluster. The HAP runtime system divides the system resources into two groups – the ones allocated to the limiter stage of the target pipeline application and the others allocated to the non-limiter stages. We define the system state space as all the possible combinations of the system resources that can be allocated to the limiter stage. Figure 1 shows the overall architecture of the HAP runtime system, which consists of three components – the performance estimator, power estimator, and runtime manager.

Performance Estimator: For a system state of interest, the performance estimator of HAP estimates the performance of the target pipeline application. The performance estimator assumes that each of the stage worker threads of the target pipeline application is assigned with its dedicated core. The performance estimator employs a linear model that assumes that the performance of each worker thread of the limiter stage is proportional to the computation capacity

Fig. 1. The overall architecture of the HAP runtime system

of its allocated core. If the performance ratio of the big core to the little core is r_0 at the frequency of f_0 , the computation capacities of the big and little cores running at the frequencies of f_B and f_L are $r_0 \cdot \frac{f_B}{f_0}$ and $\frac{f_L}{f_0}$. The performance ratio (r_0) can be either statically determined based on the architectural characteristics (e.g., instruction issue width) of heterogeneous cores or dynamically determined based on the runtime information collected during the execution of the target pipeline application. As discussed later, r_0 is dynamically computed based on the runtime information encoded in the heartbeats.

Power Estimator: For a system state of interest, the power estimator of HAP estimates the power consumption of the underlying HMP system. The power estimator estimates the power consumption of the big and little core clusters based on a linear regression model, which assumes that the power consumption of each cluster is proportional to the sum of the utilization of the cores in the corresponding cluster. The regression coefficient values are determined for every available frequency of each cluster. The regression coefficients are computed based on the data collected through the offline experiments with our microbenchmark that can stress the underlying HMP system with different configurations (e.g., core type, count, frequency, and utilization). Currently, the power estimator assumes that the power consumption of other hardware components (e.g., memory) is constant, which can be extended with more sophisticated models. In summary, the power estimator uses Eq. 1 to estimate the power consumption of the underlying HMP system.

$$P = \alpha_{B,f_B} \cdot \sum_{i=0}^{N_B-1} U_{B,i} + \beta_{B,f_B} + \alpha_{L,f_L} \cdot \sum_{i=0}^{N_L-1} U_{L,i} + \beta_{L,f_L} + \gamma \quad (1)$$

The power estimator assumes that all the cores allocated to the limiter stage are fully utilized because, by its definition, it is the performance bottleneck among all the stages. To estimate the utilization of the cores allocated to the non-limiter stages, the power estimator sorts the non-limiter stage worker threads in an increasing order of the throughput and the cores allocated to the non-limiter stages in a non-increasing order of the computation capacity, respectively. The power estimator assumes that the runtime manager establishes the one-to-one

mapping of each worker thread to core in the sorted order, which is the actual scheduling performed by the runtime manager. Based on the queuing theory [12], the utilization of each core allocated to a non-limiter stage s is approximated to be $U_s = \frac{\lambda_{lim} \cdot t_s}{N_s}$, where λ_{lim} , t_s , and N_s are the throughput of the limiter stage, the work-item processing time, and the worker thread count of the stage s .

We note that the utilization of cores where the non-limiter stages are scheduled can be computed more precisely based on the queuing theory extended for heterogeneous servers [5] because of the different processing capabilities of the stage worker threads scheduled on different clusters. However, we decide to use an approximate solution because the computational complexity of the precise solution is high and the accuracy of the approximate solution is expected to be reasonable [5]. By substituting the estimated utilization of all the cores into Eq. 1, the power estimator estimates the power consumption of the target pipeline application for a system state of interest.

Runtime Manager: The HAP runtime manager explores the system state space to find an efficient system state that significantly reduces the energy consumption of the target pipeline application. The system state space rapidly grows with the worker thread count of the limiter stage and architectural parameters of the underlying HMP system (e.g., core types, counts, and voltage/frequency levels). For instance, if the limiter stage consists of two worker threads and the underlying HMP system consists of four big cores with N_{f_B} voltage/frequency levels and four little cores with N_{f_L} voltage/frequency levels, the number of all the system states is $N_{f_B} + N_{f_B} \cdot N_{f_L} + N_{f_L}$. Due to the large system state space, the runtime manager explores the system state space based on an incremental and greedy algorithm, inspired by the hill-climbing algorithm [13].

The runtime manager executes in two phases – the *adaptation* and *idle* phases. Algorithm 1 shows the pseudocode for the runtime manager. During the adaptation phase, the runtime manager checks if an adaptation period has been reached (Line 7) for every new heartbeat generated when a stage worker thread finishes the processing of a work item. The adaptation phase consists of three sub-phases – the initial, observation, and exploration sub-phases.

The runtime manager runs in the *initial* sub-phase until the first adaptation period is reached. At the end of the initial sub-phase (Line 9), the runtime manager retrieves the information on the target pipeline application such as the thread ID of each stage worker thread. The runtime manager then sets the affinity of all the threads of each stage to each of the available cores, transitioning to the observation sub-phase.

The runtime manager runs in the *observation* sub-phase until the second adaptation period is reached. At the end of the observation sub-phase (Line 11), the runtime manager retrieves the dynamic information on the target pipeline application such as the throughput of each stage using the data encoded in the received heartbeats and the equation discussed in Sect. 2. The runtime manager then identifies the limiter stage, sets the system state to the initial state, and transitions into the exploration sub-phase.

Algorithm 1. The pseudocode for the HAP runtime manager

```

1: limiter  $\leftarrow$  invalidStageId; pScore  $\leftarrow$  0
2: firstAdtPeriod  $\leftarrow$  false; cScore  $\leftarrow$  0
3: pState  $\leftarrow$  obsState; cState  $\leftarrow$  obsState
4: phase  $\leftarrow$  adaptation; subPhase  $\leftarrow$  initial
5: procedure MAIN
6:   while true do
7:     if isAdaptPeriod() then
8:       if phase = adaptation then
9:         if subPhase = initial then
10:           subPhase  $\leftarrow$  observation
11:         else if subPhase = observation then
12:           subPhase  $\leftarrow$  exploration
13:           limiter  $\leftarrow$  findLimiter()
14:           firstAdtPeriod  $\leftarrow$  true
15:           cState  $\leftarrow$  initState
16:         else  $\triangleright$  Exploration sub-phase
17:           cScore  $\leftarrow$  computeScore()
18:           exploreSystemStateSpace()
19:         else  $\triangleright$  Idle phase
20:           if triggerReadaptation() = true then
21:             phase  $\leftarrow$  adaptation
22:             subPhase  $\leftarrow$  initial
23:             resetVariables()
24:           applyState()
25: procedure EXPLORESYSTEMSTATESPACE
26:   if firstAdtPeriod = false  $\wedge$  cScore - pScore
    <  $\delta_c$  then
27:     cState  $\leftarrow$  pState
28:     phase  $\leftarrow$  idle
29:   else
30:     if firstAdtPeriod = true then
31:       firstAdtPeriod  $\leftarrow$  false
32:       pState  $\leftarrow$  cState
33:       pScore  $\leftarrow$  cScore
34:       cState  $\leftarrow$  getNextState()
35:     if cState = invalidState then
36:       cState  $\leftarrow$  pState
37:     phase  $\leftarrow$  idle
38:   procedure GETNEXTSTATE
39:     bestState  $\leftarrow$  invalidState
40:     bestScore  $\leftarrow$  minScore
41:     candStates  $\leftarrow$  cState  $\cup$  getNeighborStates(cState)
42:     for candState in candStates do
43:       estScore  $\leftarrow$  estimateScore(candState)
44:       if estScore > bestScore then
45:         bestState  $\leftarrow$  candState
46:         bestScore  $\leftarrow$  estScore
47:     if bestState = cState then
48:       bestState  $\leftarrow$  invalidState
49:     return bestState

```

During the *exploration* sub-phase (Line 16), the runtime manager explores the system state space to find an efficient system state that results in the significantly reduced energy consumption of the target pipeline application. The runtime manager explores the system state space based on an incremental and greedy algorithm. At each adaptation period,¹ the runtime manager invokes the `exploreSystemStateSpace` function, which subsequently calls the `getNextState` function to determine the next system state to transition. The runtime manager adapts the system state in an incremental manner in that the candidate system states are generated by incrementally changing the system resources allocated to the limiter stage from the current system state.

Specifically, the runtime manager considers the system states that are within the Manhattan distance d from the current system state in the three dimensional system state space (i.e., the big core count allocated to the limiter stage,² the frequencies of the big and little core clusters) as the candidate system states (Line 41). For instance, with the current system state of (n_B, f_{B_i}, f_{L_j}) and $d = 1$, the candidate system states are $(n_B + 1, f_{B_i}, f_{L_j})$, $(n_B - 1, f_{B_i}, f_{L_j})$, $(n_B, f_{B_{i+1}}, f_{L_j})$, $\dots$, and $(n_B, f_{B_i}, f_{L_{j-1}})$. With larger d , the runtime manager

¹ At the end of the first adaptation period of the exploration sub-phase, the runtime manager computes the performance ratio (r_0) of the big core to the little core based on the work-item processing time (t_W) data encoded in the heartbeats generated by the stage worker threads scheduled on the big and little core clusters.

² Since the little core count of the limiter stage can be determined from its big core count, we do not consider its little core count when computing d .

explores the system state space more exhaustively at the potential cost of higher performance overheads and instability due to abrupt system state changes.

The runtime manager adapts the system state in a greedy manner in that it chooses the next system state as the one that is estimated to result in the highest energy efficiency among all the candidate system states (Lines 42–46).³ The energy efficiency (i.e., joules per processed work item) of each candidate system state is estimated to be $\frac{P_{Est}}{\lambda_{Est}}$, where λ_{Est} and P_{Est} are its performance and power consumption estimated through the performance and power estimators.

The runtime manager transitions to the idle phase in the following two cases. First, if the energy efficiency of the current period is lower than the previous period, the runtime manager restores the previous system state and transitions to the idle state (Lines 26–28). The energy efficiency of the current period is computed based on the actual energy consumption data collected using the sensors discussed in Sect. 4 and the throughput data. Second, if none of the candidate states are expected to achieve higher energy efficiency than the current system state (Lines 35 and 47), the runtime manager transitions to the idle phase.

During the idle phase (Line 19), the runtime manager executes the target pipeline application without performing any adaptation but keeps monitoring the application to detect its phase changes. When detecting a program phase change, the runtime manager terminates the idle phase and triggers the entire adaption process again (Lines 20–23). To detect phase changes, the runtime manager computes the work ratio (r_W) of the limiter stage to the non-limiter stages. If the work ratios between the consecutive periods differ by r_{th} and N_R times in a row, the runtime manager determines that the program phase of the target pipeline application has changed and transitions to the adaptation phase to find a new efficient system state. Unless stated otherwise, d , r_{th} , and N_R are set to 5, 25% and 3.

4 Evaluation

Methodology: To quantify the effectiveness of HAP, we use a full heterogeneous multiprocessing (HMP) system, the ODROID-XU3 embedded development board. The board is equipped with the Exynos 5422 processor based on the ARM’s big.LITTLE architecture [7]. The processor consists of the four big cores (i.e., $N_B = 4$) and the four little cores (i.e., $N_L = 4$). The board is installed with Xubuntu 14.04 and the Linux kernel 3.10.69, which implements the HMP scheduler. The configurable frequency ranges of the big and little clusters are 0.2 – 2.0 GHz and 0.2 – 1.4 GHz, respectively. The board is equipped with sensors that periodically sample the power consumption of the big cluster, little cluster, memory, and GPU, which we use to construct the linear regression model of the power estimator and to measure the energy consumption of the HMP system during the execution of the target pipeline application.

³ Note that HAP can be generalized to perform optimizations based on other metrics (e.g., energy-delay product) by customizing the `estimateScore` function (Line 43).

We use the following six pipeline benchmarks, some of which are modified to exploit pipeline parallelism – **blackscholes** (BL) [3], **binomialoptions** (BO) [3], **bzip2** (BZ) [2], **dedup** (DD) [1], **ferret** (FR) [1], **montecarlo** (MC) [3]. The number of stages of BL, BO, BZ, DD, FR, and MC are 3, 3, 3, 5, 6, and 3, respectively. We also use a microbenchmark that is fully configurable in terms of pipeline parameters such as the number of stages, workers per stage, workload per worker.

Our evaluation aims to investigate the following. First, we quantify how much energy efficiency gain can be achieved through the use of HAP. Second, we evaluate the effectiveness of the re-adaptation functionality of HAP when the target pipeline application has multiple distinct program phases. Third, we investigate the sensitivity of the energy efficiency and performance overhead of HAP to the search distance parameter (d), which controls the exhaustiveness of the system space exploration.

Fig. 2. Normalized energy

Energy Efficiency: We evaluate the energy efficiency of HAP. We run the each benchmark with the following five OS or runtime versions – (1) the Linux HMP scheduler with the lowest big and little core frequencies (S-MIN), (2) the Linux HMP scheduler with the highest big and little core frequencies (S-MAX), (3) the Linux HMP scheduler with the big and little core frequencies that result in the best energy efficiency among all the possible combinations of the minimum, medium, and maximum frequencies (S-BEST),⁴ (4) feedback-directed pipeline parallelism (FDP), which implements the runtime system proposed in [14, 15],⁵ and (5) HAP. In addition, to investigate the effectiveness of HAP with different system utilization levels, we configure each benchmark in the following two settings – (1) *full subscription*, in which the worker thread counts of the limiter and each of the non-limiter stages are set to $N_B + N_L - S + 1$ and 1, where S is the number of stages and (2) *moderate subscription*, in which the worker thread counts of the limiter and each of the non-limiter stages are set to 2 and 1.

⁴ Due to the large system space which requires infeasibly long time for collecting profiled data, we selectively use the most representative frequencies (i.e., min, medium, and max) to determine the configuration for S-BEST.

⁵ Due to space limit, we refer to [14, 15] for more details on FDP.

Figure 2(a) shows the energy consumption of the five OS and runtime versions normalized to S-MIN with the full subscription setting, demonstrating the following data trends. First, HAP significantly outperforms the Linux HMP scheduler in terms of energy efficiency. Specifically, HAP reduces the energy consumption of the target pipeline applications by 42.4, 64.8, and 20.8 % on average (i.e., geometric mean), compared with the S-MIN, S-MAX, and S-BEST versions. HAP outperforms S-BEST mainly due to the performance inefficiency of the current version of the Linux HMP scheduler. For some benchmarks (e.g., BL), we observe that the Linux HMP scheduler often heavily biases CPU-intensive stage worker threads to the big cores even when the little cores are idle, eventually causing performance and energy efficiency degradation due to the load imbalance.

Second, HAP significantly outperforms FDP, which is a state-of-the-art runtime system for adaptive pipeline parallelism. Specifically, HAP reduces the energy consumption of the target pipeline applications by 63.8 % on average, compared with FDP. This is mainly because FDP lacks the capability of controlling voltage/frequency levels of heterogeneous core clusters, which are critical hardware knobs for achieving high energy efficiency. In contrast, HAP manages the full system resources (i.e., core types, counts, and voltage/frequency levels), significantly improving the energy efficiency of the target pipeline applications.

Figure 2(b) shows the energy consumption of the five OS and runtime versions normalized to S-MIN with the moderate subscription setting. HAP continues to achieve higher energy efficiency gains over the other OS and runtime versions with moderate subscription. Specifically, HAP reduces the energy consumption of the target pipeline applications by 55.3, 67.5, 32.0, and 63.4 % on average, compared with the S-MIN, S-MAX, S-BEST, and FDP versions. Since the system is less utilized with moderate subscription, HAP discovers more opportunities for reducing the energy consumption of the target pipeline application (e.g., setting the frequency of the unused core cluster to the lowest level), achieving higher energy efficiency gains than the case with full subscription. In summary, our experimental results show that HAP is effective in that it significantly outperforms all the other OS and runtime versions in terms of energy efficiency.

Fig. 3. Effectiveness of re-adaptation

Effectiveness of Re-adaptation: To evaluate the effectiveness of the re-adaptation functionality of HAP, we use a microbenchmark, which is configured to exhibit three distinct phases. Figure 3(a) shows the runtime behavior of the microbenchmark. At $t = 23.0$, the microbenchmark transitions to the second phase in which the work ratio (r_W) of the limiter to the non-limiter stages significantly changes. HAP robustly detects the phase change and accordingly adapts the system state after observing that the three consecutive samples of r_W are consistent (i.e., $N_R = 3$). At $t = 48.8$, the microbenchmark transitions to the third phase in which one of the non-limiter stages becomes the new limiter stage. HAP also robustly detects the phase change and accordingly performs adaptations. Figure 3(b) demonstrates the effectiveness of the re-adaptation functionality of HAP in that HAP significantly outperforms all the other OS and runtime versions in terms of energy efficiency, including a variant of HAP (i.e., HAP-NR) with which the re-adaptation functionality is intentionally disabled for illustrative purposes.

Fig. 4. Sensitivity to the search distance

Sensitivity to the Search Distance: Finally, we investigate the sensitivity of the energy efficiency and performance overhead of HAP to the search distance (d) parameter. Figure 4(a) shows the average (i.e., geometric mean) energy consumption of HAP across all the evaluated benchmarks, normalized to OS-MIN when d varies from 1 to 7. With larger d , the energy efficiency of HAP generally improves because it explores the system state more exhaustively. When d is sufficiently large (i.e., $d > 5$), the energy efficiency of HAP slightly decreases as d increases. This is mainly because HAP may converge to a slightly suboptimal system state when the system state changes too abruptly with larger d . Nevertheless, HAP consistently provides significant energy-efficiency gains over the Linux HMP scheduler both in the full and moderate subscription settings.

To quantify the performance overhead of HAP, Fig. 4(b) shows the sensitivity of the CPU utilization of HAP to the search distance. With larger d , the CPU utilization of HAP tends to gradually increase because it explores the system state space more exhaustively. However, the CPU utilization of HAP is insignificant (i.e., $< 1.0\%$) across all the configurations. Interestingly, with sufficiently large d (i.e., $d > 4$), the CPU utilization of HAP slightly decreases.

This is mainly because HAP converges faster with sufficiently large d and then consumes significantly less CPU cycles afterward. In summary, our experimental results demonstrate that HAP is an effective runtime system for adaptive pipeline parallelism in that it significantly improves energy efficiency, robustly adapts to program phase changes, and incurs a low performance overhead.

5 Related Work

Prior work has proposed runtime techniques for adaptive pipeline parallelism [8, 14, 15]. While insightful, the proposed techniques target runtime support for symmetric multiprocessing (SMP) systems [14] or lack the management of full system resources (e.g., no DVFS) of HMP systems [8, 15], resulting in sub-optimal energy efficiency as quantified by our experimental results. Further, the techniques proposed in [8, 15] have been evaluated using architectural simulators without in-depth investigation of the interaction among the target pipeline application, runtime, and OS. Our work differs in that HAP effectively manages the full system resources (core types, counts, and voltage/frequency levels) and is implemented and evaluated based on a real HMP system with the full system software stack.

Prior work has proposed architectural [9] and system software [11, 16] techniques to enhance the power and/or energy efficiency of conventional applications on HMP systems. Our work differs in that we propose an energy-efficient runtime system for adaptive pipeline parallelism in the context of HMP. In addition, recent work has investigated application-level techniques to improve the energy efficiency of the web browser [17] and DBMS [10]. While similar in that they utilize the application-level knowledge to enhance energy efficiency, our work differs as HAP targets efficient runtime support for adaptive pipeline parallelism.

6 Conclusions

This work presents HAP, a heterogeneity-conscious runtime system for adaptive pipeline parallelism. HAP dynamically controls the full system resources of the underlying HMP system to maximize the energy efficiency of the target pipeline application. In addition, HAP provides a simple and easy-to-use application programming interface that programmers can use to exploit the energy efficient adaptive parallelism supported by HAP. Our quantitative evaluation demonstrates the effectiveness of HAP in that it significantly outperforms the Linux HMP scheduler and the state-of-the-art runtime system for adaptive pipeline parallelism in terms of energy efficiency, robustly adapts to the phase changes of the target pipeline application, and incurs a small performance overhead. As our future work, we plan to extend HAP by investigating more advanced search algorithms that explore the system state space with higher coverage and efficiency.

Acknowledgements. This research was supported by ICT R&D program of MSIP/IITP (B0101-16-0661).

References

1. Bienia, C., et al.: The PARSEC benchmark suite: characterization and architectural implications. In: PACT 2008 (2008)
2. BZIP2SMP. <http://bzip2smp.sourceforge.net/>
3. CUDA Samples. <http://docs.nvidia.com/cuda/cuda-samples/>
4. Gordon, M.I., et al.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: ASPLOS XII (2006)
5. Gumbel, H.: Waiting lines with heterogeneous servers. *Oper. Res.* **8**(4), 504–511 (1960)
6. Hoffmann, H., et al.: Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In: ICAC 2010 (2010)
7. Je, B.: Big.LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration. In: DAC 2012 (2012)
8. Joao, J.A., et al.: Bottleneck identification and scheduling in multithreaded applications. In: ASPLOS XVII (2012)
9. Kumar, R., et al.: Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In: MICRO 36 (2003)
10. Mühlbauer, T., et al.: Heterogeneity-conscious parallel query execution: getting a better mileage while driving faster! In: DaMoN 2014 (2014)
11. Muthukaruppan, T.S., et al.: Hierarchical power management for asymmetric multi-core in dark silicon era. In: DAC 2013 (2013)
12. Navarro, A., et al.: Analytical modeling of pipeline parallelism. In: PACT 2009 (2009)
13. Skiena, S.S.: *The Algorithm Design Manual*, 2nd edn. Springer-Verlag, London (2008)
14. Suleman, M.A., et al.: Feedback-directed pipeline parallelism. In: PACT 2010 (2010)
15. Suleman, M.A.: An asymmetric multi-core architecture for efficiently accelerating critical paths in multithreaded programs. Ph.D. thesis. University of Texas at Austin (2010)
16. Yun, J., et al.: HARS: A heterogeneity-aware runtime system for self-adaptive multithreaded applications. In: DAC 2015 (2015)
17. Zhu, Y., et al.: High-performance and energy-efficient mobile web browsing on Big/Little systems. In: HPCA 2013 (2013)