Energy Efficient Job Scheduling in Single-ISA Heterogeneous Chip-Multiprocessors

Ying Zhang[1], Lide Duan[3], Bin Li[2], Lu Peng[1], Srinivasan Sadagopan[3]
[1] Department of Electrical and Computer Engineering
[2] Department of Experimental Statistics
Louisiana State University, Baton Rouge, LA, USA
{yzhan29, bli, lpeng}@lsu.edu
[3] Advanced Micro Devices, Inc
Austin, TX, USA
{lide.duan, sadagopan.Srinivasan}@amd.com

Abstract

In recent years, single-ISA heterogeneous chip multiprocessors (CMP) consisting of big high-performance cores and small power-saving cores on the same die have been proposed for the exploration of high energy-efficiency. On such heterogeneous platforms, an appropriate runtime scheduling policy lies at the heart of program executions to benefit from the processor heterogeneity. To date, most prior works addressing this problem concentrate on the performance enhancement; however, they lack detailed justification of the runtime energy consumption and do not result in the most energy-efficient execution all the time. In this work, we pay attention to reducing the energy consumption for workloads running on heterogeneous CMPs and propose a scheduling algorithm based on dynamic execution behaviors to exploit better energy-efficiency. Our strategy is capable of significantly reducing the energy consumption while delivering comparable performance to a recently proposed heterogeneous scheduler (MLP-ratio), thus improving the energy-efficiency impressively.

Keywords

heterogeneous architecture, scheduling, energy efficiency; statistical modeling

1. Introduction

Recently, the ever increasing power crisis makes the energy-efficiency a first-order concern in addition to raw performance in a wide spectrum of computing platforms ranging from large-scale data centers to small portable devices, thus necessitating further optimization of the energy-efficiency in future computing environments. Considering the diversity of program characteristics, conventional CMPs which consist of multiple identical cores might not be the perfect candidate for the exploitation. In this situation, computer architects propose the "heterogeneous CMP", where processor cores with disparate architectures are integrated on the same silicon, as an alternative design paradigm of traditional CMPs. It is widely acknowledged that heterogeneous chip multi-processors can effectively improve the energy-efficiency compared to homogeneous CMPs [5][10].

A heterogeneous CMP can be implemented in various manners, while the single Instruction-Set Architecture (ISA) organization is a representative design paradigm. A single-ISA heterogeneous CMP is usually composed of big cores equipped with complex out-of-order issue logic and sufficient computing resources and small cores on which the executions are driven by simple in-order pipelines. Put it another way, big cores provide high performance by consuming more power while the small cores reduce power dissipation at the expense of slower execution. To more effectively utilize the core heterogeneity and leverage their respective advantages, an appropriate job scheduler that is responsible for dynamic program-to-core assignment is in high demand. In prior works addressing this problem, the program relative performance between big and small cores is widely adopted as the heuristic to guide the scheduling [2][5]. Specifically, programs that gain impressive performance boost on faster cores are selected to run on big cores; on the contrary, programs demonstrating mild performance improvement on big cores are chosen to execute on small cores. When applications with such distinctive features are concurrently running on a heterogeneous CMP system, a good scheduler will be capable of making the right decision and significantly improve the overall performance. Correspondingly, workloads falling into this category are considered as scheduling-sensitive [5].

Table 1. Power and performance ratios for *bzip2* and *vpr*

Program	Big core power(W)	Small core power (W)	Performance ratio
bzip2	24.64	9.18	2.51
vpr	18.97	10.32	2.48

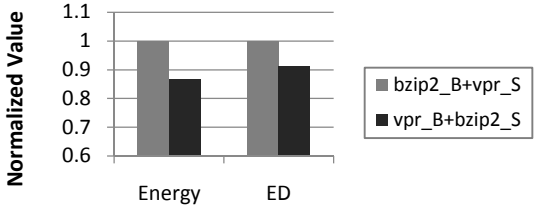


Figure 1. Energy and ED for *bzip2+vpr* with different mappings

While those proposed scheduling strategies are capable of improving performance, they do not necessarily lead to the most energy-efficient execution all the time, especially when the programs to be scheduled are scheduling-insensitive due to similar relative performance. We use the co-execution of programs *bzip2* and *vpr* on a dual-core heterogeneous CMP as an example to justify this argument. As listed in Table 1, these two programs demonstrate fairly similar performance ratio between big and small cores. Therefore, by employing an existing heterogeneity-aware scheduler based on relative performance, it is likely that 1) they are randomly mapped to different cores since the scheduler recognizes these workloads as scheduling-insensitive, or 2) *bzip2* is assigned to the big core as it demonstrates slightly higher performance gain than its co-runner. In Figure 1 we illustrate the energy consumption and energy-delay product (ED) for two possible scheduling, namely *bzip2_B+vpr_S* and *vpr_B+bzip2_S*, where the former one indicates that *bzip2* is running on the big core and *vpr* is on the small core. Similarly, the latter notation corresponds to the opposite scheduling decision. As can be observed, the second scheduling (i.e., *vpr_B+bzip2_S*) turns out to be more energy-efficient than its alternative due to the significant power reduction on the big core. Two implications can be noticed from this example. First, scheduling aiming to minimize the energy consumption and ED does not always reach consensus with the performance-oriented scheduling in a heterogeneous system. Second, for scheduling-insensitive workloads where programs have fairly close speedup on the big core, there is still plenty of headroom for the energy efficiency optimization.

Drawing upon these observations, we consider that an appropriate scheduling policy targeting on energy minimization is of great significance in emerging heterogeneous systems. Therefore in this work, we propose a rule-set guided schedul-

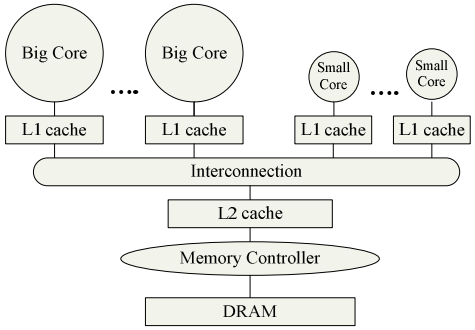


Figure 2. Architectural overview of the heterogeneous CMP

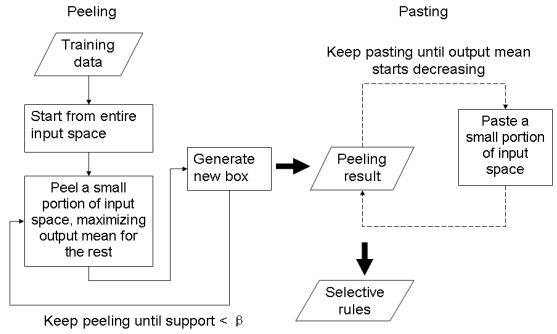


Figure 3. PRIM training procedure with peeling and pasting

ing strategy to minimize the energy consumption for workloads running on heterogeneous CMPs. Meanwhile, our scheduler is able to deliver comparable performance to the optimal existing heterogeneous scheduler, thus achieving higher energy efficiency than previous schemes. We employ an advanced statistical tool to facilitate the development of our algorithm. The tool is able to generate a set of "IF-ELSE" conditions with regard to common performance metrics on involved cores. Each condition is expressed as an inequality such as "$X_i \leq$ (or $\geq$) N", where X_i is an easily measured performance metric and N is a certain value. The scheduler then dynamically makes decisions for program assignment by comparing the runtime execution behaviors with the selected rules at each scheduling interval. When the conditions on both cores are satisfied, the scheduler predicts that a job swap will be more energy-saving than the current mapping, thus switching the programs on the big and small cores accordingly. Otherwise the current scheduling is maintained for the next period. In general, compared to prior schemes, our rule-set based scheduling strategy has the following advantages:

- It is capable of identifying the most energy-efficient execution patterns on heterogeneous CMPs. Instead of exclusively concentrating on performance gain, it pays attention to the energy saving and effectively increases the overall energy efficiency by scheduling intervals that might be inappropriately handled or ignored in throughput-targeted scheduling.
- It is a rigorous approach as the selective rules are generated by an advanced statistical tool. This means that the scheduling is guided by factors that are essentially influential to the energy efficiency rather than arbitrary variables.
- It is easy to be implemented in practice since no extra hardware is required. The performance counters available in most commercial processors can be utilized to monitor the execution behaviors for scheduling.

2. Heterogeneous Architecture

The heterogeneous platform considered in this study is a single-ISA CMP containing an equal number of big and small cores. Figure 2 illustrates its architectural overview. As can be seen, each core is equipped with private L1 caches, connecting to the shared L2 cache via an interconnection. The main memory stands as the lowest level in the memory hierarchy and communicates with the shared cache through a memory controller. Note that although our study is conducted on CMPs with shared last-level caches (LLC), the rule-set guided scheduling approach can also be adapted to systems without shared LLC and effectively increase the energy efficiency.

3. Statistical Tools

As described in section 1, the proposed scheduling scheme is built on the measurements of common performance metrics. This introduces two challenging problems to our study. First, we should identify the important factors which impose relatively large impact on the overall energy consumption. Second, we need to quantitatively formulate the scheduling condition with regard to the selective performance metrics. Taking these into consideration, we employ advanced statistical tools to facilitate the rule extractions.

3.1. Patient rule induction method (PRIM)

In this study, we employ the Patient Rule Induction Method (PRIM) [6] to bridge the gap between program execution behaviors and the scheduling condition. PRIM is naturally suitable to facilitate this study since its objective is to find a region in the input space that gives relatively high values for the output response. The selected region (or "box") is described in an interpretable form involving a set of "rules" depicted as $B = \bigcap_{j=1}^{p} (x_j \in s_j)$, where x_j represents the jth input variable and s_j is a subset of all possible values of the jth variable.

As shown by Figure 3, the construction of the selected region is composed of two phases: (1) patient successive top-down peeling process; (2) bottom-up recursive pasting process. The top-down peeling starts from the entire space (box B) that covers all the data. At each iteration, a small subbox b within the current box B is removed, which yields the largest output mean value in the result box B-b. We perform this operation iteratively and stop when the support of the current box B is below a chosen threshold β, which is actually the proportion of the intervals suitable for job swaps.

The pasting algorithm works inversely from the peeling results and the final box can sometimes be improved by readjusting its boundaries. Starting with the peeling solution, the current box B is iteratively enlarged by pasting onto it a small subbox that maximizes the output mean in the new (larger) box. The bottom-up pasting is iteratively applied, successively enlarging the current box, until the addition of the next subbox causes the output mean begin to decrease.

3.2. Classification and regression tree (CART)

Although the described statistical technique PRIM is able to build a rigorous correlation between multiple input variables and a response, the accuracy of the model depends on features of the applications in the training set. The PRIM algorithm is prone to build a model that fits the majority situations in the training instances. As a result, the established model might ignore samples appearing less frequently. Considering the diversity of program characteristics, this limitation might significantly decrease the prediction accuracy when the model is applied to different program phases or applications that demonstrate completely distinct execution behaviors to training samples.

In order to figure out this problem, we propose to partition the entire data set into several categories, each of which

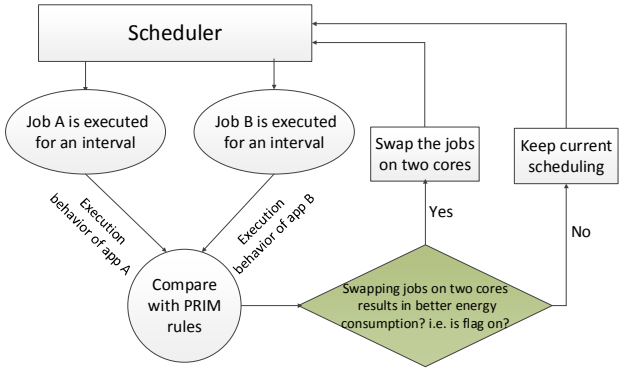


Figure 4. PRIM rules guided scheduling for dual-program execution

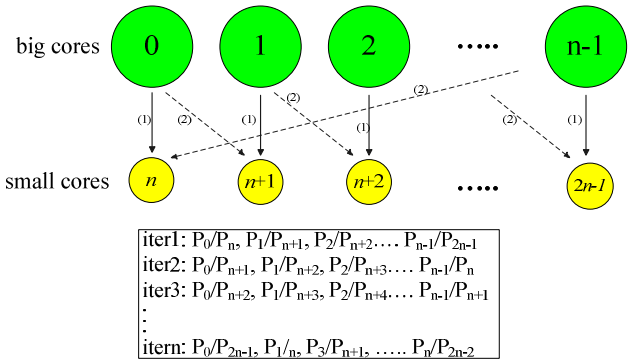


Figure 5. Pair-wise comparison illustration for 2*n*-program scheduling on an *n*B+*n*S platform

contains instances demonstrating similar characteristics. If we train a PRIM model for each data subset and generate a group of rule sets correspondingly, the obtained rules are supposed to be more robust and be effective to handle different execution scenarios. To achieve this goal, we employ another statistical tool named Classification and Regression Tree (CART) [3] for the data segmentation. CART has been in use for about 25 years and remains a popular data analysis tool. It provides an alternative to linear and additive models for regression problems. The CART models are fitted by a recursive partitioning whereby a dataset is successively split into increasingly homogenous subsets until the information gained by additional splits is not outweighed by the additional complexity due to the tree's growth. Trees are adept at capturing non-additive behavior, e.g. interactions among input variables are routinely and automatically handled.

4. Rule-set Guided Scheduling

In this section, we introduce our rule-set guided scheduling strategy in detail. We first sketch the workflow and training process of the algorithm, and demonstrate its scalability on heterogeneous systems with different configurations afterwards.

4.1. Algorithm description

Without loss of generality, we consider a scenario where two programs (A and B) run on a dual-core CMP consisting of one big core and one small core. For a scheduling interval, we need to compare the total energy consumption of the following two cases: (1) A on the big core and B on the small core; and (2) B on the big core and A on the small core. Between these two scheduling, we should choose the one with the lower energy consumption. Clearly, an oracle scheduler will examine these two cases during runtime (at each scheduling point) and swap the two programs if necessary to achieve the optimal energy efficiency. However, dynamically determining the optimal schedule for a workload at runtime is a challenging problem. To overcome this conundrum, we employ Patient Rule Induction Method (PRIM) to generate some selective rules on a number of performance measurements. In a scheduling interval, if the measured performance counters conform to these rules, the scheduler switches the two programs on the two types of cores.

More specifically, the PRIM model training is composed of the following steps. First, we randomly select a certain number of program pairs. For each program pair (A, B), we assume that A runs on the big core and B runs on the small core. For each interval (a certain time unit), we can obtain the following information by executing A and B on the big and small cores, respectively:

Program A: $< X_b^1, X_b^2, X_b^3, \ldots\ldots\ldots, X_b^m >$
Program B: $< X_s^1, X_s^2, X_s^3, \ldots\ldots\ldots, X_s^n >$

In the tuple of a program, the X variables denote the measured performance counters, such as the number of cache misses and the number of branch mispredictions. The subscript of each variable indicates the corresponding platform (i.e., b = big core, s = small core). In this example, we measure m performance counters on the big core and n counters on the small core. Second, we compare the energy consumption of this schedule with its counterpart (re-running B on the big core and A on the small core), setting a Boolean variable (flag) to one if swapping these two programs will consume lower energy. Note that in practice, the runtime energy can be either measured by using the model specific register (MSR) [1] equipped in most current processors or through dynamic estimation with a prediction model [8]. Consequently, we can form a PRIM training sample by combining the above information:

$< X_b^1, X_b^2, X_b^3, \ldots\ldots\ldots, X_b^m, X_s^1, X_s^2, X_s^3, \ldots\ldots\ldots, X_s^n,$ flag $>$

Finally, we feed the training samples measured from all selected program pairs into the PRIM model to extract the rules. The inputs are the m+n performance counters from both cores, and the output is the flag that indicates if these two programs need to be switched in the next interval. Recall from section 3.1 that PRIM rules identify the input space subregion that has the highest response values. Therefore, the above generated rules quantify the situations that a program switch between the big and small cores is needed (to achieve better energy efficiency).

The selective PRIM rules are then engaged by the operating system to guide the scheduling of the two programs A and B. Figure 4 illustrates how the rule set interacts with the operating system and makes decision for program assignment at runtime. The two programs are first executed on two cores (one big and one small) for an interval, respectively. At a scheduling point, the performance measurements of the current interval are compared with the extracted PRIM rules. If conditions on both cores are satisfied, the model predicts that swapping the two programs will lead to better energy efficiency; otherwise the present scheduling is preserved. The scheduler then makes the assignment based on the prediction result and continues the execution to the next scheduling point. Note that the rule-set guided scheduling is sufficiently flexible to manage the program execution for optimizing different metrics. For instance, by changing the objective during the model construction, this approach can be easily applied to guide the scheduling in a system where performance maximization is the prime concern. Nevertheless, our concentration in this paper is energy minimization.

As described in section 3.2, the effectiveness of the rule guided scheduling is largely determined by the features of the programs in the training set. In case where the programs for validation demonstrate significantly different execution be-

haviors from the training programs, the derived rules may not be effective in identifying the swapping cases. In this situation, the model accuracy can be further improved by preprocessing the training data. Instead of training a single PRIM model, we can build a number of different PRIM models according to the similarity of different training samples. Specifically, we use the CART mechanism to partition the input space into a few subregions. The instances belonging to each individual subregion are similar in terms of energy efficiency. After that, we build a separate PRIM model for each of these subregions. Consequently, we will have a group of rule sets. When making predictions during runtime, we first identify which subregion the current input sample locates in, then use the corresponding rule set to determine if a program switch is needed. In practice, the number of subregions doesn't need to be large. Our experiments show that partitioning the input space into 4 subregions (and also training 4 PRIM models accordingly) can result in prediction accuracy within only 5% difference from the oracle scheduler. This approach is termed Hierarchical PRIM (or H-PRIM).

4.2. Algorithm scalability

Our approach is sufficiently scalable to be adopted by a system with more than two cores. We assume a CMP with an equivalent number of big and small cores while the core count of each processor type is n. In this scenario, the optimal energy efficiency can be achieved by performing n iterations of parallel pair comparison. The scheduling process is illustrated in Figure 5. As shown in the figure, in the first iteration, a big core with the index i ($i \in [0, n-1]$) is compared with the small core whose index is ($n + i\%n$). All n pairs of comparisons are performed in parallel. In the second iteration, the big core i will form a pair with the small core ($n+(1+i)\%n$) and make comparison correspondingly. Similarly, the comparison will be conducted between the big core i and the small core ($n+(n-1+i)\%n$) in the nth iteration. Note that the mod operations are involved to emulate the rotational comparisons. We prove that this method will lead to the optimal scheduling as follows.

Since we have n big cores and n small cores, as well as $2n$ jobs running on them, the optimal schedule is a situation that n jobs suitable running on the big cores for low energy consumption (we label the jobs as "1"s) will be assigned to n big cores and the left n jobs, denoted as "0"s, will be allocated on n small cores. We claim that all "1" programs will be assigned to big cores and all "0" jobs will be allocated on small cores after n iterations, even though we are unaware of the program classification at the beginning, i.e., whether a program belongs to "1" category or "0" category. During each of the n iterations, we have n parallel comparisons between big and small cores. For each comparison, we seek better energy efficiency for two programs running on a big-small core pair. Therefore, we have four possible situations before the comparison:

(1) a "1" job running on a big core compared with a "0" job running on a small core;

(2) a "0" job running on a big core compared with a "1" job running on a small core;

(3) a "1" job running on a big core compared with another "1" job running on a small core;

(4) a "0" job running on a big core compared with another "0" job running on a small core.

For the first two cases, it will generate an ideal situation that a "1" job will be assigned on a big core. For the third case, a "1" job will also be allocated on a big core, no matter which "1" job is selected. Similarly, a "0" job will be set on a big core for the fourth case. However, there must be a "1" job running on a small core at this point, considering that the number of "1" jobs is equal to the total number of big cores. This implies an opportunity for this "1" job running on a small core to be compared with a "0" job executed on a big core in a future iteration, since we have n iterations of parallel comparisons. Thus, any case (4) comparison will fall into case (2) comparison eventually. Based on this analysis, we conclude that all "1" jobs will finally go to big cores, meaning that the optimal schedule is achieved after n iterations.

Table 2. Architectural parameters of system components

Component	Parameter	Value
Big core	Pipeline type	out-of-order
	Processor width	4
	ALU/FPU	4/2
	ROB/RF	120/160
	L1I cache size	32KB
	L1D cache size	32KB
	L1 associativity	4
	BTB entries	2048
Small core	Pipeline type	in-order
	Processor width	2
	ALU/FPU	2/1
	L1I cache size	16KB
	L1D cache size	16KB
	L1 associativity	2
	BTB entries	1024
Other parameters	L2 cache size	4MB
	L2 associativity	8
	Cache block size	32B
	Branch Predictor	Hybrid
	Frequency	3G

5. Experimental Setup

5.1. Simulation environment

We use a modified SESC simulator [21] to evaluate the effectiveness of the proposed algorithm. The simulator is configured to contain a number of big and small cores, whose architectural parameters are listed in Table 2. McPAT v0.8 [14] is used for dynamic and leakage power estimation. We combine programs from SPEC CPU2000 and SPEC CPU2006 with *ref* input as the workloads and randomly select 220 program pairs for simulation. Each program of a workload is simulated 1 billion instructions after fast-forwarding the initial 2 billion. Within the 220 workloads, we choose 180 program combinations for PRIM model training and use the remaining ones to evaluate the effectiveness. Recall that the training procedure is conducted offline, which takes about 3 seconds on a Dell Precision T7500 workstation equipped with an Intel E5530 CPU. In addition, we always run as many cores as programs for scheduling.

The scheduling interval is set to 2.5ms in this study. As shown in prior works [5], this granularity is small enough to capture the variations in program execution behaviors and assist the scheduler to make energy-efficient assignments more precisely. We do account the migration overhead due to architectural state retrieving and set it to 150μs [13]. The additional energy dissipation due to the migration is also appropriately modeled. For instance, the energy consumed by cache re-warming can be calculated from the corresponding cache access times. We compare performance, total energy consumption, and ED product resulted from different schedulers to assess the effectiveness.

5.2. Scheduling algorithms for comparison

In this subsection, we introduce the scheduling strategies that are implemented for comparison.

Static scheduling: This is the baseline scheduler implemented for the comparison. The programs are randomly assigned to processor cores and execute till completion.

Round-robin (R-R): The programs running on the big and small cores are swapped every 10 intervals. The scheduler blindly swaps the jobs at a preset frequency without considering the runtime execution behaviors.

Sample-Optimize-Symbios (SOS): With this scheduling policy, the execution proceeds in a pattern consisting of three steps. In the "sampling phase", the programs are executed on each type of core for an interval, so the energy consumption of each assignment is available after this process. In the "optimization phase", the most energy-efficient scheduling is identified. Finally in the "symbios phase", all programs are running *N* intervals with the optimal mapping. In this study, *N* is set to 10. Note that this strategy is also called "sampling" in a few prior works [2].

MLP-ratio: This scheme is introduced in a recent work [5] aiming to improve the system throughput on heterogeneous CMPs. Although it does not focus on energy saving, it stands as one of the best heterogeneity-aware schedulers to date, thus deserving a comparison with our strategy. Note that the optimal scheduler proposed in [5] takes both the instruction-level parallelism (ILP) and the memory-level parallelism (MLP) into consideration. Nevertheless, the authors demonstrate that the algorithm based on only MLP-ratio delivers fairly close performance to their optimal scheduler. Considering the complexity to calculate the ILP on the fly, we implement a scheduling scheme based on only MLP estimation due to its simplicity. With this scheduler, the memory-level parallelism (MLP) ratios of all programs between the big and small cores are evaluated. Programs with higher MLP ratios are placed on the big cores while those with lower ratios are assigned to small cores.

PRIM: At a scheduling point, the performance metrics collected from a pair of big and small cores are compared with the PRIM rules. If the conditions for both core types are satisfied, the jobs on two cores are swapped; otherwise the current assignment is maintained. When the number of cores is greater than or equal to four, the optimal scheduling is achieved in a few steps as described in section 4.2.

Hierarchical PRIM (H-PRIM): We use CART to partition the training data into 4 categories according to the performance measurements and train a PRIM model for each subset. At a scheduling point, we first identify to which subset a pair of program executions belongs. We then compare the corresponding PRIM rules with the execution behaviors of these two programs and make scheduling accordingly.

Oracle: In this scheduling policy, we assume that the scheduler knows the energy consumption of each program mapping in apriori and performs the optimal scheduling based on that information. To implement this algorithm, we measure the total energy of each program assignment for each scheduling interval.

6. Results

In this section, we perform a detailed evaluation of the rule-set guided scheduling algorithm by comparing it with a set of existing schemes. We start by giving a brief explanation to the extracted rules.

6.1. Selective rules for PRIM

By training a PRIM model, we can generate the following rule sets to guide the scheduling for two programs running on a pair of big and small cores:

Rule set:

Big Core Rules:

L1D.nMiss<13430 && nStall.SmallIQ>256949 && nFetch>4879317

Small Core Rules:

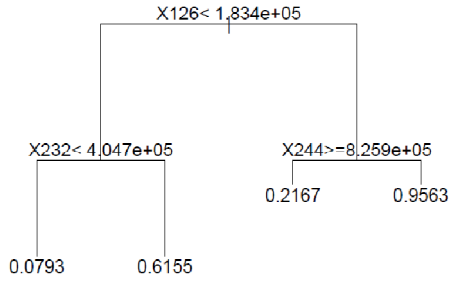


Figure 6. Data segmentation result from CART

BP.nMiss>9674 && iLoad.count>198169 && iALU.count < 400026 && L1D.nMiss < 17701

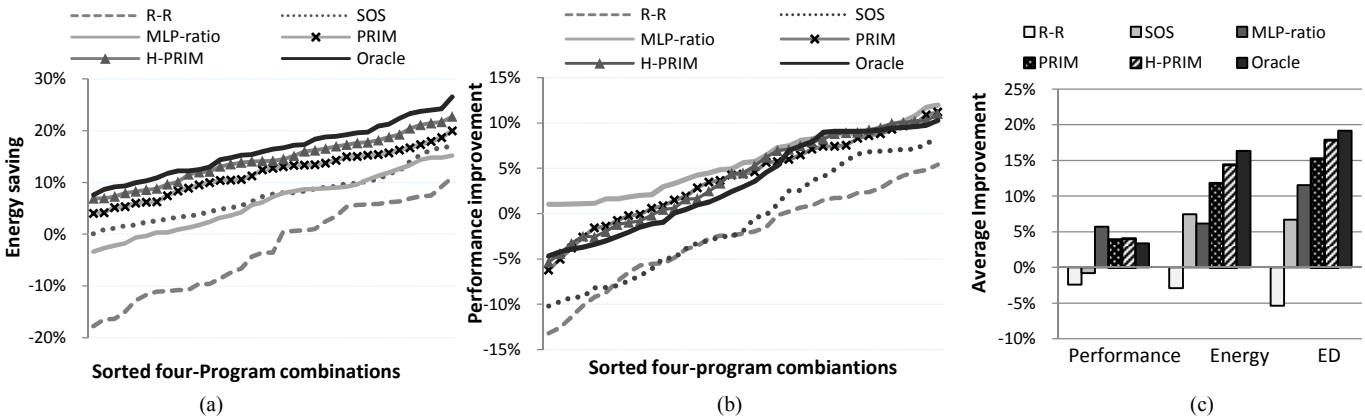
At a scheduling point, the scheduler compares the performance metrics collected from a pair of big and small cores. If the measurements on both sides are satisfied with these rules, the scheduler predicts that swapping the two programs will decrease the total energy consumption.

Since our prediction is made at each scheduling interval, minimizing the energy consumption essentially translates to lowering down the total power of that period. Also, the big cores always consume much larger power than the small ones, thus dominating the total power consumption all the time. We present these two statements to assist the interpretation of the PRIM rules. Note that we will analyze the correlation between execution behaviors and power consumption including both dynamic power and leakage power. Let us focus on the big core rules at first. As can be seen, the big core rules suggest that a program with a low L1 cache miss rate (*L1D.nMiss*), a high instruction fetch rate (*nFetch*), and substantial internal stalls (e.g., stalls due to small instruction issue queue, or *nStall.smallIQ*) should be exchanged to the small core for execution. It is straightforward to understand the metrics related to L1 cache and instruction fetch. A Low L1 cache miss rate and high fetch speed indicates that the current program on the big core is executed at relatively high speed without suffering from frequent cache misses. However, from the power perspective, this implies large dynamic power on many function units due to high activities. As for the second condition, a large *nStall.smallIQ* value implies persistent high utilization on the issue queue. This also increases dynamic power consumption on this component because of frequent operations such as checking the operands status. On the other hand, components including IQ and integer ALU tend to become the hotspot on die. As a consequence, the leakage power on these units is rapidly increasing since it is positively related to the temperature. In one word, the big core rules outline the features of intervals which tend to consume both high dynamic and leakage power. For the purpose of energy saving, these intervals should be migrated to the small core for execution.

We now shift our concentration to the small core rules. Recall that the total power consumption is dominated by the big core. Therefore, the rules for the small core essentially characterize the execution phases that are not likely to result in extreme high power on a big core. Specifically, the first and the third conditions respectively set constraints for the occurrences of branch mispredictions (*BP.nMiss*) and number of integer ALU instructions (*iALU.count*). A branch misprediction will lead to a pipeline flush and lower down the execution speed. Fewer number of ALU instructions can alleviate the utilization on ALUs, reducing the dynamic power and cooling down the component accordingly. These two conditions jointly reduce the power consumed by the core running this program. On the other hand, the rule sets require that the number of load instructions (*iLoad.count*) should be

Table 3. PRIM rules for each data subset

Segment	Big Core Rules	Small Core Rules	Performance metrics description
Subset 1	*LDSTUnit.util* < 0.27 && *nStall.noCachePort* < *10785*	*L1I.nMiss* > *3791*	*LDSTUnit.util*: the utilization of load/store unit *nStall.noCachePort*: cumulative stall cycles due to cache port contention *L1I.nMiss*: number of misses in L1I cache
Subset 2	*IQ.avgFree* < 3 && *L1I.nHit* > 817392	*avgBranchPenalty* > *73* && *iComplex.count* < *3921*	*IQ.avgFree*: the average number of free entries in IQ, indicating the IQ utilization *L1I.nHit*: number of hits in L1I cache *avgBranchPenalty*: average penalty (in cycles) of branch misprediction *iComplex.count*: number of integer complex instructions, such as division
Subset 3	*nStall.noCachePort* < *44750* && *nStall.smallREG* > *851493*	*L1I.avgMissLat* < *513*	*nStall.smallREG*: cumulative stall cycles due to available registers *L1I.avgMissLat*: average penalty (in cycles) of a miss in L1I cache
Subset 4	N/A		



(a) (b) (c)

Figure 7. Evaluation results of four-program workloads running on a 2B+2S platform: (a) energy saving, (b) performance improvement, (c) average improvement for energy, performance and ED

no smaller than a certain value while the misses in L1 data cache (*L1D.nMiss*) cannot go beyond a threshold. These two conditions imply that this program potentially issues a large amount of memory requests, but most of them can be served by the L1 cache. Nevertheless, the stress on L1 cache will not significantly increase the total power consumption since the L1 cache consumes relatively small power compared to other components. In general, the intervals filtered by the small core rule set tend to result in moderate power if executed on the big core, thus reducing the total power consumption.

6.2. Hierarchical PRIM

As described in section 3.2, CART is able to partition the entire data set into several subsets, each of which contains similar samples. Therefore, if we train a PRIM model for each subset, the effectiveness of our strategy is expected to increase due to the similarity of instances within the same subset. Taking this into account, we use CART to perform a data segmentation operation in prior to the PRIM model training. Figure 6 demonstrates the segmentation result for all training instances. As can be seen, the entire data set are partitioned into 4 categories, as represented by the 4 leaf nodes on the generated tree. Each branch represents a condition on the performance metrics on a big or small core and is expressed in a form of "$X_i \geq$ (or $\leq$) M", where X_i denotes a performance metric and M denotes a value to segment the data set. Specifically in the tree shown in Figure 6, *X126* indicates the number of branch mispredictions on the big core (*BP.nMiss*) and *X232* corresponds to the *iALU.count* metric which records the number of ALU instructions on the small core. *X244* tracks the number of fetched instruction on the small core (*nFetch*). The value at each leaf node is the average of CART response for that partition.

We train a PRIM model for each data segment and list their respective rule sets in Table 3. Note that for the fourth subset (i.e., the rightmost leaf node in Figure 6), more than 95% of the included samples maintain a flag "1", meaning that the majority of this partition are candidates for job swap. Consequently, we do not train an extra PRIM model for this subset and directly use its branch conditions to guide the scheduling. During the execution, this tree is accessed at each scheduling point in order to classify a program pair into an appropriate subset. The access starts from the root node and eventually ends at one of the four leaf nodes according to the execution behaviors. The corresponding PRIM rules are then used to assist job assignments.

6.3. Effectiveness comparison

Figure 7(a) illustrates the energy reduction for these workloads on the first platform (2B+2S) when distinct schedulers are engaged. Note that all results are compared with those corresponding to the static scheduling case. The workloads are sorted in ascending order according to the degree of energy saving. As can be observed, our PRIM and H-PRIM strategies always outperform other scheduling algorithms with respect to energy consumption. This is because the ruleset guided scheduler is capable of effectively identifying the most appropriate program assignment at runtime to minimize the energy consumption. Furthermore, the total energy consumed by H-PRIM is fairly close to the oracle case, since the data segmentation increase the accuracy of identifying the candidate intervals. Other schedulers suffer from distinctive drawbacks which adversely impact their effectiveness. For example, the SOS scheduler assumes a continuum of program characteristics in the symbios stage (i.e., the following N intervals after sampling), which might not be true as the execution behaviors usually vary across different phases. This causes inefficient executions in many intervals and thus raise the energy consumption. With the round-robin scheduler, the programs may be coincidentally scheduled in a correct way during some intervals; however, they also tend to run with fairly low energy-efficiency in other periods. There-

fore, we do not observe impressive energy saving with this strategy. The MLP-ratio algorithm, on the other hand, aims to improve the system performance. As we demonstrated in section 1, this scheduler can increase the total energy consumption for some intervals, thus trailing our strategies in saving the total energy.

Figure 7(b) shows the performance improvement delivered by these schedulers. Note that the workloads are sorted according to the performance gain in this figure. Also recall that the oracle scheduler is the optimal with respect to the energy consumption instead of performance. As can be seen, the MLP-ratio strategy always leads to better performance (i.e., positive value) compared to the baseline. This does not go beyond our expectation because the goal of this scheduler is to enhance the overall performance, thus the programs are assigned in a manner to maximize the execution speed. On the other hand, both performance improvement and degradation (i.e., negative value) are observed in other scheduling policies. The performance loss mainly stem from two sources, namely migration overhead and slower execution in certain intervals. Our scheduler eliminates unnecessary job swaps during the execution compared to round-robin and SOS, thus delivering better performance.

Figure 7(c) demonstrates the average performance gain, energy saving and ED reduction for all workloads. From the performance respective, MLP-ratio stands as the optimal by accelerating the execution by 5.7% while PRIM and H-PRIM respectively enhance the performance by 3.9% and 4.1%. Note that the MLP-ratio scheduler is more effective for scheduling-sensitive workloads [6]. However, the performance gains for applications demonstrating less sensitivity to program assignment are fairly modest. Therefore in general, our schedulers lead to comparable performance to MLP-ratio on average. For energy saving, the PRIM and H-PRIM algorithms are able to reduce the energy consumption by 11.8% and 14.8% compared to 16.3% delivered by the oracle scheduler. Finally, for the ED metric, the PRIM and H-PRIM algorithms respectively reduce its value by 15.3% and 17.9% while the oracle scheduler can decrease the product by 19.1%. The SOS and MLP-ratio policies lead to less impressive savings. In other words, our best scheduler H-PRIM outperforms the MLP-ratio policy, which is one of the optimal state-of-the-art heterogeneous schedulers by 7.8% and 5.7%, respectively on energy and ED.

7. Related Work

Within past years, several researchers have authored outstanding studies in the heterogeneous architecture field. Kumar et al. [10] propose one of the earliest single-ISA heterogeneous multiprocessor and discuss the sampling-based scheduling algorithm. In [11], the performance for multithreaded workload executing on a single-ISA heterogeneous processor is analyzed in detail. Becchi and Crowley evaluate a set of heterogeneity-aware scheduling policies in [2] and show that dynamic job scheduling largely outperforms the static assignment by delivering higher throughput. In [9], Koufaty et al. introduce the bios scheduling which is similar to the policies based on memory intensity. Saez et al. present a series of works that target the performance enhancement on asymmetric CMP platforms [18][19][20]. Radojkovic et al. [17] consider a scenario with massive multithreaded processors where an exhaustive search for the optimal task assignment is unfeasible. Therefore, they introduce a statistical approach to seek the best work distribution. Li et al. [15] implement a scheduler composed of fast-core-first assignment and migration on a performance-asymmetric CMP architecture. Lakshminarayna, on the other hand, propose an age based approach that maps the thread with longer remaining execution time to a faster core [12]. More recently, Craeynest et al. present a heterogeneous scheduler via performance impact estimation (PIE) [5].

There are few studies addressing energy minimization on heterogeneous platforms. In [4], Chen and John present a scheduler based on weighted Euclidean distances to improve the energy efficiency on heterogeneous CMPs. Grant et al. [7] introduce a scheduling mechanism to save energy on asymmetric multiprocessors for scientific applications. In their proposed algorithm, one core is reserved for running the operating system at adjustable frequencies while other processors are executing the user threads at full speed.

8. Conclusion

In this paper, we propose a scheduling strategy for energy-efficient execution on single-ISA heterogeneous chip-multiprocessors. We demonstrate that performance-oriented scheduling may lead to executions that are not sufficiently energy-efficient. Due to this limitation, we concentrate on energy saving and introduce a rule-set guided scheduling to exploit the optimal energy efficiency on heterogeneous CMPs. The evaluation results show that our proposed algorithm impressively outperforms existing schedulers and leads to better energy efficiency.

9. References

[1] Intel 64 and IA-32 architectures software developers manual volume 3: system programming guide. Oct. 2011.

[2] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In CF, May 2006.

[3] Breiman, L., Friedman, J.H., Olshen, R.A., and Stone, C.J. Classification and Regression Trees, Wadsworth Press, 1984.

[4] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In DAC, Jun. 2009.

[5] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation. In ISCA, Jun. 2012.

[6] J. Friedman and N. Fisher. Bump hunting in high-dimensional data. In Statistics and Computing, 9, 1999.

[7] R. Grant, and A. Afsahi. Power-performance Efficiency of Asymmetric Multiprocessors for Multi-threaded Scientific Applications. In the 2nd workshop on High-Performance, Power-Aware Computing, with IPDPS, Apr. 2006.

[8] S. Kaxiras, and M. Martonosi. Computer Architecture Techniques for Power Efficiency. Morgan and Claypool Publishers.

[9] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In EuroSys, Apr. 2010.

[10] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, D.M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In MICRO. Dec. 2003.

[11] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In ISCA. Jun. 2004.

[12] N. B. Lakshminarayna, J. Lee, H. Kim. Age based scheduling for asymmetric multiprocessors. In SC, Nov. 2009.

[13] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In ExpCS, Jun. 2007.

[14] S. Li et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In MICRO'09.

[15] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In SC. Nov. 2007.

[16] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating System Support for Overlapping-ISA Heterogeneous Multi-core Architectures. In HPCA. Jan. 2010.

[17] P. Radojkovic et al. Optimal Task Assignment in multithreaded processors: a statistical approach. In ASPLOS, Mar. 2012.

[18] J. C. Saez, A. Fedorova, M. Prieto, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In EuroSys, Apr. 2010.

[19] J. C. Saez, A. Fedorova, D. Koufaty, and M. Prieto. Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. In ACM TOCS, Apr. 2012.

[20] J. C. Saez, D. Shelepov, A. Fedorova, and M. Prieto. Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. In JPDC, Aug. 2010.

[21] J. Renau et al. SESC Simulator.