Predictive Coordination of Multiple On-Chip Resources for Chip Multiprocessors

Jian Chen and Lizy K. John
Department of Electrical and Computer Engineering
The University of Texas at Austin, Austin, Texas, USA
chenjian@mail.utexas.edu, ljohn@ece.utexas.edu

ABSTRACT

Efficient on-chip resource management is crucial for Chip Multiprocessors (CMP) to achieve high resource utilization and enforce system-level performance objectives. Existing multiple resource management schemes either focus on intra-core resources or inter-core resources, missing the opportunity for exploiting the interaction between these two level resources. Moreover, these resource management schemes either rely on trial runs or complex on-line machine learning model to search for the appropriate resource allocation, which makes resource management inefficient and expensive. To address these limitations, this paper presents a predictive yet cost effective mechanism for multiple resource management in CMP. It uses a set of hardware-efficient online profilers and an analytical performance model to predict the application's performance with different intra-core and/or inter-core resource allocations. Based on the predicted performance, the resource allocator identifies and enforces near optimum resource partitions for each epoch without any trial runs. The experimental results show that the proposed predictive resource management framework could improve the weighted speedup of the CMP system by an average of 11.6% compared with the equal partition scheme, and 9.3% compared with existing reactive resource management scheme.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Performance

Keywords

Microprocessor, Resource management, Program characteristics, Performance modeling

1. INTRODUCTION

Chip Multiprocessors (CMP) have become mainstream platforms to improve the system throughput for multi-threaded and multi-programmed workloads in high-performance computing. However, their energy efficiency and end-performance is strongly dependent on management of the ever-increasing on-chip resources. It is well known that unrestricted sharing of inter-core resources such as L2 cache and memory bandwidth, can lead to destructive interference between the running threads [20], resulting in large performance variation and throughput degradation. Yet, managing inter-core resources alone is not sufficient as modern CMPs, such as Intel Nehalem processor [6], support per-core Simultaneous Multi-threading (SMT). Under such circumstance, the resource sharing in a CMP is compounded with both inter-core and intra-core resources, and any resource management scheme without coordinating between these two types of resources could lead to suboptimal system performance and inability to enforce system performance objectives.

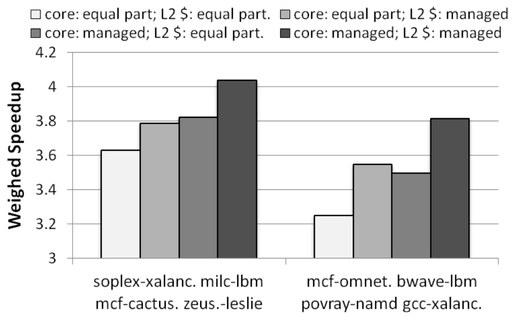


Figure 1: Performance comparison for different resource management policies. Results are based on a quad-core CMP with per-core 2-way SMT (Detailed configurations in table 3).

As an example, Figure 1 shows the comparison of the weighted speedups for different combination of inter-core and intra-core resource management schemes in a quad-core 2-way SMT CMP system. Inter-core resource here is represented by L2 cache, and intra-core resources include issue queue (IQ), reorder buffer (ROB), and physical registers, all partitioned in proportion to each other [9]. As we can see, although separate management of L2 cache or intra-core resources improves the performance over the scheme of equal partition, it still misses a large amount of potential for improving system performance compared with the one that coordinates the allocation of L2 cache and intra-core resources. This is because the application's demands on different resources are correlated, and the change of the application's intra-core resource allocation could affect the its demands on inter-core resources. For example, the increase of ROB size may expose more memory level parallelism (MLP), and consequently increase the number of outstanding load misses. Since multiple outstanding load misses could hide the latency with each other, the average cache miss penalty is reduced, hence the requirement of L2 cache size is smaller in order to maintain the same performance. Therefore, coordinating between intra-core and inter-core resources is necessary to achieve high utilization and system performance in the CMP+SMT environment.

However, existing management schemes for multiple interacting resources focus on either intra-core resource partitioning for a single-core SMT processor or inter-core resource allocation for a chip multiprocessor. Cazorla et al. [8] proposed a resource sharing model to estimate the anticipated resource needs of a thread, and dynamically allocate shared resources to the thread that utilizes the resource most efficiently. Yet, this method only *indirectly* improves the performance and is unable to control the end performance. Choi and Yeung [9] improve the SMT resource partition by using the *direct* performance feedback to learn the desired resource allocation via *hill-climbing*. However, this method requires tentative runs to explore a large amount of trial resource partitions, fundamentally limiting its potential for performance improvement. Moreover, these methods only address intra-core resource allocation, and are not suitable for CMP inter-core resource management. To manage multiple inter-core resources, Bitirgen et al. [3] proposed an on-line machine learning model to capture the performance impact of multiple interacting resources. However, their model requires extensive training/re-training before it can accurately predict the application's performance, and incurs significant cost in hardware implementation and validation. Moreover, the original proposal of their model only addresses inter-core resource management, hence yields only suboptimal performance in the CMP+SMT scenario. Finally, while these existing policies recognize the importance of providing Quality-of-Service (QoS) on performance [13] for the co-executing applications, none of them provide a comprehensive solution to enforce performance objectives on a CMP platform where both inter-core and intra-core resources can vary simultaneously.

To address these limitations, this paper presents a comprehensive yet cost-effective resource management framework that can coordinate both intra-core and inter-core shared resources meanwhile simultaneously enforce QoS performance objectives. Unlike the existing resource management schemes, the proposed framework leverages an analytical performance model to predict the performance, and enforces resource allocations without any trial resource partitioning or training. By using the application characteristics dynamically collected during the application's execution, the performance model can update the performance prediction at each resource adaptation epoch, allowing the resource allocation to dynamically adapt to program phase changes. In particular, the contributions of this paper are as follows:

- We build a comprehensive yet cost-effective dynamic on-line profiler, and a performance model that utilizes the online profile to accurately predict the performance of the applications under different allocations of both inter-core and intra-core resources. We show that with about 22kilobytes of hardware, the performance model could predict the performance with an average relative error of 8.1%.

- We propose a framework for multiple resource management based on this performance model. This framework eliminates the need of trial-runs or training for dynamic resource allocation, and allows the enforcement of QoS performance objectives. We compare our approach with a set of resource management schemes from prior work, and show that on average, our approach improves the weighted speedup by 11.6% over the equal partition management scheme, and 9.3% over the reactive *hill-climbing* method [9].

The organization of this paper is as follows. Section 2 gives the overview of the proposed resource management framework. Section 3 describes the performance model. Section 4 shows the structures of the online profilers. Section 5 presents the resource partitioning algorithm. Section 6 analyzes the hardware cost of the implementation. Section 7 describes the experiment methodology. Section 8 discusses the results. Section 9 describes the related works, and section 10 concludes of this paper.

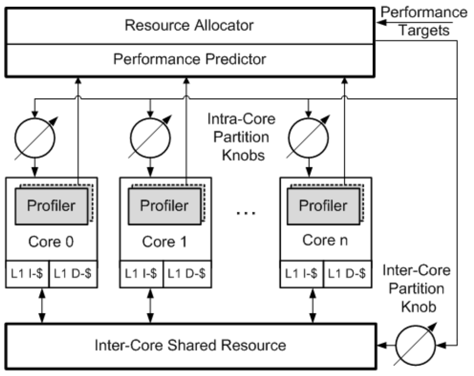


Figure 2: The overview of the predictive resource management framework.

2. OVERVIEW OF THE FRAMEWORK

The proposed framework for multiple resource management consists of three major components: the on-line profilers, the performance predictor, and the resource allocator, as shown in Figure 2. The on-line profiler non-invasively profiles each thread running on each core, and extracts the inherent characteristics of the thread for performance prediction. The performance predictor collects the profiled characteristics of the thread at the end of each resource allocation epoch, and estimates the thread's performance for different resource allocations. The resource allocator uses a built-in search engine to identify the appropriate resource allocations under the constraint of the given performance targets, and enforces the resource partition for each thread through a set of partition knobs.

The intra-core partition knobs regulate the allocation of the intra-core resources, which include IQ, ROB, and physical registers. These resources are interdependent, and are allocated *in proportion to* each other, similar with the way employed in the work by Choi et al. [9]. On the other hand, the inter-core partition knobs control the distribution of Last Level Cache (LLC) size and the power consumption of each core. In this paper, we assume that the CMP uses L2 cache as LLC and supports per-core Dynamic Voltage and Frequency Scaling (DVFS). In DVFS, the voltage and frequency are correlated, hence the power management can be achieved by controlling the operating frequency of each core meanwhile keeping the total power within the budget. This framework does not explicitly manage the memory bandwidth. Instead, it uses PAR-BS memory scheduling policy [19] to ensure the fairness and QoS of bandwidth usage.

While this framework addresses the resource allocation issues in the CMP+SMT scenario, it could also be applied in the cases where each core only supports single thread but can be dynamically reconfigured. Nevertheless, this paper focuses on the CMP platform with each core supporting 2-way SMT to demonstrate the effectiveness of the framework. In the following sections, we explain each component of the proposed framework in detail.

3. PERFORMANCE PREDICTOR

Predicting the performance impact of different resource allocations is the key step to avoid expensive trial runs and enable fast identification of appropriate resource distributions. Unlike the machine learning model proposed by Bitirgen et al. [3], our performance predictor is based on an analytical model, which does not require training/retraining and is easy to implement and validate.

3.1 Basic Performance Model

The performance model is based on the previously proposed interval analysis [15][11], which treats the exhibited Instruction-Per-Cycle (IPC) rate as a sustained ideal execution rate intermittently disrupted by long time miss events, such as, L2 cache misses and branch misprediction, etc. With the interval analysis, the total Cycle-Per-Instruction (CPI) of an application can be treated as the sum of three CPI components:

$$CPI_{total} = CPI_{exe} + CPI_{mem} + CPI_{other} \qquad (1)$$

CPI_{exe} represents the steady-state execution rate when the execution is free from any miss events. It is fundamentally constrained by the inherent Instruction Level Parallelism (ILP) of the application and the issue width of the processor. The ILP of the application is typically characterized by the critical dependency chain of the instructions in the instruction window. Assume an instruction window size w, and average critical dependency chain length l_w. On an idealized machine with unit execution latency, l_w indicates the average number of cycles required to execute the instructions in the instruction window, hence the average throughput is w/l_w. For a more realistic machine with non-unit execution latency, this number should be further divided by the average execution latency lat_{avg} according to Little's law [15]. Therefore, the average ILP, α_{avg}, can be obtained by $w/(lat_{avg} \cdot l_w)$, which also represents the steady-state execution rate if the instruction issue width is unlimited. However, for a realistic processor with limited issue width β, the ideal execution rate would be saturated at either the average ILP or the issue width, whichever is smaller. As a result, CPI_{exe} can be obtained by $1/min(\alpha_{avg}, \beta)$.

CPI_{mem} represents the penalty caused by the load misses in the last level cache (L2 cache in this paper). It can be calculated by the multiplication between the number of L2 load misses N_{L2}, and the average memory access latency lat_{mem}, assuming there are no multiple L2 cache misses outstanding. In practice, in order to hide the load miss latency, L2 caches are usually non-blocking and multiple L2 cache load misses could be outstanding. Under this circumstance, it has been proven that the average load miss latency is reduced to lat_{mem}/m_{ovp} [15], where m_{ovp} is the average number of outstanding load misses. Therefore, CPI_{mem} can be calculated by $lat_{mem} \cdot N_{L2}/(m_{ovp} \cdot N_{inst})$, where N_{inst} is the total number of retired instructions. Note that the term N_{L2}/m_{ovp} could also be treated as the number of L2 load misses that are not overlapping with each other, and hence is referred to as the *non-overlapped L2 load misses* N_{novp}.

CPI_{other} is the CPI component caused by other miss events, such as instruction cache misses, branch mispredictions, etc. In this paper, we do not change the resources related with these miss events. Therefore, this CPI component is approximately constant for an application with different resource allocations, as long as the application is in a stable execution phase. This CPI component can be obtained by transforming equation (1) to $CPI_{other} = CPI_{total} - CPI_{exe} - CPI_{mem}$, where CPI_{total} can be obtained from the performance counter, CPI_{ideal} and CPI_{mem} can be derived from the observed program characteristics. Once CPI_{other} has been deduced, it can be plugged into the performance model to estimate the performance of other cores. As a result, we have our basic performance model as follows:

$$CPI_{total} = \frac{1}{min(\alpha_{avg}, \beta)} + \frac{lat_{mem} \cdot N_{novp}}{N_{inst}} + CPI_{other}$$

With the basic performance model, the performance impact of different clock frequencies can be captured by converting the CPI to the delay in terms of absolute execution time. Hence, we have:

$$Delay = \frac{N_{inst}}{min(\alpha_{avg}, \beta) \cdot f} + t_{mem} \cdot N_{novp} + C_{other}/f \qquad (2)$$

where f is the operating frequency, t_{mem} represents the absolute memory access latency, and C_{other} refers to $CPI_{other} \cdot N_{inst}$, representing the cycles spent on other miss events.

3.2 Interaction of Co-executing Threads

The basic performance model only captures the performance of a thread when it is executed alone on a core and is free to access all available intra-core resources. However, when multiple threads simultaneously execute on a core, these threads will compete each other for the shared intra-core resources, causing interference on the performance of each co-executing thread. In practice, to achieve controllable performance for each thread, the shared intra-core resources are dynamically partitioned among the threads [9] except for the issue/dispatch width, which often remains as shared such that one thread can exploit the full execution bandwidth when the other thread is waiting for its miss events to be served [10]. In such case, the effective issue width of each thread may be significantly different from the physical issue width, and the basic performance model needs to be augmented accordingly.

Assuming a processor with 2-way SMT and per-thread retirement capability, the effective execution rate of the thread can be estimated by analyzing the ILP of the co-executing threads. For example, if the ILP of thread T_0 (referred to as α_{T0}) and the ILP of thread T_1 (referred to as α_{T1}) are both larger than the issue width β of the processor core, on average each thread can execute at a rate equal to half of the issue width. If we could further obtain the fraction of the time that T_0 is in long latency miss event, the effective execution rate of T_1 can be derived by considering the additional execution bandwidth T_1 has during that fraction of time. Similarly, if α_{T0} and α_{T1} are both smaller than β but the sum of these two is larger than β, on average the effective issue width of a thread is determined by the occupancy of its ready instructions: $\alpha_{T0} \cdot \beta/(\alpha_{T0} + \alpha_{T1})$ for T_0 and $\alpha_{T1} \cdot \beta/(\alpha_{T0} + \alpha_{T1})$ for T_1. By considering the fraction of the time in serving the long latency miss event, the effective execution rate can be also derived. Table 1 summarizes the calculation of the effective execution rate under different scenarios. These values are used as the background steady-state execution rates of the performance model in the presence of SMT. Note that these estimations are based on the assumption that IQ uses the oldest-first policy to dispatch ready instructions.

3.3 Non-overlapped L2 Load Misses

For a given application, the number of non-overlapped L2 Load Misses (LLM) is affected by two factors: the L2 cache size, which determines the total number of L2 load misses, and the ROB size, which controls the amount of exposed MLP. Therefore, when both ROB size and L2 cache size can be reconfigured, their compounded effect has to be modeled in order to estimate the number of non-overlapped LLM.

To do so, we introduce the *load histogram* to hold the statistics of the number of loads occurred within a certain ROB size. Specifically, each time when the number of retired instructions equals the given ROB size, the number of loads observed in those retired instructions is used as an index to the load histogram, and corresponding entry in the load histogram is incremented by one. With the load histogram, we are able to model the "*window*" effect the ROB has on the non-overlapped LLM. As illustrated in Pseudocode 1, if the calculated number of LLM in an instruction window is less

Table 1: Estimation of Average Execution Rate for 2-Way SMT

Cases:	Effective Average Execution Rate		Notes
	Thread 0 (T_0)	Thread 1 (T_1)	
$\alpha_{T0} < \beta, \alpha_{T1} < \beta$, $\alpha_{T0} + \alpha_{T1} < \beta$	α_{T0}	α_{T1}	α_{T0}: average ILP of thread 0 α_{T1}: average ILP of thread 1 β: issue width of the core f_{T0}: the fraction of time that thread 0 is in long latency events f_{T1}: the fraction of time that thread 1 is in long latency events
$\alpha_{T0} < \beta, \alpha_{T1} < \beta$, $\alpha_{T0} + \alpha_{T1} > \beta$	$\frac{\alpha_{T0}*\beta}{\alpha_{T0}+\alpha_{T1}} * (1 - f_{T1}) + \alpha_{T0} * f_{T1}$	$\frac{\alpha_{T1}*\beta}{\alpha_{T0}+\alpha_{T1}} * (1 - f_{T0}) + \alpha_{T1} * f_{T0}$	
$\alpha_{T0} > \beta, \alpha_{T1} < \beta$, $\alpha_{T0} + \alpha_{T1} < 2\beta$	$\frac{\alpha_{T0}*\beta}{\alpha_{T0}+\alpha_{T1}} * (1 - f_{T1}) + \beta * f_{T1}$	$\frac{\alpha_{T1}*\beta}{\alpha_{T0}+\alpha_{T1}} * (1 - f_{T0}) + \alpha_{T1} * f_{T0}$	
$\alpha_{T0} > \beta, \alpha_{T1} < \beta$, $\alpha_{T0} + \alpha_{T1} > 2\beta$	$\frac{2*\beta-\alpha_{T1}}{2.0} * (1 - f_{T1}) + \beta * f_{T1}$	$\frac{\alpha_{T1}}{2.0} * (1 - f_{T0}) + \alpha_{T1} * f_{T0}$	
$\alpha_{T0} < \beta, \alpha_{T1} > \beta$, $\alpha_{T0} + \alpha_{T1} < 2\beta$	$\frac{\alpha_{T0}*\beta}{\alpha_{T0}+\alpha_{T1}} * (1 - f_{T1}) + \alpha_{T0} * f_{T1}$	$\frac{\alpha_{T1}*\beta}{\alpha_{T0}+\alpha_{T1}} * (1 - f_{T0}) + \beta * f_{T0}$	
$\alpha_{T0} < \beta, \alpha_{T1} > \beta$, $\alpha_{T0} + \alpha_{T1} > 2\beta$	$\frac{\alpha_{T0}}{2.0} * (1 - f_{T1}) + \beta * f_{T1}$	$\frac{2*\beta-\alpha_{T0}}{2.0} * (1 - f_{T0}) + \beta * f_{T0}$	
$\alpha_{T0} > \beta, \alpha_{T1} > \beta$	$\frac{\beta}{2.0} * (1 - f_{T1}) + \beta * f_{T1}$	$\frac{\beta}{2.0} * (1 - f_{T0}) + \beta * f_{T0}$	

Pseudocode 1 Non-overlapped L2 Load Miss Estimation

```
    #def $N_l$ //maximum number of loads in the ROB size $i$
    #def $N_{novp}$ //number of non-overlapped L2 load misses
    #def $MLP_i$ //average load MLP rate in ROB size $i$
    #def $ld\_miss\_rate$ //L2 load miss rate
    #def $ld\_hist_i[N_l]$ //load histogram for ROB size $i$

 1  for ( j=0; $j < N_l$; j++ )
 2    if ($j * ld\_miss\_rate < 1$)
 3        $temp = ld\_hist_i[j] * j * ld\_miss\_rate$;
 4    else
 5        if ($j * ld\_miss\_rate/MLP_i < 1$)
 6          $temp = ld\_hist_i[j]$;
 7        else
 8          $temp = ld\_hist_i[j] * j * ld\_miss\_rate/MLP_i$;
 9        end if
10    end if
11    temp_novp = temp_novp + $temp$;
12  end for
13  $N_{novp}$ = $ceiling$(temp_novp);
```

than 1 (line 2), there is no overlapped LLM and MLP is not considered. Otherwise, this number is divided by MLP. A result less than 1 (line 5) means all L2 load misses are overlapped and the number of non-overlapped LLM is 1. The total number of non-overlapped LLM can be obtained by accumulating these values in all cases. By using a set of load histograms with each dedicated to a certain ROB size, we are able to estimate the non-overlapped LLM for different ROB sizes. On the other hand, the L2 load miss rates for different L2 cache sizes can be estimated with the stack distance model, which is explained in section 4.3.

Figure 3 shows the accuracy of the estimation technique for program *libquantum* under different ROB and L2 cache sizes. We observe a close match between the measured and the estimated non-overlapped L2 load misses when both ROB size and L2 cache size vary. We also validate this technique using other SPEC CPU2006 programs, and we observe the average error rate of the estimation is 12.2%. Most of the errors are caused by the artifact that a small number of L2 load misses leads to a large relative error even though the absolute difference between the measured and the estimated is small. However, since a small number of L2 load misses means a small impact on the overall CPI, the influence of the estimation error passed down to the estimated CPI is also insignificant.

4. ONLINE PROFILER

The proposed performance model requires a set of program characteristics from which the key parameters for the model can be derived. These characteristics include: a). the critical dependency chain, for deriving the average ILP; b). the dependent load miss statistics, for estimating the memory level parallelism under different ROB sizes; c). the stack distance statistics [18], for estimating the number of L2 load misses with different L2 cache sizes. In this section, we present a set of non-invasive and cost-effective online profilers to dynamically extract these characteristics during the application's execution.

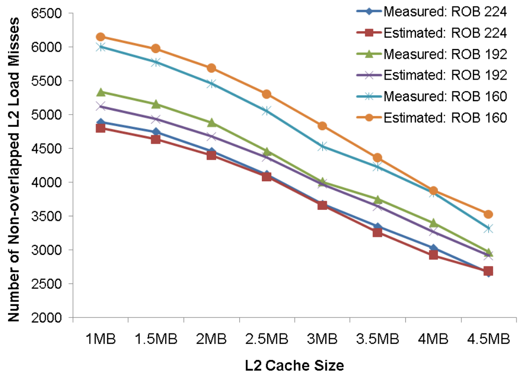


Figure 3: Comparison of the estimated and measured non-overlapped L2 load misses for SPECCPU2006 program *libquantum*. Data are collected at a 2M instructions interval.

4.1 Critical Dependency Chain Profiler

The critical dependency chain in this paper refers to the longest instruction dependency chain in the instruction window. To capture the length of the critical dependency chain, we propose a token-passing technique inspired by Fields et al's work [12]. A token is a field in each issue queue entry that keeps track of the dependency chain length, as shown in Figure 4(a). When an instruction enters the issue queue, its token field is set to zero; when an instruction leaves the issue queue for execution, its token field is incremented by one. The incremented token is propagated along with the result tag of the instruction. When the instruction finishes execution and its result tag matches the source tag of the waiting instruction in the issue queue, the propagated token also compares the token of the waiting instruction. The larger one between these two is stored in the token field of the waiting instruction. Hence, by the time an instruction is ready for execution, its token holds the length of the longest dependency chain for this instruction.

For each thread, the critical dependency chain profiler compares the token of every issued instruction of that thread, and keeps track of the maximum observed token, which is further used as an index to the critical dependency chain histogram. The histogram is controlled by an instruction counter that monitors the number of issued instructions. When this number reaches the interested ROB size, the histogram entry indexed with the maximum observed token is

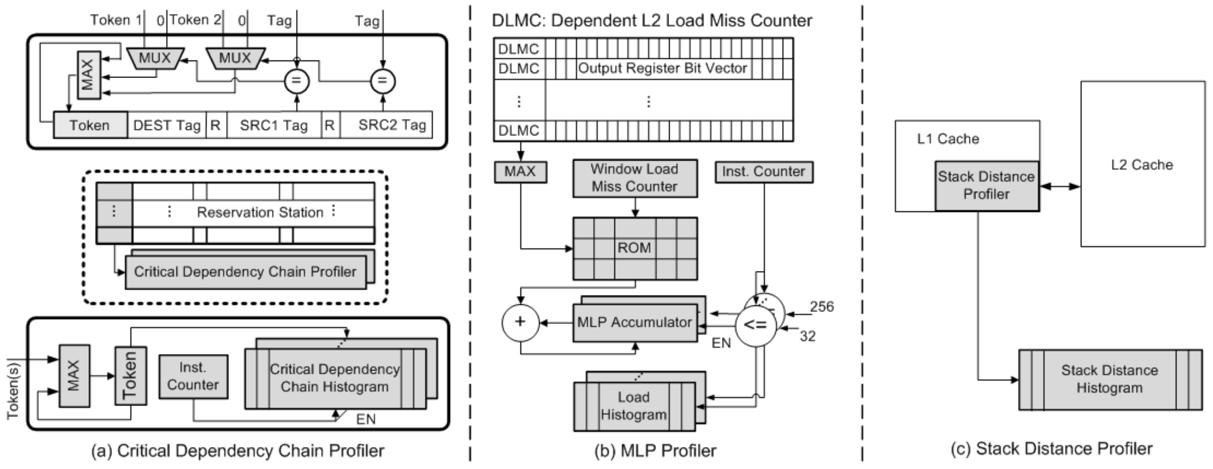


Figure 4: The structure of the online profilers.

incremented by 1. Meanwhile the register that holds the maximum token is reset to zero. Consequently, the critical dependency chain histogram holds the information of the longest dependency chain length for each instruction window. At the end of each epoch, this histogram is used to calculated the average length of the critical dependency chains, and then reset to zeros for the next epoch.

In order to obtain the dependency chain length for different ROB sizes, we need a set of critical dependency chain histograms, with one histogram dedicated to one specific ROB size. All histograms share one instruction counter to count the number of issued instructions. When the number equals one of the interested ROB sizes, the corresponding histogram is updated, and the counter continues counting until it equals the largest ROB size. Then, the counter is reset and starts counting from zero again. In this way, the token fields designed to profile for the largest ROB size can be reused by multiple histograms for different ROB sizes.

4.2 MLP Profiler

The MLP profiler is to capture the L2 load miss parallelism for different ROB sizes. As shown Figure 4(b), this profiler contains a L2 *Load Miss Event Table* (LMET), which has a *Dependent Load Miss Counter* (DLMC) and a *Output Register Bit Vector* (ORBV) in each table entry, similar with the one proposed by Eyerman and Eeckhout [10]. Each time a load that missed L2 cache is retired, a new entry in the table is created and the corresponding DLMC is updated with the number of L2 load misses that this load is dependent on in the current window. Meanwhile, the ORBV is initialized by setting '1' to the bit indexed by the output register ID of this load, and setting '0' to the remaining bits. Each retired instruction thereafter needs to check its dependency on this long-latency load by looking up the ORBV bit at the position corresponding to the input register ID of the retired instruction. A '1' in this bit position indicates this instruction depends on the previous long-latency load, and hence the bit indexed by the output register ID of the retired instruction is also set to '1'; whereas a '0' means this instruction is independent with the previous long-latency loads, and no further actions is needed. This process continues until the number of analyzed instructions reaches the largest ROB size of interest, in this paper, 256, and then the table is reset.

Besides the load miss event table, the profiler also has a MLP lookup table, which is a Read-Only-Memory (ROM) structure populated with pre-computed MLP values. The MLP value is obtained by dividing the its column index with the row index, and is represented in a 8-bit fixed-point format with 4 bits for integer and 4 bits for fraction. Each time when the analyzed instruction number equals an interested ROB size R, the MLP table is looked up by the largest DLMC in LMET and the Window Load Miss Counter (WLMC) that holds the number of L2 load misses occurred in the ROB window. The corresponding MLP value is then added to the MLP accumulator associated with the interested ROB size R. At the end of each epoch, the average load MLP rate of ROB size R can be obtained by dividing the values in the MLP accumulator with the number of accumulations occurred on this accumulator in the epoch.

The profiler also has a load histogram for each possible ROB size. The histogram collects the number of loads occurred in each ROB window, and is used to estimate the non-overlapped L2 load misses.

4.3 Stack Distance Profiler

To estimate the number of L2 load misses for different cache sizes, we employ the previously proposed Mattson's stack distance histogram at the granularity of cache ways [18][20]. This stack distance model exploits the inclusion property of Least Recently Used (LRU) replacement policy, i.e., the content of an N-way cache line is a subset of the content of any cache line with associativity larger than N. As an example, figure 5 shows the MSA histogram of program *xalancbmk* on an 8-way associative cache, organized from MRU position to LRU position. For caches with its associativity reduced to 6-ways (dash line in the figure), the data with stack distance larger than 6 could not be hold in the cache, generating cache misses. Therefore, with the stack distance histogram, we are able to estimate the cache miss rate for any cache ways less than the profiled ways and consequently derive the number of L2 misses.

Profiling the stack distance requires an Auxiliary Tag Directory (ATD) and hit counters for each cache set [20]. The ATD has the same associativity with L2 cache in the chip and uses LRU replacement; whereas the hit counter counts the number of hits on each cache way. To reduce the hardware overhead caused by ATD, we employ the Dynamic Set Sampling (DSS) technique, which essentially uses a few sets (in our case 32 sets) to approximate the entire cache behavior [20].

4.4 Profiling for Other Parameters

Other parameters in the performance model can be obtained from the standard performance counters. For example, the performance counters in Intel$^{®}$ Core^{TM} architecture [1] are able to provide the instruction mix and cache hit/miss statistics. With these statistics,

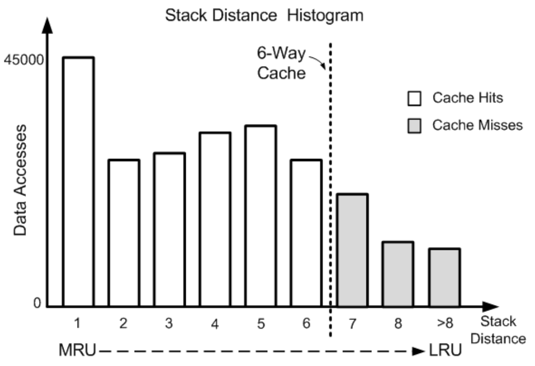


Figure 5: Stack Distance Histogram of SPEC CPU2006 program *xalancbmk*.

the average latency lat_{avg} can be derived by weight-averaging the percentage of each instruction type with the corresponding execution latency. Note that the load that misses L1 cache but hits in L2 cache is treated as an instruction with long execution latency.

5. ALLOCATION ALGORITHMS

With the online profilers and the performance predictor, the performance of the application under different resource allocations can be estimated by simply evaluating an equation, which fundamentally eliminates the need of trial runs and significantly improves the quality and efficiency of multiple resource management.

Pseudocode 2 Coordinated Predictive Hill-Climbing

```
#def $N_{tt}$ //total number of threads
#def $N_{res}$ //the number of resources independently partitioned
#def $delta$ //resource partition granularity
#def $P_{th}$ //convergence threshold
#def $part[0 : N_{tt}][0 : N_{res}]$ //the resource partition array
#def $max\_id(A, n)$ //get the index of the largest value in A[0:n]
#def $max(A, n)$ //get the largest value in A[0:n]
#def $perf\_eval(part)$
//estimate the overall performance for resource array $part$
#def $perf(part, i)$
//estimate the performance of thread $i$ for resource array $part$

1  old_part_perf = $perf\_eval(part)$;
2  copy $part[0 : N_{tt}][0 : [N_{res}]$ to $temp\_part[0 : N_{tt}][0 : N_{res}]$;
3  while(TRUE)
4    for( $i = 0; i < N_{res}; i$++)
5      for( $j = 0; j < N_{tt}; j$++ )
6        $temp\_part[i][j] = part[i][j] + delta$;
7        $pos\_perf[j] = perf(temp\_part, j)$;
8        $temp\_part[i][j] = part[i][j] - delta$;
9        $neg\_perf[j] = perf(temp\_part, j)$;
10     end for
11     $pos\_tid[i] = max\_id(pos\_perf, N_{tt})$;
12     $neg\_tid[i] = max\_id(neg\_perf, N_{tt})$;
13     if($max(pos\_perf, N_{tt}) > max(neg\_perf, N_{tt})$)
14       $part[pos\_tid[i]][i] = part[pos\_tid[i]][i] + delta$;
15       $part[neg\_tid[i]][i] = part[neg\_tid[i]][i] - delta$;
16     end if
17   end for
18   $new\_part\_perf = perf\_eval(part)$;
19   if ( $abs(new\_part\_perf - old\_part\_perf) < P_{th}$) break;
20   else $old\_part\_perf = new\_part\_perf$;
21 end while
```

To efficiently manage multiple resources, this paper presents a predictive and coordinated resource management algorithm that leverages the performance predictor to identify the optimum resource distribution for the workload. As shown in Pseudocode 2, the proposed algorithm uses *hill-climbing* to search for the appropriate resource distribution, hence the name *Coordinated Predictive Hill-Climbing* (CPHC). Specifically, it first uses the performance model to evaluate the performance of each thread as one of the resources is incremented or decremented by a certain amount *delta* (line 5 to line 10). It then moves *delta* amount of the resource from the thread that has the lowest performance degradation to the thread that benefits most from the additional resource, provided that the overall performance gain is positive (line 13 to line 16). This process iterates through different resources, and repeats itself until the estimated performance reaches the given target or no noticeable performance gain is attainable (line 19). In this way, this algorithm explores the resource allocation in the positive-gradient direction, and hence achieves fast convergence.

In this algorithm, power as a resource is *indirectly* managed by controlling the operating frequency of each core in a CMP. Specifically, for a quad-core CMP, the total power consumption can be written as $a_1v_1^2f_1 + a_2v_2^2f_2 + a_3v_3^2f_3 + a_4v_4^2f_4$, where v_i and $f_i(i = 1..4)$ are the voltage and frequency of core i respectively, and $a_i(i = 1..4)$ is the product of the activity factor and the effective capacitance for core i. In a fully-loaded CMP system, the power is usually consumed as close to the given power budget as possible to maximize performance, and $a_1, .., a_4$ are generally very close to each other. Therefore, the problem of power management can be transformed to the problem of allocating frequencies such that $v_1^2f_1 + v_2^2f_2 + v_3^2f_3 + v_4^2f_4$ remains constant. Note that the frequency and voltage are correlated with each other under DVFS, and for a given frequency, the corresponding voltage can be found by looking up a table. Therefore, by controlling the frequencies, the power can be allocated the same way as other resources.

Besides this proposed algorithm, we also evaluate a set of other resource allocation algorithms for comparison, which include:

Equal Partition: This algorithm distributes all shared resources equally among the threads. Specifically, the inter-core resources are equally partitioned for all active threads in the CMP, and the intra-core resources are equally partitioned for the threads that are simultaneously executed in the core. This algorithm is used as the baseline management scheme in this paper.

Coordinated Reactive Hill-Climbing (CRHC): Like the proposed predictive scheme, this algorithm also attempts to manage both intra-core and inter-core resources, but without a performance prediction model. Therefore, it has to rely on trial runs to explore the gradient direction for resource allocation. Specifically,the algorithm randomly selects two threads (for inter-core resource) or a pair of co-executing threads (for intra-core resource), tentatively moves *delta* amount of resource from one thread to the other, and runs the workload for one epoch. It then moves the resource in opposite direction for these two threads, and runs the workload for another epoch. The resource allocation that gives the higher performance during these two trial runs is enforced in the next epoch. The process keeps on repeating itself for different resources and different threads.

Intra-core Reactive Hill-Climbing (Intra-RHC): This algorithm is similar with the one proposed by Choi et al. [9], and it uses trial runs to search for the appropriate resource allocations. The resource adaptation only happens on the intra-core level, and the inter-core resources are equally partition for all threads.

Inter-core Reactive Hill-Climbing (Inter-RHC): This algorithm is similar with CRHC except that the resource adaptation only happens on the inter-core level, and the intra-core resources are equally partition for the co-executing threads in the core.

Oracle: This algorithm assumes the application's performance under different resource allocation in the next epoch is known *a priori*. It uses these *future* performance data to enforce the resource allocation that gives highest performance in the next epoch. While it is unrealistic in practice, it sets an upper bound of the potential performance improvement.

6. IMPLEMENTATION COST ANALYSIS

Both the on-line profilers and the resource allocator are implemented in hardware, and they are the major sources of the implementation cost in the proposed framework. The cost of the profilers depends on the ROB size, the L2 cache size, the number of SMT threads, as well as the partition granularity. Assuming a 256-entry ROB with 32-entry partition granularity, 160 issue queue size, 32-bit physical address space, 16MB 32-way shared L2 cache, and 2-way SMT, the total hardware cost amounts to approximately 22KB, as shown in Table 2. Under this circumstance, the dimension of MLP lookup table is set to 16-by-16, and the profilers need 8 critical dependency chain histograms and 8 load histograms since there are 8 possible ROB sizes. Note that the hardware cost may be further reduced by using a smaller number of histogram counters based on the observation that the critical dependency chain length is far smaller than the ROB size. However, even without such optimization, the hardware overhead incurred by the online profilers only amounts to 0.14% of the 16MB L2 cache size. Note also that these profilers are not in the critical path, and does not affect the application's execution.

Table 2: Hardware Cost of the Online Profilers

Profiler	Components	Costs
Critical Dependency Chain Profiler	token fields	8*256 bits
	multiplexors, comparator	(8*2+8)*160bits
	histogram counters	16*256*8*2bits
MLP Profiler	LMET	(4+32)*16*2bits
	MLP accumulator	16*8*2 bits
	WLMC	5*8*2 bits
	MLP lookup table	16*16*8 bits
	comparators	8*8*2 bits
	load histogram	16*256*8*2 bits
Stack Distance Profiler	valid bits per ATD entry	1 bits
	addr. bits per ATD entry	12 bits
	total ATD cost (32 sampled sets, 2 threads)	(3+1+12)* 32*32*2 bits
	Hit Counters	16*32*2 bits
Total Cost of Profilers per core		21812 Bytes

On the other hand, the cost of the resource allocator is mainly caused by converting the profiled histograms to the parameters for the performance model and searching for the appropriate resource allocation with the performance model. For example, to obtain the average critical dependency chain length from the dependency chain histogram, approximately 300 multiply-add operations are required. To further quantify the hardware cost, we implemented the resource allocator in Verilog HDL, and synthesized it into a netlist. The design employs pipelining so that arithmetic units can be reused. Overall, it has two adders, two multipliers and one divider, all in 32-bit fixed-point. The total area of the resource allocator is estimated to be 0.632 mm^2 under 65nm technology. Each performance estimation requires 20 cycles to complete, and the search process takes less than 30000 cycles before it converges (we enforce convergence if the iterations is larger than 20). Since the resource allocation is made only once every epoch, the latency can be completely hidden by starting resource exploration procedure several thousands of instructions before the end of the epoch.

7. EXPERIMENT METHODOLOGY

7.1 Simulation Platform

We use Simics [16], extended with the Gems toolset [17], to simulate a quad-core SPARCv9 CMP system running under OpenSolaris operating system. Each core in the CMP is a 4-issue out-of-order processor and supports 2-way SMT with ICOUNT [23] instruction fetch policy. The simulated CMP system also contains a detailed memory subsystem model, which includes an inter-core last-level cache network and a detailed memory controller. Table 3 lists the configurations of the CMP system in detail. We use Wattch [4] to estimate the dynamic power of the processor as well as the resource allocator, and use Cacti 5 [22] to estimate the leakage power on caches and other SRAM structures in the core. We use Orion [24] to estimate the power on the interconnection network of last level caches. These estimated power data are used in evaluating the efficiency of the system.

Table 3: Configurations of the CMP system

	Parameter	Configurations
Core	Max. Clock Frequency	4GHz
	Fetch/Issue/Commit	4/4/4
	Ld/St Units	2/2
	I-ALU	4(fused multiply/add for I-ALU)
	FP Units/FP Multipliers	4/2
	ROB size/Issue Queue	256/160
	Load/Store Queue Size	64/64
	Branch Predictor	YAGS, 16 PHT bits, 10 Tag bits
	Physical Register Number	380
Cache	L1 I-Cache/D-Cache	32KB, 2-way, 64B, LRU, 1 cycle
	L2 Cache size	16MB shared
	L2 Cache parameter	32-way, 64B, LRU, 12 cycle
	L2 MSHR Entry	32
	Coherence Protocol	Directory-based MOESI
Memory	Size/Model	4GB/DDR2-800
	Controller	PAR-BS policy [19]
	Organization	8 banks per rank, 2 ranks per DIMM

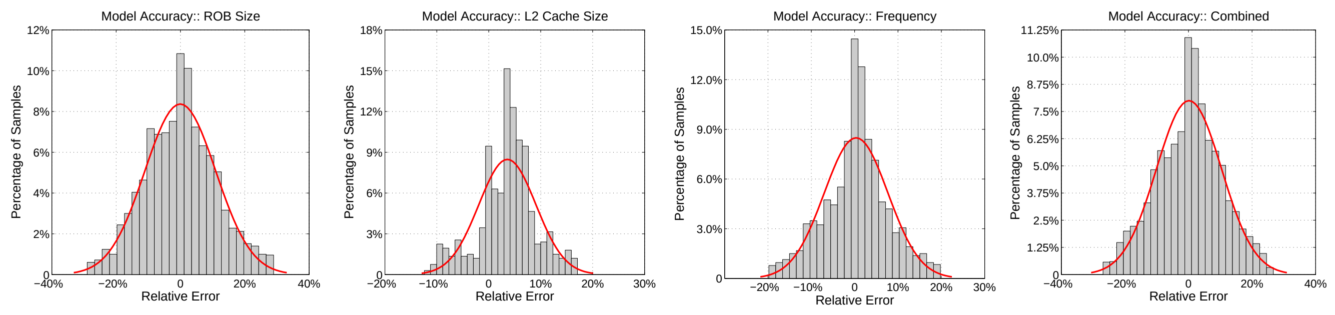
The ROB in the core is partitioned at the granularity of 32 entries. Other intra-core resources such as issue queue size and physical register number are partitioned in proportion to the ROB size. Each thread is guaranteed to have at least 32 entries of ROB size. The L2 cache size is partitioned at the granularity of cache ways, with each thread allocated with at least one cache way. The CMP system supports per-core DVFS, with the frequency of each core ranging from 2GHz to 4GHz at the step of 0.1GHz. We assume that the CMP system reaches the power budget when it is fully loaded and each core is running at 3GHz.

Table 4: Workloads and Their Characteristics

Workload Mix	Symbol	Category
povray, calculix, sjeng, hmmer perlbench, wrf, dealII, tonto	pcshpwdt	ILP
gcc, povray, astar, calculix gobmk, hmmer, bzip2, dealII	gpacghbd	
astar, bzip2, gobmk, povray sjeng, perlbench, dealII, gamess	abgpspdg	
namd, gcc, gromacs, perlbench h264ref, tonto, sphinx3, sjeng	nggphtss	
mcf,omnetpp,bwaves,lbm povray, namd, gcc, xalancbmk	moblpngx	MIX
dealII, sjeng, libquantum, omnetpp povray, soplex, perlbench, milc	dslopspm	
libquantum, cactusADM, xalancbmk calculix,wrf,mcf, soplex, omnetpp	lcxcwmso	
leslie3d,tonto,sphinx3, omnetpp hmmer, libquanutm, astar, zeusmp	ltsohlaz	
soplex, xalancbmk, milc, lbm mcf, cactusADM, zeusmp, leslie3d	sxmlmczl	MEM
leslie3d,soplex, zeusmp, bwaves wrf, cactusADM, xalancbmk, lbm	lszbwcxl	
lbm, milc, xalancbmk, leslie3d zeusmp, wrf, mcf, soplex	lmxlzwms	
milc, xalancbmk, mcf, cactusADM soplex, leslie3d, bwaves, wrf	mxmcslbw	

7.2 Workloads

The workload of the experiment is composed of the programs from SPEC CPU2006 benchmark suite [2], with each compiled



(a) (b) (c) (d)

Figure 6: Performance Model Accuracy. (a)The ROB size varies from 32 to 256 at the step of 32. (b)The L2 cache size varies from 512KB to 4MB at the step of 512KB. (c) Frequency varies from 2GHz to 4GHz at the step of 0.1GHz. (d) 500 random configurations when all three resources vary simultaneously.

to SPARC ISA. We construct 12 heterogeneous multiprogrammed workloads, each containing 8 programs, as shown in Table 4. These workloads are grouped into three categories: CPU-intensive (high-ILP), memory-intensive, and the mixture of both. Each workload will be running on the aforementioned CMP systems. For each run, we fast-forward the workload for 4 billion instructions to reach its steady state execution, and then use the next 100 million instructions to warmup the cache subsystem. We then simulate the full system for 200M instructions to evaluate the performance of various resource allocation policies.

7.3 Metrics

The metric we use to evaluate the system performance is the weighted speedup, which is defined as $\sum_i IPC_i^{shared}/IPC_i^{alone}$ [21]. To measure the efficiency of the system, we use the metric $mips^3/W$, which is inverse to energy-delay-square (ED^2) and has been accepted as the efficiency metric for high-performance systems [5].

8. EVALUATION

8.1 Model Accuracy

The accuracy of the performance model could largely impact the effectiveness of the proposed resource management framework. To evaluate the model accuracy, we run every SPEC CPU2006 program on a simulated processor for an interval of 2 million instructions, and use the performance model to estimate the program's CPI on processors with different resource configurations. Meanwhile, we also simulate the program on those processors for the same interval and compare the observed CPI values with the estimated ones. Figure 6(a)-(c) show the accuracy when only one resource changes. As we can see, the relative error between the estimated CPI and the observed one follows normal distribution. The average errors (using absolute values) are 8.7% for different ROB sizes, 5.3% for different L2 cache sizes, and 6.7% for different frequencies, indicating the performance model tracks well with the observed performance when only one resource varies its configuration. Figure 6(d) further shows the relative estimation error for 500 random configurations when all three resources vary simultaneously. The average CPI estimation error in this scenario is 8.1%, and the largest one is 26.7%. We also observe that this relative error follows normal distribution.

8.2 Epoch Size Sensitivity

The epoch size determines the frequency of resource adaptation during the execution of the workload, and can indirectly influence the overall performance of our resource management framework. Figure 7 shows the performance trend of three workloads as the epoch size increases from 0.5 million to 5 million instructions. We observe that as the epoch size increases, the weighted speedup first increases, then reaches a plateau, and then gradually decreases. This is because with a relatively small epoch size, the on-line profilers may not be fully warmed up to capture the corresponding program characteristics, which could affect the accuracy of the performance predictor, and in turn pulls down the performance of the resource management. This is particularly true for the stack distance profiler since this profiler employs set sampling technique, which provides a good accuracy only when it has be exercised with sufficient amount of L2 accesses. On the other hand, a large epoch size would miss the opportunity for adapting resource distribution to some finer grain program phases, which also degrades the end performance. In this work, we find that 2 million instruction is a reasonable epoch size that balances the accuracy of the performance predictor and the responsiveness of the resource allocation.

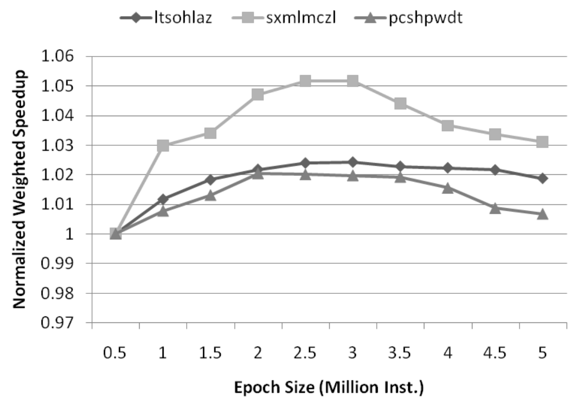
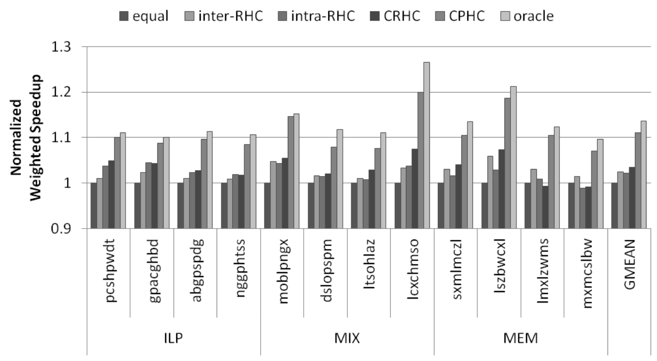


Figure 7: Performance impact of epoch size.

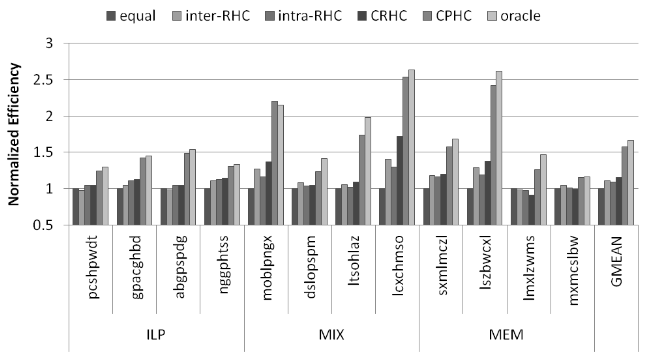
Note that such choice of epoch size is based on the assumption that different voltage and frequency pairs can be enforced instantaneously. In practice, this is not true because it may take the voltage regulator hundreds of micro-seconds to stabilize voltage. Under such circumstance, the epoch size need to incorporate this additional time for voltage regulation.

8.3 Performance & Efficiency

Figure 8(a) shows the comparison of the weighted speedups between different resource allocation policies. As expected, equal partition policy usually yields lowest weighted speedup among all



(a) Improvement in Weighted Speedup



(b) Efficiency Improvement

Figure 8: Performance and efficiency comparison for different resource management policies.

the policies investigated in this paper. Inter-RHC and Intra-RHC improves the performance over equal partition policy as it dynamically adapts allocations for either inter-core or intra-core resources. CRHC further improves the weighted speedup, as it attempts to adjust the resource allocation on both inter-core and intra-core level. However, for some workloads, these reactive allocation policies may leads to inferior performance compared with equal partition. This is because they rely on the trial runs to search for the appropriate resource allocation, which means workloads may spend some trial runs in an inappropriate resource allocation. That also explains why these dynamic policies only have a small improvement over the equal partition policy. Our proposed predictive hill-climbing scheme avoids trial runs, and achieves an average of 11.6% over the baseline scheme and 9.3% over the CRHC scheme. In general, CPHC yields higher speedup in the workloads that belong to the MIX category because in such workloads, the resource requirements of the programs are more diversified, resulting in higher potential for resource management. Compared with the oracle scheme, the CPHC has approximately 3% less speedup. This is attributed to: (a) the imperfection of the performance model;(b) the lack of future knowledge of program phase behavior; (c) hill-climbing being trapped in local optima.

Figure 8(b) further shows the efficiency improvements for different resource allocation policies. We observe that CPHC has an average efficiency improvement of 57.4% over the baseline, and 36.5% over CRHC.

8.4 QoS Enforcement

The QoS target is defined as the target IPC relative to the alone-execution IPC, expressed in the form of percentages [13][7]. The proposed resource management framework can convert this QoS target into resource usage requirements [13], thereby enforce QoS for an application by regulating the amount of allocated resources. The quality of such QoS enforcement is demonstrated in Figure 9, where for each workload, only one program is enforced with the QoS targets and the remaining programs do not have QoS objectives. The resource allocator attempts to satisfy the QoS target for that program and maximize the overall performance for the remaining programs. As we can see, the relative IPCs of the programs keep a good track of the QoS targets. For some programs, such as $povray$, gcc, and $astar$, the relative IPC at the 20% QoS target is significantly off the target. This is because even with the minimum allocation on each resource, the relative performance of these programs are still much larger than 20%. Hence, such QoS target is *ill-suited* for these programs. Overall, we observe that the proposed framework could enforce QoS within 6.1% for 80% target, 6.7% for 60% target, and 5.9% for 40% target. Hence, this framework is suitable for the enforcement of elastic QoS objectives [13].

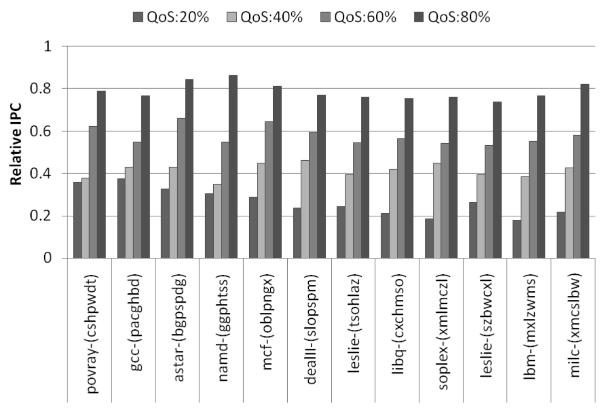


Figure 9: QoS targets enforcement.

9. RELATED WORK

Dynamic Resource Partition for SMT Threads: Cazorla et al. [8] proposed a DCRA mechanism to dynamically allocate shared resources to each thread in an SMT processor. Their method uses a resource sharing model to estimate the thread's anticipated resource needs, and allocate resources to the thread that utilizes the resource most efficiently. Like other SMT resource sharing policies [23], this method improves the SMT performance only *indirectly*, not only potentially missing opportunities for further performance improvement but also unable to control the end performance. Choi and Yeung [9] improves the SMT resource distribution by *directly* using the performance feedback to partition the resources for a specific performance goal. Their method requires a number of trial resource partitions before it learns the appropriate resource distribution, fundamentally limiting its potential for performance improvement. In contrast, our work uses an analytical model to predict the performance, hence eliminating the need of trial partitions. In addition, our work coordinates both intra-core and inter-core resources, whereas previous SMT resource partition techniques only consider the intra-core resources.

CMP Shared Resource Partition: Qureshi and Patt [20] propose to partition the last level cache to prevent negative interference between threads and maximize the utilization of the cache capacity. However, this utility based cache partitioning scheme is only applicable to cache, and can not manage the partition of multiple resources. Isci et al. [14] propose to manage global power by estimating the performance impact of per-core DVFS with an analytical model. Again, their proposal is only applicable to managing power alone. Bitirgen et al. [3] proposed a technique based on online machine learning to manage multiple CMP resources. However, the on-line machine learning model requires extensive training and retraining before it can accurately predict the application's performance. Moreover, it incurs significant hardware cost and is hard to implement and validate. On the contrary, our scheme uses cost-effective online profilers and an analytical model to predict the performance. It does not require any training and could manage the partition of multiple inter-core and intra-core resources.

Resource Partition for QoS: Guo et al. [13] propose a mechanism to support QoS in CMP by controlling the L2 cache allocation. Cazorla et al. [7] leverages OS-processor interaction to achieve QoS in SMT processors. These works address QoS problem separately at either inter-core or intra-core level. In contrast, our framework provides QoS support by coordinating both inter-core and intra-core resources.

10. CONCLUSIONS

This paper presents a showcase study of using an on-line analytical model to manage multiple interacting resources for throughput and QoS. In this paper, we find that for a Chip Multiprocessors (CMP) supporting per-core Simultaneous Multithreading (SMT), both intra-core and inter-core resources need to be managed simultaneously in order to achieve high resource utilization and deliver controllable performance. We thereby present a predictive resource management framework that coordinates both inter-core and intra-core resources. This framework uses a set of hardware-efficient online profilers and an analytical performance model to predict the application's performance with different intra-core and/or inter-core resource allocations. Based on the predicted performance, the resource allocator identifies and enforces near optimum resource partitions for each epoch without any trial runs. Our study shows that the proposed framework improves weighted speedup by an average of 11.6% compared with equal partition scheme, and 9.3% compared with the learning-based resource manager. We also show this framework enforces QoS targets within 6.7%. We believe that this predictive resource management framework offers a promising way to coordinate both inter-core and intra-core resources in CMPs.

11. ACKNOWLEDGMENT

The authors would like to thank anonymous reviewers for their valuable feedback. This work is supported in part through the NSF Award number 0702694. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of NSF.

12. REFERENCES

[1] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide*.

[2] Spec cpu2006 benchmark suit. In *http://www.spec.org*.

[3] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st Int'l Symposium on Microarchitecture*, pages 318–329, 2008.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Int'l symposium on Computer architecture*, pages 83–94, 2000.

[5] D. M. Brooks, et al. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[6] J. Casazza. White paper first tick, now tock: Intel microarchitecture (nehalem). 2009.

[7] F. J. Cazorla, et al. Predictable performance in smt processors. In *Proceedings of the 1st conference on Computing frontiers*, pages 433–443, 2004.

[8] F. J. Cazorla, et al. Dynamically controlled resource allocation in smt processors. In *Proceedings of the 37th Int'l Symposium on Microarchitecture*, pages 171–182, 2004.

[9] S. Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. In *Proceedings of the 33rd int'l symposium on Computer Architecture*, pages 239–251, 2006.

[10] S. Eyerman and L. Eeckhout. Per-thread cycle accounting in smt processors. In *Proceeding of the 14th ASPLOS*, pages 133–144, 2009.

[11] S. Eyerman, et al. A performance counter architecture for computing accurate cpi components. In *Proceeddings of the 11th ASPLOS*, pages 175–184, 2006.

[12] B. Fields, et al. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Int'l symposium on Computer architecture*, pages 74–85, 2001.

[13] F. Guo, et al. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th Int'l Symposium on Microarchitecture*, pages 343–355, 2007.

[14] C. Isci, et al. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th Int'l Symposium on Microarchitecture*, pages 347–358, 2006.

[15] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Int'l symposium on Computer architecture*, pages 338–349, 2004.

[16] P. Magnusson, et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2 2002.

[17] M. M. K. Martin, et al. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[18] R. L. Mattson, et al. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, 1970.

[19] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proceedings of the 35th Int'l Symposium on Computer Architecture*, pages 63–74, 2008.

[20] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Int'l Symposium on Microarch.*, pages 423–432, 2006.

[21] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of ASPLOS-IX*, pages 234–244, 2000.

[22] S. Thoziyoor, et al. Cacti 5.1. *HP Technical Reports*, 2008.

[23] D. M. Tullsen, et al. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Int'l symposium on Computer architecture*, pages 191–202, 1996.

[24] H.-S. Wang, et al. Orion: a power-performance simulator for interconnection networks. In *Proceedings. 35th Int'l Symposium on Microarchitecture*, pages 294 – 305, 2002.