Modeling Program Resource Demand Using Inherent Program Characteristics

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

ABSTRACT

The workloads in modern Chip-multiprocessors (CMP) are becoming increasingly diversified, creating different resource demands on hardware substrate. It is necessary to allocate hardware resources based on the needs of the workloads in order to improve system efficiency and/or ensure Quality-of-Service (QoS) at certain performance levels. Therefore, it is extremely important to identify the resource demand of the workload in terms of the performance and power efficiency. Existing models are inappropriate for estimating resource demands as they require either partial simulations or time-consuming training. This paper presents an integrated framework that is able to identify the single-resource or multi-resource demands on an array of hardware resources ranging from the issue width of the processor to the memory bandwidth. With an analytical model based on program inherent characteristics, this framework does not require any detailed simulation or training yet is still able to capture the performance trend of the program accurately. Our experiment shows that the proposed framework on average provides no larger than 8.6% error to any given performance target for multi-resource demand estimation. By using the proposed performance model, the framework identifies the multi-resource demands up to 40X faster compared to the state-of-the-art analytical model. The proposed framework can be applied in workload capacity planning, hardware resource adaptation as well as coordinated resource management for QoS in CMP systems.

Categories and Subject Descriptors

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

General Terms

Measurement, Performance

Keywords

Microprocessor, Resource Demands, Program Characteristics, Performance Modeling

1. INTRODUCTION

The workloads on modern general purpose or embedded computing systems are becoming increasingly abundant and diversified, imposing various demands on hardware resources, such as cache sizes and memory bandwidth. Efficiently modeling and identifying these resource demands is fundamental for many applications, including efficient single-ISA heterogeneous computing [17], resource management for throughput, power efficiency and/or Quality-of-Service (QoS) in Chip Multiprocessors (CMP) [8], and resource consolidation for balancing computation fidelity and response latency in embedded systems [21]. For example, when executing an application in a single-ISA heterogeneous multi-core processor, the application's resource demand needs to be carefully explored and classified so that the application can be properly scheduled to the core that matches its demand for energy efficient execution. Similarly, when managing resources in CMP systems, the resource demands of an application has to be efficiently identified before allocating appropriate hardware resources to meet the power and performance constraints.

While there are some methods proposed to identify the resource demands of the workloads [3][5][14], these methods suffer from either inefficiency or high implementation costs. Existing methods typically leverages analytical models [14][15], regression models [12][18], or neural network models [11] to estimate the performance and/or power of the application-processor pairs, and identify the resource demand of a workload by searching through the design space off-line for the optimum configurations. However, these models require either partial simulations or iterative training for each application, which is expensive and inefficient for resource demand estimation. Moreover, the requirement of partial simulations also implies that when using these models on-line, resource demands can be only identified by using tentative runs in a trial-and-error way [5], which may require many trial iterations and cause significant overhead in performance and energy. To avoid trial runs for resource demand estimation, some predictive schemes have recently been proposed, which include the marginal utility monitoring for last-level cache partitioning [23] and system-level bandwidth management [16], and the on-line machine learning for coordinated management of multiple resources [3]. However, these schemes are either limited to manage only single resource, or impractical to implement and validate.

Therefore, there is a need for a model that does not require any partial simulations or training, is easy to implement, yet still able to identify the demands on single and multiple resources accurately. This paper attempts to develop such model by leveraging the recent advances in analytical modeling and workload characterization. It exploits the fact that resource demand is estimated based on performance trend rather than absolute performance, and hence is insensitive to the second-order effects of the performance. In particular, the contributions of this paper are as follows:

- **Analytical Model Based on Program Characteristics:** We develop an analytical model based on the program character-

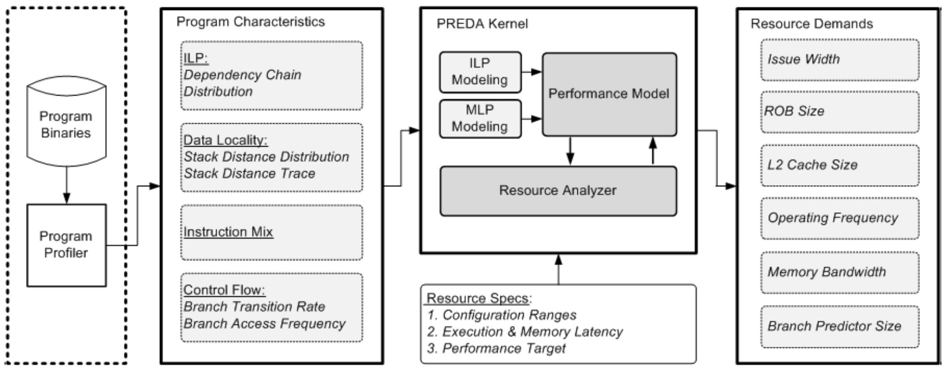


Figure 1: The PREDA framework

istics, such as Instruction Level Parallelism (ILP), Memory Level Parallelism (MLP) and branch predictability. Unlike existing analytical models [14][7], which require simulations on caches and branch predictors, our model avoids any partial detailed simulation; yet is still able to accurately model the performance trend for different hardware configurations. The experimental results show that the modeled performance trend is on average less than 10.7% off the simulated one.

- **Efficient Estimation of Multi-Resource Demands:** We propose a set of algorithms and heuristics that can efficiently estimate the demands on both single resource and the multiple resources under any given performance target. The algorithm for identifying the multi-resource demand is based on marginal utility [23] and the gradient performance gain, which leads to a fast convergence with only a few iterations. We show that the estimated multi-resource demands on average achieves no larger than 8.6% error to any given performance target.

- **Integrated Framework for Resource Demand Estimation:** We encapsulate the analytical model and the resource demand estimation algorithms into an integrated framework called *Program REsource Demand Analyzer* (PREDA), which automatically estimates a broad set of resource demands for a workload. Compared with the framework using state-of-the-art analytical model [7], our framework achieves up to 40X speedup in estimating multi-resource demands.

The rest of the paper is organized as follows. Section 2 gives the overview of the proposed PREDA framework. Section 3 presents the working mechanism of the PREDA kernel. Section 4 describes the experiment and evaluation methodology. Section 5 reports our experimental results. Section 6 compares our work with other related works, and section 7 concludes the paper.

2. OVERVIEW OF PREDA

2.1 Resource Demand Definition

Before we continue, it is important to make a clear definition of resource demand. The meaning of resource demand contains two elements: the performance target and the energy efficiency. On one hand, different levels of performance target may lead to different resource requirements. Specifically, as the performance target increases, the amount of resources required also increase. On the other hand, for a given performance target, there may be a set of different amounts of resources being able to meet that target. Among them, we are only interested in the one that is energy efficient. Therefore, we introduce the following resource demand definition:

Definition*: Resource Demand $D(p)$ is the amount of resource a thread requires to efficiently achieve no less than $p\%$ of the maximum performance achieved with the entire resources allocated to the thread.*

Note that this definition uses a relative term for the performance target because the absolute performance target, such as the Instruction-Per-Cycle (IPC) rate, may lead to *ill-defined* cases where the target cannot be satisfied no matter how many resources are allocated. The relative performance target avoids this problem, and more importantly it is inline with the satisfiability of the QoS target proposed by Guo et al. [8]. In fact, with this definition, our framework can be treated as a conversion layer that converts the performance targets into the resource demands, which could be used as the *Resource Usage Metrics* for QoS enforcement [8]. Note also that this definition assumes performance monotonicity, which means the performance of a thread increases monotonically as the amount of resource allocated to the thread increases [22].

2.2 PREDA Framework

The proposed PREDA framework consists of two parts: the program characteristics profiler and the PREDA kernel. As shown in Figure 1, the program profiler walks through the dynamic instruction stream and extracts a set of program characteristics, which contain instruction dependency chain distribution, stack distance distribution, instruction mix, and branch transition rate and its access frequency. These characteristics are then fed to the PREDA kernel, which consists of an ILP model, an MLP model, a performance model and a resource analyzer. The models for ILP and MLP are responsible to translate the program characteristics into the ILP and MLP information that can be directly used by the performance model. Hence, these models serve as the key layer to decouple the performance evaluation from detailed simulations. The performance model takes the ILP and MLP information along with the branch predictability characteristics and estimates the program execution time on an out-of-order processor. The resource analyzer converts the estimated performance into the relative performance, and searches the configuration space for the amount of resources required to meet the performance targets. The estimated resource demands include processor issue width, processor reorder buffer (ROB) size, L2 (or last level) cache sizes, operating frequency, memory bandwidth and branch predictor size. These resource demands are estimated either in single-resource mode (other resources are fixed) or in multi-resource mode (combinations of changing resources).

This framework is designed for off-line resource demand estimation, which can be applied in areas such as early-stage design space exploration in microprocessor design and admission control to balance the workloads in computing systems [13]. However, since the proposed performance model in this framework is decoupled from detailed simulation, it could also be applied online for dynamic resource management by using on-line profilers. Nevertheless, this paper focuses on evaluating the accuracy and the complexity of off-line resource demand estimation.

For the rest of the paper, we use 22 SPEC CPU2006 programs [1] to evaluate the proposed framework (*gamess*, *dealII*, *calculix*, *povray*, *tonto*, *lbm*,*wrf* are not included in the workload as we did not manage to compile them to Alpha ISA). Each of the 22 programs is compiled to Alpha-ISA with peak configurations, and we use the single Simpoint interval with 100 million instructions [9] for each program.

3. PREDA KERNEL

This section introduces the working mechanism of the PREDA kernel, which consists of ILP modeling, MLP modeling, performance modeling and resource analyzing.

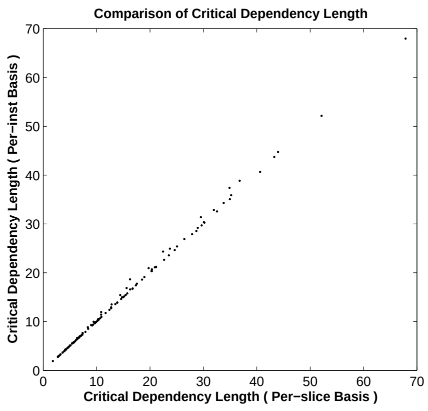
3.1 ILP Modeling

The goal of ILP modeling is to accurately estimate the background execution rate, i.e., the IPC rate the program can achieve when it is free of any miss events [14]. To do so, we employ the *critical dependency chain length* as the metric to measure the ILP of the program as it determines the number of instructions that have to be executed in serial, and hence sets the limit of the program's ILP. The same metric is also used by Eyerman et al. in their mechanistic model [7], but the difference lies in the profiling method to obtain the critical dependency chain. Eyerman et al. obtain the statistics of the critical dependency chain on a per-instruction basis. Although accurate, it requires time-consuming update in all dependency chains each time an instruction moves out of the instruction window. Our profiling scheme, however, chops the dynamic instruction stream into slices, each with the size of the *maximum* interested instruction window W_{max}. The statistics of critical dependency chain are collected on a per-slice basis, and the dependencies between adjacent slices are ignored. This profiling scheme has the complexity of $O(\frac{N}{W_{max}} \cdot W_{max}) = O(N)$ (N is the number of dynamic instructions), as compared with the complexity of $O(N \cdot W_{max})$ in Eyerman's scheme [7]. Yet, the observed difference between these two schemes is within 1.5% in terms of the profiled critical path statistics, as shown in Figure 2(a). Note that the profiler considers both register dependence and memory dependence when searching for the critical dependency chain because memory dependence could also serialize the instruction execution.

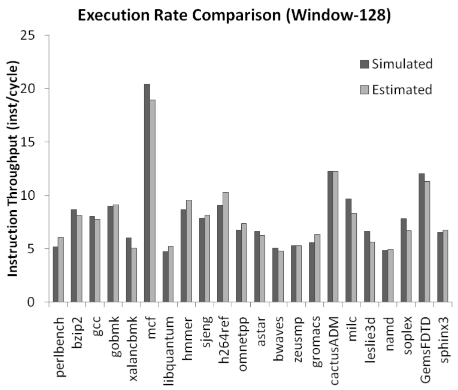
Once the statistics of the *critical dependency chain length* are obtained, the ILP model calculates the average critical dependency chain length L_W for the interested instruction window W. On an idealized machine with unit execution latency, this value is equivalent to the average number of cycles it takes to execute the instructions in the instruction window. Hence, the average throughput could be modeled by W/L_W. However, for a realistic non-unit latency machine, this number should be further divided by the average execution latency Lat_{avg} according to Little's law [14]. Therefore, the average instruction throughput is:

$$\alpha_{avg}(W) = \frac{W}{Lat_{avg} \cdot L_W} \qquad (1)$$

The average latency Lat_{avg} can be derived by weight-averaging the percentage of each instruction type (from instruction mix) with the corresponding execution latency. Note that when calculating the average latency, the latency of the L1 load miss is modeled as the latency of regular functional unit assuming the load never misses L2 cache. The long latency L2 misses will be captured in the MLP model, and treated as the interrupting events that insert intervals in the smooth execution flow. The numbers of L1 and L2 misses are estimated with the stack distance model [20].



(a)



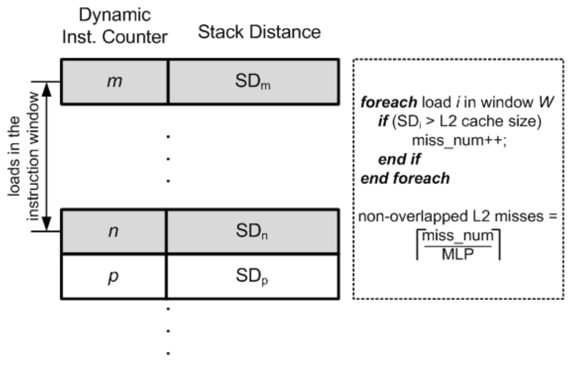
(b)

Figure 2: (a) Comparison of two different profiling schemes for critical paths. The results are based on the profiling of the 22 benchmark programs. (b) The accuracy of execution rate estimation. The simulated machine has perfect branch prediction, perfect memory disambiguation, 32KB L1 data/instruction cache with 10 cycle L1 miss penalty, and infinite L2 cache size.

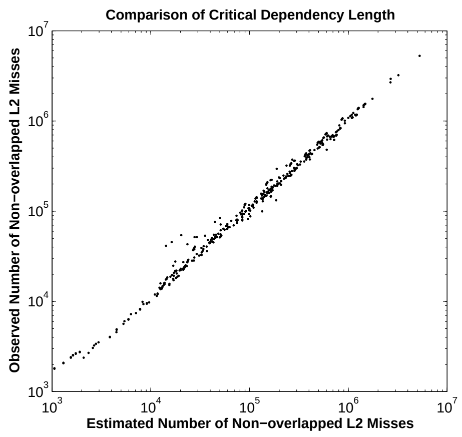
Figure 2(b) shows the accuracy of using equation 1 to estimate the execution rate. The average error between the estimated execution rate and the simulated one is 8.3% when the instruction window size is 128. The error is mainly caused by imperfect representation of program's ILP in the presence of non-unit execution latency. Specifically, when the execution has non-unit latency, the critical dependency chain in terms of the instruction number may be different with the longest dependence chain in terms of the execution cycle, resulting in mismatch between the estimated execution rate and the simulated one.

3.2 MLP Modeling

The modeling of memory level parallelism is based on Mattson's stack distance model [20]. This model exploits the inclusion property of Least Recently Used (LRU) replacement policy (i.e., the content of an N sized cache is a subset of the content of any cache larger than N) and allows us to accurately estimate the number of misses in any fully associative cache. Specifically, when a load/store accesses a data block with stack distance larger than the given cache size, that load/store triggers a cache miss event in a fully associative cache. When it comes to set-associative caches, however, the accuracy of this model slightly decreases mainly because it is unable to capture the conflict misses.


Dynamic Inst. Counter
Stack Distance
loads in the instruction window
m
SDm
n
SDn
p
SDp
foreach load i in window W
if (SDi > L2 cache size)
miss_num++;
end if
end foreach
non-overlapped L2 misses = ⌈miss_num / MLP⌉

(a)



(b)

Figure 3: (a) The estimation of non-overlapped L2 misses in the presence of MLP. (b) The accuracy of the estimated non-overlapped L2 misses. The results are based on the simulation of the 22 benchmark programs.

While the stack distance model is able to estimate the number of misses for a given cache size, it is unaware of MLP, i.e., multiple independent L2 load misses overlapping with each other. These outstanding L2 load misses could drastically change the average load miss penalty and significantly affect the performance, and hence need to be carefully modeled. Prior research obtains the program's MLP information by simulating caches in detail [14][7]. In this paper, we attempt to decouple MLP modeling from detailed cache simulation. To do so, we augment the proposed ILP profiler to generate the maximum number of loads LD_{max} in a dependency chain and the total number of loads LD_{total} in an instruction window. Then, LD_{total}/LD_{max} indicates the average number of loads that could be overlapped with each other in the instruction window. Assuming that the loads in a dependency chain have the same probability of missing L2 cache with other loads, LD_{total}/LD_{max} becomes the average number of the overlapped L2 load misses, or the MLP of the program. Meanwhile, the profiler also generates a load trace that contains the stack distance of a load and the dynamic instruction ID of the corresponding load, as shown in Figure 3(a). The MLP analyzer then walks through the trace, counts the number of L2 load misses that could happen in the instruction window for the given L2 cache size, and calculates the number of non-overlapped L2 misses by dividing the miss number with MLP. The total number of the non-overlapped L2 misses of the program is the sum of the non-overlapped misses in each instruction window:

$$N_{L2}(W,C) = \sum_i \left\lceil \frac{miss\_num(W,C)}{MLP} \right\rceil_i \quad (2)$$

where "$\lceil\ \rceil$" is the ceiling function, $miss\_num(W,C)$ is the number of L2 misses for the instruction window W and L2 cache size C. Figure 3(b) shows the accuracy of this model in estimating the number of non-overlapped L2 misses. The average error between the modeled number of non-overlapped misses and the simulated one is 9.3%, which is reasonably accurate for performance trend modeling.

3.3 Performance Model

The performance model is based on the previously proposed interval analysis [14][6], which treats the exhibited IPC rate as a sustained background execution rate intermittently disrupted by long time miss events, namely, L2 cache misses, branch misprediction, and instruction cache misses. The target of this model is not to accurately predict the absolute performance, but rather to faithfully capture the performance trend as one or multiple resource allocations change.

With ILP and MLP modeling, we are able to obtain the background execution rate, and the number of non-overlapped L2 misses. We can also easily estimate the number of instruction cache misses for any cache size with the stack distance model. However, the instruction cache miss is rare for a reasonable cache size, and its miss penalty is much smaller than that of L2 cache misses. Therefore, in this paper, we do not consider instruction cache misses in our performance model. The remaining part is the number of branch misprediction, which is difficult to accurately estimate purely based on program characteristics. However, the branch transition rate proposed by Haungs et al. [10] contains a clue as to how many branches are hard to predict, and allows us to roughly estimate the number of mispredicted branches. Branch transition rate measures the frequency at which a branch changes direction between taken and not taken. It has been demonstrated that the branches with very low or very high transition rate are easy to predict, and branches with transition rate around 50% are hard to predict. Based on this property, we estimate the number of mispredicted branches by using half the number of branches with transition rate between 0.3 and 0.7. This heuristic essentially assumes that branches with transition rate between 0.3 and 0.7 have 50% prediction rate and all other branches are predicted perfectly. It captures the general trend that the more branches with transition rate near 50%, the more mispredicted branches a program would have. Although this number is only a first-order estimation, it is still reasonable for resource demand estimation as the resource demand estimation is based on performance trend, which is relatively insensitive to the second-order errors.

Hence, our performance model can be built by combining the three major components extracted from the program characteristics, that is, the cycles spent on executing instructions C_{exe}, the cycles spent on accessing the memory C_{mem}, and the cycles spent on serving branch mispredictions C_{br}. Hence, the overall program

Pseudocode 1 Demand on Multiple Resources

```
#define $N$ /*the number of resources that could change simultaneously*/
#define $max\_resource\_array[N]$ /*the array of maximum available resources*/
#define eval_perf($resource\_array$) /*Evaluate the execution time with the resource configuration array $resource\_array$*/
#define est_demand($resource\_array$, $i$, $target\_perf$)
/*Estimate the demand of resource $i$ under the performance target $target\_perf$*/

for ( i=0; $i < N$; i++)
  $base\_demand[i]$ = est_demand($max\_resource\_array$,$i$,$target\_perf$);
  /* estimate the demand for resource $i$ when other resources are set to maximum*/
end for
while( TRUE )
  $perf$ = eval_perf($base\_demand$);
  if( $perf > target\_perf$ )
      set the final demands as the base demand estimates;
      break;
  else
      for ( i=0; $i < N$; i++)
         $temp\_demand[0..N]$ = $base\_demand[0..N]$; /* copy the base resource demand to $temp\_demand$ array */
         $new\_demand[i]$ = est_demand($base\_demand$,$i$,$target\_perf$);
         $temp\_demand[i]$ = $new\_demand[i]$;
         $perf\_gain[i]$ = $perf$ - eval_perf($temp\_demand$);
      /* calculate the performance gain with the newly estimated resource demand */
      end for
  find the index $max\_index$ of the maximum value in array $perf\_gain[N]$;
  $base\_demand[max\_index]$ = $new\_demand[max\_index]$;
end while
```

execution time is:

$$\begin{aligned} Delay &= (C_{exe} + C_{mem} + C_{br})/f \\ &= \frac{N_{inst}}{min(\alpha_{avg}(W), IW) \cdot f} + N_{L2}(W,C) \cdot T_{mem} \\ &+ N_{br} \cdot T_{br} \end{aligned} \quad (3)$$

where N_{inst} is the total number of instructions, N_{br} is the estimated number of mispredicted branches, IW is the instruction issue width, and f is the operating frequency. T_{mem} and T_{br} represent the absolute memory access latency and the absolute time of branch misprediction penalty respectively.

3.4 Resource Demand Analysis

While the performance model allows us to quickly evaluate the performance for a specific resource allocation, it is the resource analyzer that translates the given performance target to a set of resource demands. This subsection presents the details of the resource demand estimation for each type of resources.

3.4.1 Demand on Multiple Resources

In this paper, the estimation of multi-resource demands is built on top of the single-resource demand estimation, which uses the marginal utility to determine the demand on the corresponding resource. The marginal utility originates from economic theory, and is defined as the ratio between the incremental utility over the amount of incremental resource. It has been successfully used as the metric for last-level cache partitioning [23][16]. In this paper, we further extend the application of marginal utility to different hardware resources, and define the marginal utility as follows:

$$MarginalUtility(D_\beta) = \frac{Perf(R_\beta + D_\beta) - Perf(R_\beta)}{D_\beta} \quad (4)$$

where R_β is the amount of resource β, and D_β is the amount of increment in resource β. Note that the maximum marginal utility represents the best (or most efficient) use of a resource increment. Therefore, with marginal utility, we could transform the problem of resource demand estimation to the problem of finding the amount of resource that meets the performance target meanwhile has the maximum marginal utility. Thus, the estimation of the single resource demand becomes straightforward: sweeping the interested resource from its minimum to its maximum while keeping other resources fixed, and searching for the amount of resource that satisfies the performance target and has the largest marginal utility. However, there is an exception: when the performance with the minimum resource allocation is larger than the target performance, the resource demand is set to the minimum value.

While the single-resource demand estimation is straightforward, the estimation of multi-resource demands is non-trivial because the marginal utility is only comparable among the resources with the same type. To address this problem, we propose an algorithm that uses the gradient performance gain to search for the multi-resource demands, as shown in Pseudocode 1. The first step of this algorithm is to estimate the demand on each resource individually when other resources are configured to be the maximum. The estimated single-resource demands are then combined together as the initial multi-resource configuration, which serves as the starting point of an iterative searching process. In each iteration, the algorithm identifies the single-resource demand that has the largest performance gain over the performance of the multi-resource configuration estimated in the previous iteration. This single resource demand is selected to update the multi-resource configuration, and the process continues until the performance meets the target. The complexity of this algorithm is $O(n \cdot k)$, where k is the number of iterations, and n is the number of the changing resources. This algorithm can estimate the multi-resource demands on four types of resources, including ROB size, issue width, L2 cache size, and frequency.

3.4.2 Demand on Memory Bandwidth

The program's demand on memory bandwidth is important for CMP systems, where multiple programs share the limited memory bandwidth resource. It consists of the bandwidth demand on memory read and memory write. Assuming a write-back L2 cache, a read request to the main memory can be triggered by a load/store miss, and a write request can only occur when a dirty cache block is evicted (i.e., cache write-back). While the conventional stack distance model can capture the read traffic to the memory, it is unable to estimate the write-back traffic. To solve this problem, we augment the conventional stack distance model to capture both reads and write-backs to the main memory.

To do so, during stack distance profiling, we associate each cache block with a dirty bit and mark the dirty bit whenever the block has been written to. We then use a *Dirty Stack Histogram* to record the largest stack distance of a dirty cache block. The reason for only considering the largest stack distance is to avoid multiple write-back counts for one store. The details of updating the dirty stack histogram are described in Pseudocode 2. Note that once the dirty bit is set, it will never be reset during profiling. Therefore, the dirty bit is unaware of multiple writes to the same block at different stack distances, hence is unable to capture the situation where one block may miss cache multiple times and generate multiple write-backs. To handle this situation, we also differentiate the dirty block according to whether the block was most recently accessed by a read or a write. Specifically, if the dirty block was most recently accessed by a write, the corresponding counter in the dirty histogram will be incremented regardless of the stack distance. With the dirty histogram, we are able to estimate the number of dirty evictions by using the property of the conventional stack distance model. Specifically, a dirty eviction happens whenever the dirty stack distance of a block is larger than the given cache size.

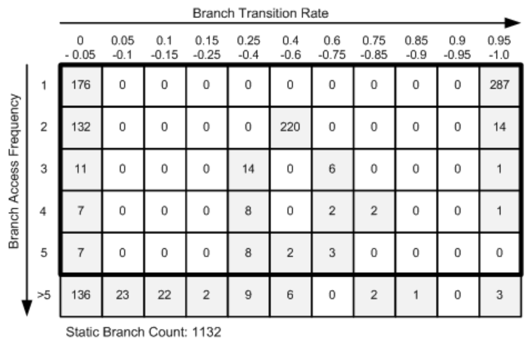
Pseudocode 2 Update of the Dirty Stack Histogram

```
if( $dirty == 1$ )
   if( the block was most recently accessed by a read
   && $stack\_distance > dirty\_stack\_distance$)
         $dirty\_histogram[dirty\_stack\_distance]$- -;
         $dirty\_histogram[stack\_distance]$++;
         $dirty\_stack\_distance = stack\_distance$;
   else if( the block was most recently accessed by a write)
         $dirty\_histogram[stack\_distance]$++;
         $dirty\_stack\_distance = stack\_distance$; end if
end if
if( the current access is a write )
   $dirty = 1$;
   $dirty\_stack\_distance = 0$;
end if
```

3.4.3 Demand on Branch Predictor Size

Branch predictor uses branch history to predict the outcome of a branch instruction before its execution, and usually takes a large fraction of the processor area. Therefore, the program's demand on branch predictor size needs to be identified to prevent unnecessary resource over-provisioning. However, due to the lack of analytical models that can translate predictor size to prediction accuracy, the demand of branch predictor size may have to be estimated by *directly* using the program's branch characteristics. Moreover, since different types of predictors may yield different prediction accuracy levels, the demand on predictor size also needs to be estimated in an *ad hoc* way. Current implementation of PREDA only supports estimating the demand on the size of a two-level PAg predictor [25]. The demand estimation for other branch predictors is in our future work.

PREDA estimates the demand on predictor size based on two branch characteristics: the branch transition rate, and the branch access frequency. As mentioned previously, branch transition rate has its implication on branch history length, which in turn affects the branch predictor size. Branches with very high or very low transition rate are easy to predict and only require short history registers; whereas branches with near 50% transition rate is hard to predict and require long history registers. However, branch transition alone could not tell how often a branch is executed in the dynamic instruction stream. For those branch instructions with very few accesses, they have negligible effect on the overall IPC whether they are predicted correctly or incorrectly, hence should be filtered when determining the demand of branch predictor size. Note that these two branch characteristics are in correspondence with the two-level PAg branch predictor, where the first level table (Per-Address History Table) is essentially a cache holding the frequently accessed branches, and the second level is indexed with history register reflecting the predictability of the branches. As an example, Figure 4(a) shows the branch transition rate distribution as well as branch access frequency distribution of the SPEC CPU program *leslie3d*. The total static branch count is 1132, which seems to indicate that the first-level table should contain 1K entries. However, if we filter out the branch instructions with small access frequencies (less than 5 in this case), the static branch count becomes 204, indicating that 256 entries in the first-level table would be sufficient. This is proved by Figure 4(b), which shows that the prediction accuracy does not degrade until the first-level entry is smaller than 256.

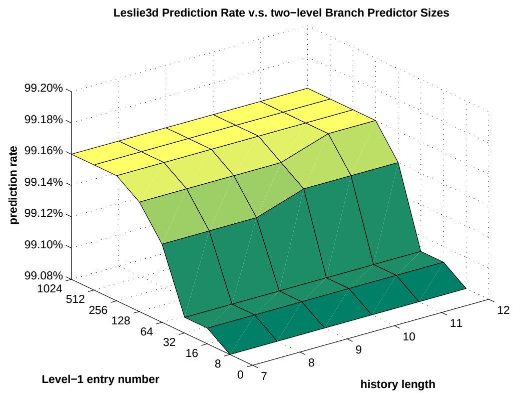


Branch Transition Rate →; Branch Access Frequency ↓

	0 - 0.05	0.05 -0.1	0.1 -0.15	0.15 -0.25	0.25 -0.4	0.4 -0.6	0.6 -0.75	0.75 -0.85	0.85 -0.9	0.9 -0.95	0.95 -1.0
1	176	0	0	0	0	0	0	0	0	0	287
2	132	0	0	0	0	220	0	0	0	0	14
3	11	0	0	0	14	0	6	0	0	0	1
4	7	0	0	0	8	0	2	2	0	0	1
5	7	0	0	0	8	2	3	0	0	0	0
>5	136	23	22	2	9	6	0	2	1	0	3

Static Branch Count: 1132

(a)



(b)

Figure 4: The estimation of branch predictor size demand

Based on this observation, we use the heuristics shown in Pseudocode 3 to estimate the demand on the first level table size and the branch history length. Note that in order to prevent branch filtering from aggressively impacting the prediction accuracy, we ensure that the total number of filtered dynamic branches is less than 0.1% of the total dynamic branches. Note also that the transition rate buckets used in determining the history length are consistent with those used in branch classification by Haungs et al. [10].

4. EXPERIMENT METHODOLOGY

We extensively modified the SimProfile from Simplescalar tool set [2] to profile programs and collect the statistics of the aforementioned program characteristics. We also implemented the PREDA

Pseudocode 3 Demand on Branch Predictor Size

```
#define $access\_threshold$ 16
while( TRUE )
  foreach static branches
    if ( $branch\_access\_frequency < access\_threshold$ )
      $filtered\_static\_branch$ ++;
      $filtered\_dynamic\_branch = filtered\_dynamic\_branch$
      + $branch\_access\_frequency$; end if
  end foreach
  if ($filtered\_dynamic\_branch < 0.001 * total\_dynamic\_branch$)
      break;
  else $access\_threshold$ - -; end if
end while
$first\_level\_entry = total\_static\_branch - filtered\_static\_branch$;
foreach remaining branches
  if $\exists transition\_rate \in [0.4, 0.6]$
      $history\_length = max\_history$;
      /* $max\_history$ is the maximum history length
      specified in the design space */
  else if $\exists transition\_rate \in [0.25, 0.4) \bigcup (0.6, 0.75]$
      $history\_length = (max\_history\text{-}1) > min\_history$ ?
          $max\_history\text{-}1 : min\_history$;
      /* $min\_history$ is the minimum history length
      specified in the design space */
  else if $\exists transition\_rate \in [0.15, 0.25) \bigcup (0.75, 0.85]$
      $history\_length = (max\_history\text{-}2) > min\_history$ ?
          $max\_history\text{-}2 : min\_history$;
  else if $\exists transition\_rate \in [0.1, 0.15) \bigcup (0.85, 0.9]$
      $history\_length = (max\_history\text{-}3) > min\_history$ ?
          $max\_history\text{-}3 : min\_history$;
  else if $\exists transition\_rate \in [0.05, 0.1) \bigcup (0.9, 0.95]$
      $history\_length = (max\_history\text{-}4) > min\_history$ ?
          $max\_history\text{-}4 : min\_history$;
  else if $\exists transition\_rate \in [0, 0.05) \bigcup (0.95, 1.0]$
      $history\_length = (max\_history\text{-}5) > min\_history$ ?
          $max\_history\text{-}5 : min\_history$;
  end if
end foreach
```

kernel with C++ and encapsulate it with the profiler into an integrated framework.

The framework is evaluated on an out-of-order superscalar processor with two-level cache subsystem. The configuration ranges of relevant resources are listed in Table 1. Note that the cache associativity and the block size are constant across all possible L2 cache sizes as we do not explore these aspects in this paper. The number of execution units is chosen such that the overall configuration is balanced. In total, the listed configurations cover over 100K design nodes. When evaluating the estimation of single-resource demand, it is required that other resource configuration are fixed. However, due to the large design space, it is impossible for us to evaluate our framework exhaustively over all configurations. Therefore, we use three representative configuration sets: config-S(mall), config-M(edium), and config-L(arge), as the base configurations to evaluate our resource estimation model. The details of these configuration sets are also shown in Table 1.

Table 1: Configuration Options

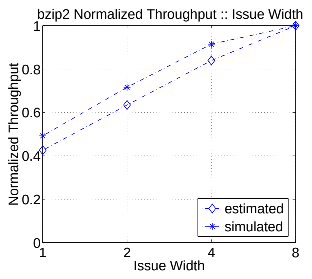
Items	Configuration Options	config-S	config-M	config-L
Issue Width	1 :: 2x :: 8	1	4	8
ROB size	16 :: 2x :: 512	16	128	512
L2 D-Cache	64KB::2x::2048KB	64KB	512KB	2048KB
	8-way associative	8-way	8-way	8-way
	64B	64B	64B	64B
L1 I-cache	32KB	32KB	32KB	32KB
	2-way	2-way	2-way	2-way
	64B	64B	64B	64B
L1 D-cache	32KB	32KB	32KB	32KB
	4-way	4-way	4-way	4-way
	64B	64B	64B	64B
Branch Predictor(PAg)	1st-level: 8::2x::1K	1024	1024	1024
	2nd-level: 128::2x::4K	4096	4096	4096
Clock Freq.	0.5::0.1::2 (GHz)	0.5 GHz	1 GHz	2 GHz

In this paper, we assume the memory access latency to be 200ns, or 200 cycles at the clock frequency of 1 GHz. This latency number in terms of cycles scales proportionally with the operating frequency. The hit latencies of L1 and L2 caches are calibrated against Cacti 5.0 [24] under 90nm technology. The latencies of other execution units are also scaled to 90nm technology. The branch misprediction penalty is set to 20 cycles at 1 GHz. We employ Wattch [4] to collect the performance and power data of the interested processor configurations.

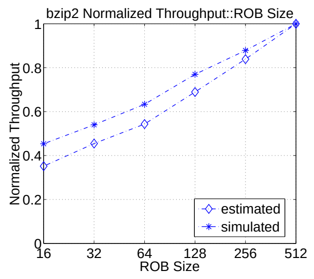
5. EVALUATION

The evaluation of the proposed framework covers three major aspects: the accuracy of the models, the accuracy of resource demand estimation and the computation complexity of the framework.

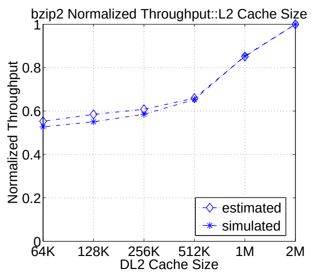
5.1 Model Accuracy



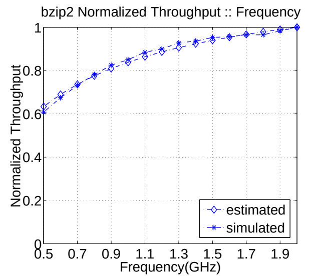
(a)



(b)



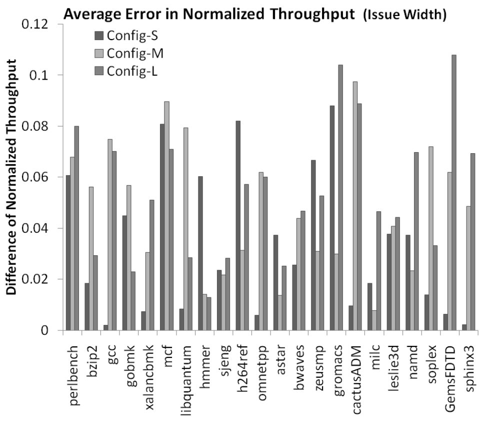
(c)



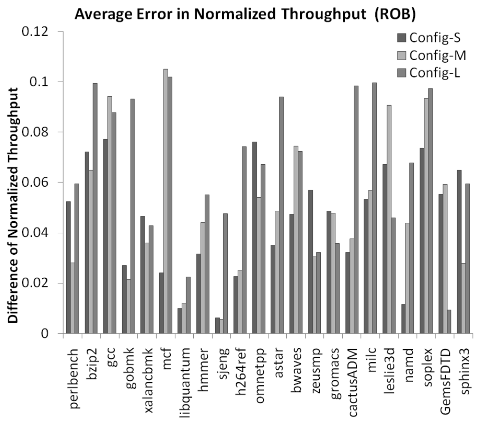
(d)

Figure 5: The comparison of normalized throughput for *bzip2* as one of the resources changes. The configurations of other unchanged resources follow config-M.

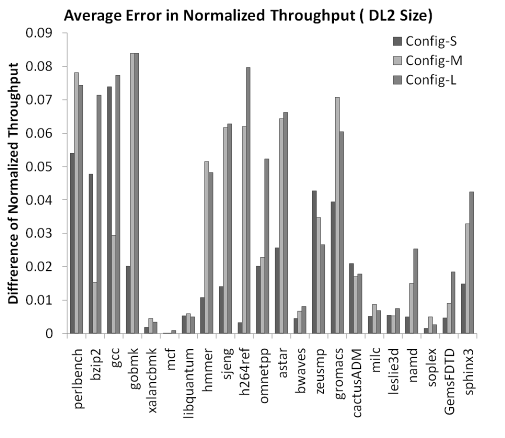
Since the resource demand estimation is based on the relative performance as opposed to the absolute one, we need to validate whether the performance model could accurately capture the performance trend as the resource allocation changes. To do so, we sweep the resource allocation and calculate the corresponding throughput with the performance model, and then normalized them with respect to the largest throughput. The normalized throughput curve is compared against the one obtained from detailed simulation. Figure 5 shows an example of such comparison for *bzip2*. Ideally, these two curves should be overlapped with each other. However, due to the imperfection of the performance model, the estimated performance curve deviates from the simulated one. To measure the difference between these two curves, we calculate the absolute difference of the normalized throughput on each node of the curves, and then calculate the average difference for each curve to evaluate the accuracy of this model. Figure 6 summarizes these differences for each program. Note that most of the programs have a relatively large error in Config-L. This is mainly because some of the second-order effects, such as the branch misprediction caused by specula-



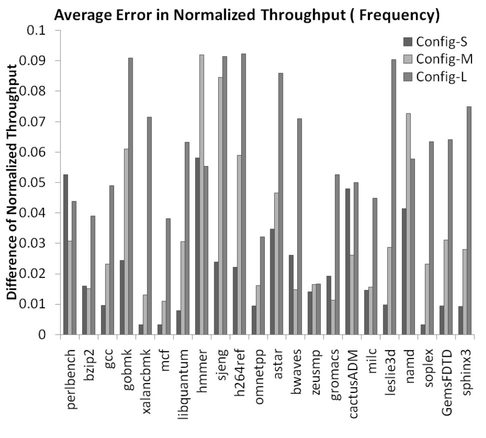
(a)



(b)



(c)



(d)

Figure 6: Average error of the normalized throughput for issue width, ROB size, L2 cache size, and frequency. Each resource estimation was evaluated on three configurations: config-S, config-M, and config-L

tive path information, becomes more outstanding in extremely wide machines; whereas our model only capture the first-order effects. However, even in the worst case, the modeled performance trend on average is only 0.107 or 10.7% off the simulated one, which is still reasonable for resource demand estimation.

The error of the performance model consist of two parts: the intrinsic error, which is the inherent modeling error caused by some simplifying assumptions of the model, and the parameter error, which is the error introduced by the estimation of model parameters using program characteristics. Figure 7 shows the comparison between these two errors. As expected, most programs have much smaller intrinsic error than the combined one, especially for *gcc* and *namd*. However, some programs see a slightly higher intrinsic error than the combined error. This is because the parameter error and the intrinsic error may be canceling each other, leading to a smaller combined error. In worst case, the average intrinsic error is 0.076 or 7.6% in terms of the normalized throughput (*mcf* in Figure 7(a)).

5.2 Accuracy of Resource Demand Estimation

5.2.1 *Single-Resource Demand Estimation*

We evaluate the estimation of single-resource demand on issue width, ROB size, L2 cache size, and frequency at 20 different performance target levels, ranging from 0 to 95% with a step of 5%. Figure 8 shows the comparison between the demand estimated with our performance model and the one obtained from detailed simulation for program *bzip2*. Because of the imperfection in performance modeling, there are differences between the estimated and the simulated demands at certain performance targets. The average amount of these differences across the entire 20 performance target levels reflects the accuracy of the demand estimation, as shown in Figure 9. We observe that the demand difference at any performance target level is no larger than 4 configuration units. The largest demand difference happens in estimating the frequency demand, and this difference is still reasonable considering there are 16 different configuration options for frequency demand.

To evaluate the estimation of memory bandwidth demand, we compare estimated memory bandwidth with the simulated one at each 100K instruction interval, and accumulate the absolute difference between these two to obtain the overall memory bandwidth estimation error. Figure 10 shows the error rates of bandwidth demand estimation for both memory read and memory write traffics at three different configurations. On average, the total memory bandwidth estimation error increases from 4.76% to 6.26% as the L2 cache size changes from 64KB to 2MB. This is mainly because as the cache size increases, memory traffics become smaller and hence the bandwidth caused by conflict L2 misses, which are not captured in our stack distance model, becomes more significant.

To evaluate the estimation for the demand of branch predictor size, we compare the size and the prediction accuracy of the estimated branch predictor configuration with that of the largest pre-

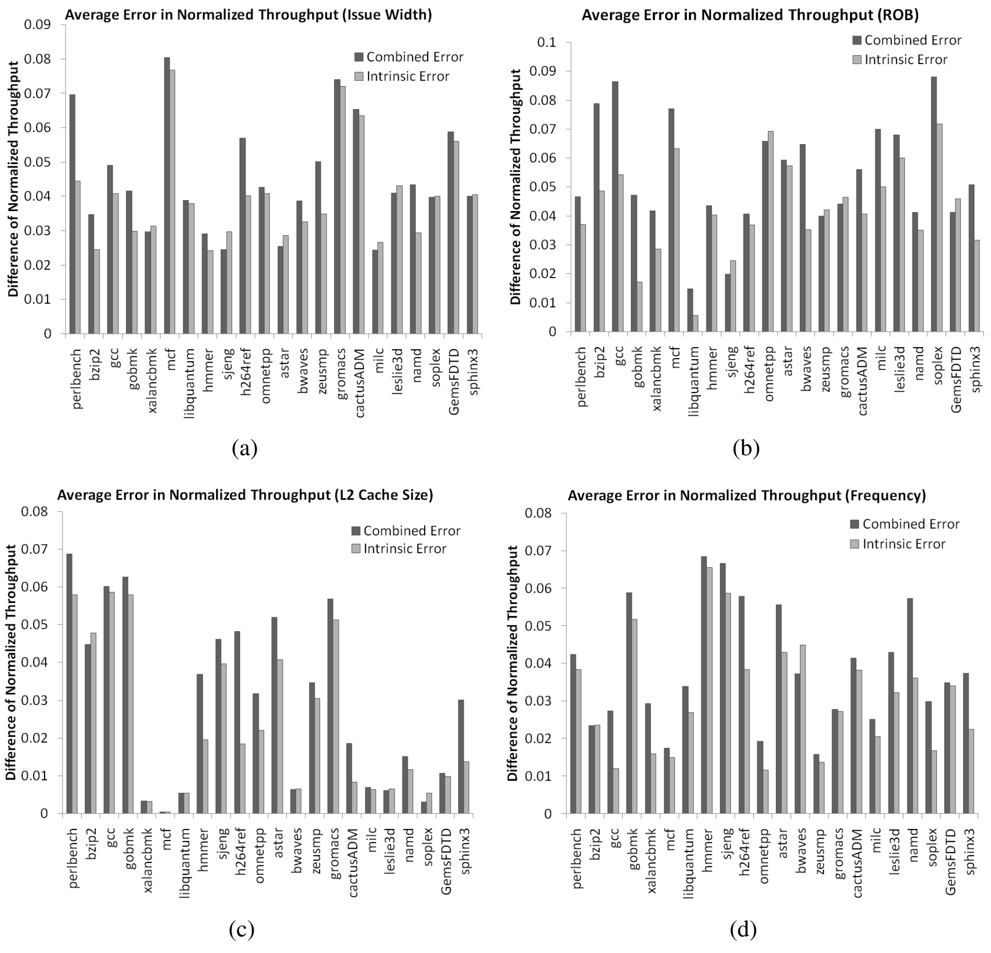


Figure 7: Comparison of the combined error and the intrinsic error in normalized throughput.The intrinsic error is obtained by using the simulated values of the non-overlapped L2 misses and the branch mispredictions in the performance model. The errors are averaged across three configurations: config-S, config-M, config-L.

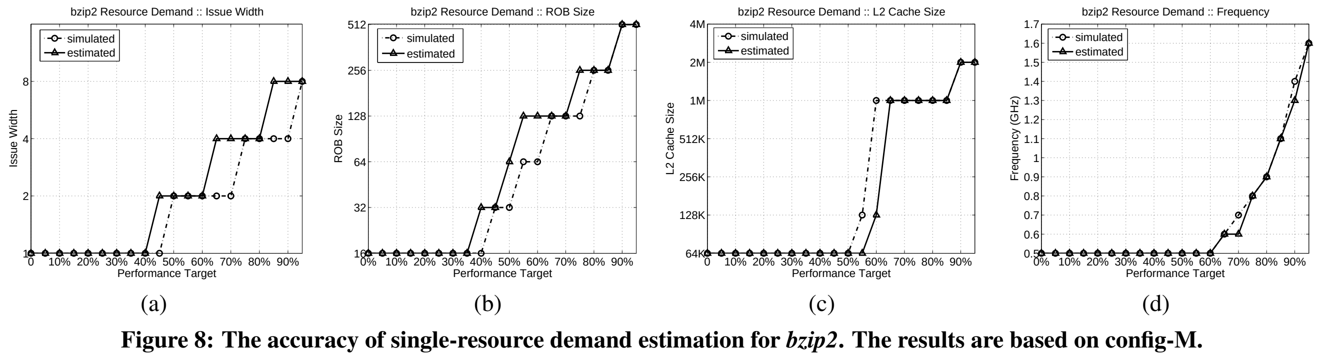


Figure 8: The accuracy of single-resource demand estimation for *bzip2*. The results are based on config-M.

dictor in the configuration range. The results are listed in the table 2. On average, by using the estimated predictor size, we could achieve 40.3% reduction in area with only 0.12% accuracy loss over the largest branch predictor. Overall, the proposed heuristic captures the demand on branch predictor size very well.

5.2.2 *Multi-Resource Demand Estimation*

The quality of multi-resource demand estimation includes two aspects: the accuracy in satisfying the performance target and the energy efficiency of the estimated resources.

To evaluate the accuracy, we used the proposed framework to estimate the resource demands at each performance target ranging from 50% to 95% with the step of 5%, and then performed detailed simulations with the estimated resource configurations for each performance target. The obtained relative performance (normalized to the largest performance in the design space) is compared against the corresponding performance target. The differences are summarized in Figure 11(a). The observed error is up to 12.7% (on *soplex*), and the maximum average error is 8.6% (on *xlanacbmk*). Note that we only report the results with performance target larger than 50% to avoid the *ill-suited* cases that some programs may have a small per-

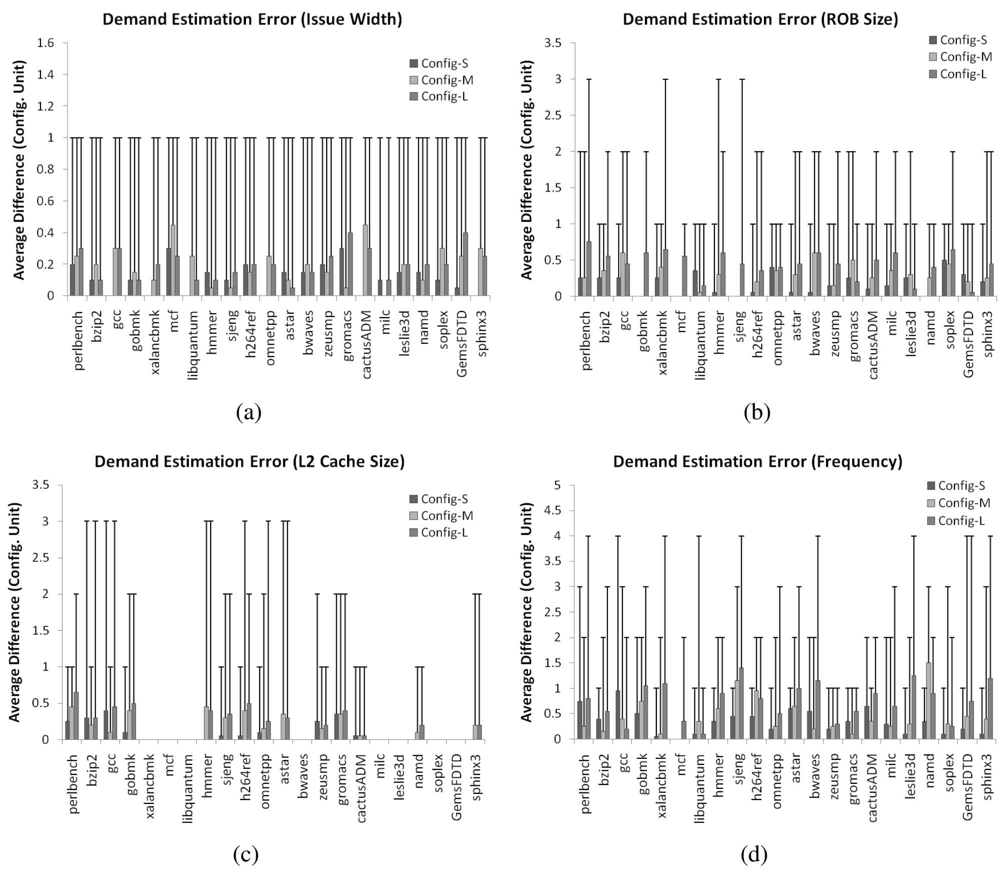


Figure 9: The error of single-resource estimation. Config unit refers to the quantization of each resource shown in Table 1. The error bar represents the largest error in demand estimation for the corresponding program.

Table 2: The Demand Estimation for Branch Predictor Size

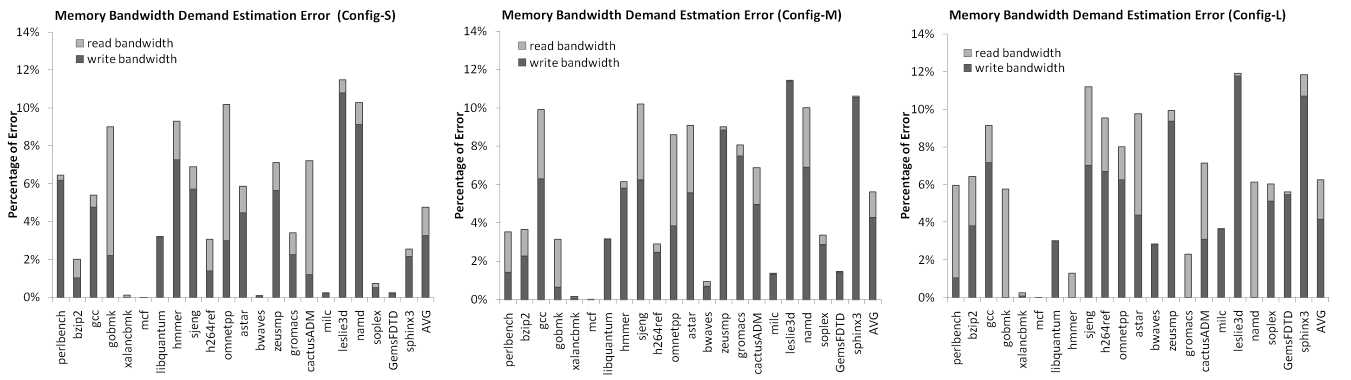
Benchmarks	Size Demand		Size Reduction	Accuracy Loss
	L1 entry	History bit		
perlbench	1024	12	0	0
bzip2	128	12	52.5%	0.26%
gcc	1024	12	0	0
gobmk	1024	12	0	0
xalancbmk	1024	12	0	0
mcf	512	12	30%	0.06%
libquantum	8	12	59.5%	0.03%
hmmer	128	12	52.5%	0.09%
sjeng	1024	12	0	0
h264ref	1024	12	0	0
omnetpp	1024	12	0	0
astar	64	12	56.3%	0
bwaves	32	12	58.1%	2.1%
zeusmp	64	9	92.2%	0.05%
gromacs	8	7	98.5%	0
cactusADM	16	7	98.2%	0
milc	32	11	78.3%	0
leslie3d	256	12	45.0%	0
namd	128	12	52.5%	0.1%
soplex	256	12	45.0%	0.01%
GemsFDTD	16	7	98.2%	0.01%
sphinx3	1024	12	0	0
avg	-		40.3%	0.12%

formance variation range and its smallest relative performance may be much larger than the performance target.

To evaluate the energy efficiency, we compare the energy consumption of the estimated multi-resource demand with the energy consumption of other resource combinations that satisfy the given performance target. Due to the large design space, it is prohibitively expensive to exhaustively compare the estimated resource configurations with every eligible design node. Therefore, we use Monte Carlo simulations to simulate 300 random samples in the design space, and group them into the buckets of (0,0.05],(0.05,0.1],..,(0.95,1] according to their performance relative to the highest one in the design space. Within each bucket, we divide the energy of the estimated multi-resource configuration with the maximum energy of the design nodes in that bucket. These ratios indicate the energy efficiency of the estimated resource demands, and are summarized in Figure 11(b). On average, the ratio is no larger than 86.5%, and can be as low as 44.4%, which means the estimated multi-resources reasonably satisfy the energy efficiency requirement in the definition of resource demand.

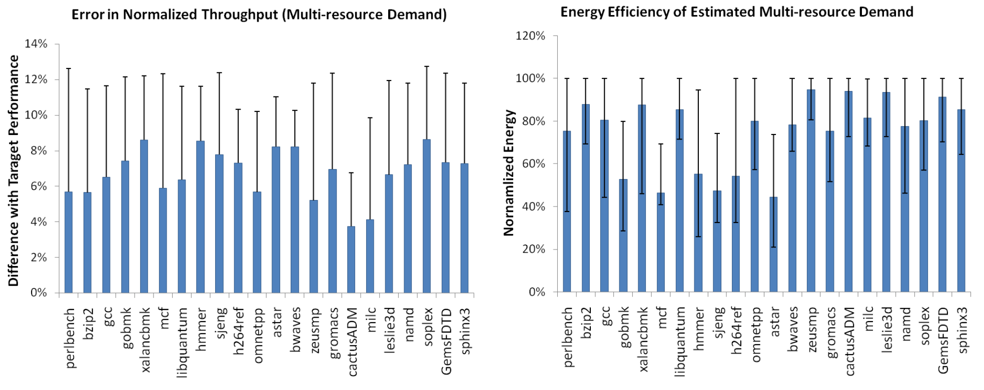
5.3 Complexity Analysis

The complexity of the PREDA framework involves the complexity of multi-resource demand searching algorithm and the time cost in evaluating the performance model. As explained previously, the complexity of the algorithm depends on the number of iterations required to reach the target performance. To reduce the number of iterations, the algorithm hoists the starting point of the searching process as the target performance increases. This feature allows the algorithm to avoid unnecessary search iterations and significantly speeds up the searching process. In our experiment, the algorithm converges in no larger than 12 iterations. We also compare the CPU time required to finish *one searching iteration* by using our performance model with the time required by using the state-of-the-art analytical model developed by Eyerman et al. [7], and we observe up to 40X speedup with our proposed model. This is mainly be-



(a) (b) (c)

Figure 10: The memory bandwidth estimation error



(a) (b)

Figure 11: Evaluation of multi-resource demand estimation. The results are based on estimating 4 different resource demands.

cause every time cache size changes, Eyerman's model requires detailed cache simulation to collect cache miss and MLP information for different window sizes, whereas our model only needs to walk through the stack distance trace. Depending on the data footprint of the programs, the profiling time cost of PREDA may be larger than Eyerman's model because of the stack distance profiling. However, this is one-time profiling cost, and could be easily amortized by the speedup in the demand estimation process.

6. RELATED WORK

Our work is most relevant to the predictive resource management framework proposed by Narayanan and Satyanarayanan [21]. However, their framework can only estimate the coarse-grain resource demands, such as CPU cycles and memory sizes; whereas our work estimates resource demands at much finer granularity, and can be applied in the areas that require fine-grain resource tuning. PUNCH proposed by Kapadia et al. [13] also shares some common grounds with our framework as both attempt to predict the resource demands by using application-specific parameters. But again, in their work, the resource demands are limited to CPU time only.

Our work is also closely related with performance modeling, which usually employs analytical models, regression models, or predictive models. The analytical model is typically based on *interval analysis*, which was used by Karkhanis and Smith for their first-order superscalar processor model [14]. They further leveraged this model to automatically explore the design space for the Pareto-optimal design parameters [15]. Recently, this model was improved by Eyerman et al. for a higher accuracy in performance modeling [7]. However, all of these models rely on detailed simulation of some components, such as caches and branch predictors, to obtain key statistics of the program-microarchitecture interactions. The requirement for partial simulation not only costs time in off-line performance modeling, but also implies that it has to follow the trial-and-error scheme when applying this model for on-line resource management. However, our approach focuses on modeling the performance trend rather than the absolute performance value, and avoids any detailed simulation of any resource component. The decoupling from detailed simulations not only ensures fast off-line resource demand estimation, but also allows this model to be applied in on-line resource management without trial runs.

Both regression models and predictive models are essentially empirical models, which hide the details of program-hardware interactions by fitting high-level equations with the simulated results. The regression model has been applied in estimating the significance of the design parameters and their interactions [12], exploring the design space [18] as well as analyzing the microarchitectural adaptivity [19]. An artificial neural network (ANN) based predictive model was also proposed by Ipek et al. for performance prediction [11]. While the empirical models are relatively simple, they require time consuming training on a per-application basis before they can reasonably model performance. The requirement for training fundamentally limits these models from being applied online. In contrast, our model is based on the analysis of program inherent characteristics and does not require any training.

Besides off-line performance modeling, some on-line resource management techniques have been proposed recently. Qureshi et al. proposed the cache utility monitor (UMON) to estimate the utility of assigning additional cache ways to an application [23]. Kaseridis et al. extended this on-line cache monitor for system-level memory bandwidth management [16]. While these works address single resource management, Bitirgen et al. attempted to manage multiple resources by using on-line machine learning techniques [3]. However, the on-line machine learning model requires periodic training and is expensive to implement and hard to validate. In contrast, our model does not require any training and could be applied on-line for both single or multiple resource management with some hardware support for on-line profiling .

7. CONCLUSIONS

As the applications in computer systems become increasingly diversified, it is important to efficiently identify the hardware resource demands of the applications so that the hardware substrate could be tailored to the needs of the applications for power efficient computing. Existing models are inappropriate for estimating resource demands as they require either partial simulations or time-consuming training. In this paper, we present an integrated framework for program resource demand analysis (PREDA), which leverages the synergy between the performance trend modeling and marginal utility to identify the resource demand of a workload without any detailed simulation. The proposed framework is able to estimate both single-resource and multi-resource demand on an array of processor resources, ranging from the issue width, the operating frequency to the memory bandwidth. Experimental results show that the proposed framework on average provides no larger than 8.6% error to any given performance target for multi-resource demand estimation. By using the proposed performance model, the framework achieves up to 40X speedup in multi-resource demand estimation compared with that by using state-of-the-art analytical model. The proposed framework is useful for workload capacity planning in computing systems, early stage design space exploration, as well as coordinated multiple resource management for Quality-of-Service in CMP systems.

8. ACKNOWLEDGMENTS

The authors would like to thank Giuliano Casale and other 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.

9. REFERENCES

[1] SPEC cpu2006 benchmark suite. In *http://www.spec.org*.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35:59–67, February 2002.

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

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00*, pages 83–94, 2000.

[5] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02*, pages 233–244, 2002.

[6] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate cpi components. In *ASPLOS-XII*, pages 175–184, 2006.

[7] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 27(2):1–37, 2009.

[8] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO'40*, pages 343–355, 2007.

[9] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. In *Journal of Instruction Level Parallelism*, volume 7, pages 1–28, 2005.

[10] M. Haungs, P. Sallee, and M. Farrens. Branch transition rate: a new metric for improved branch classification analysis. In *HPCA'00*, pages 241 –250, 2000.

[11] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS-XII*, pages 195–206, 2006.

[12] P. Joseph, K. Vaswani, and M. Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. In *HPCA'06*, pages 99 – 108, 2006.

[13] N. Kapadia, J. Fortes, and C. Brodley. Predictive application-performance modeling in a computational grid environment. In *Proceedings of The Eighth International Symposium on High Performance Distributed Computing*, pages 47 –54, 1999.

[14] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA'04*, pages 338–349, 2004.

[15] T. S. Karkhanis and J. E. Smith. Automated design of application specific superscalar processors: an analytical approach. In *ISCA'07*, pages 402–411, 2007.

[16] D. Kaseridis, J. Stuecheli, J. Chen, and L. John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems. In *HPCA'10*, pages 1–11, 2010.

[17] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06*, pages 23–32, 2006.

[18] B. Lee and D. Brooks. Illustrative design space studies with microarchitectural regression models. In *HPCA'07*, pages 340 –351, 2007.

[19] B. C. Lee and D. Brooks. Efficiency trends and limits from comprehensive microarchitectural adaptivity. In *ASPLOS XIII*, pages 36–47, 2008.

[20] R. L. Mattson, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, 1970.

[21] D. Narayanan and M. Satyanarayanan. Predictive resource management for wearable computing. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, MobiSys '03, pages 113–128. ACM, 2003.

[22] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07*, pages 57–68, 2007.

[23] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO'06*, pages 423–432, 2006.

[24] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. *HP Technical Reports*, 2008.

[25] T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *ISCA '93*, pages 257–266, 1993.