A PRACTICAL METHOD FOR ESTIMATING PERFORMANCE DEGRADATION ON MULTICORE PROCESSORS, AND ITS APPLICATION TO HPC WORKLOADS

by

Tyler Dwyer

B.Sc., Queen's University, 2009

A Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in the
School of Computing Science
Faculty of Science

SIMON FRASER UNIVERSITY
Fall 2012

APPROVAL

Name: Tyler Dwyer

Degree: Master of Science

Title of Thesis: A Practical Method for Estimating Performance Degradation on Multicore Processors, and its Application to HPC Workloads

Examining Committee: Dr. Arrvindh Shriraman
Chair

Dr. Alexandra Fedorova, Senior Supervisor

Dr. Jian Pei, Co-Supervisor

Dr. Greg Mori, SFU Examiner

Date Approved: _____

Abstract

When multiple threads or processes run on a multicore CPU they compete for shared resources, such as caches and memory controllers, and can suffer performance degradation as high as 200%. We design and evaluate a new machine learning model that estimates this degradation online, on previously unseen workloads, and without perturbing the execution.

The motivation for this thesis is to help data center and HPC cluster operators effectively use workload consolidation. Data center consolidation is about placing many applications on the same server to maximize hardware utilization. In HPC clusters, processes of the same distributed applications run on the same machine. Consolidation improves hardware utilization, but may sacrifice performance as processes compete for resources. Our model helps determine when consolidation is overly harmful to performance. Our work is the first to apply machine learning to this problem domain, and we report on our experience reaping the advantages of machine learning while navigating around its limitations. We demonstrate how the model can be used to improve performance fidelity and save energy for HPC workloads.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Workload consolidation refers to a resource allocation principle where we try to place many applications on the same server, so as to not leave any cores idle. Consolidation makes a fundamental trade-off between performance and hardware utilization (and thus power efficiency). We improve utilization of the machine's resources, but sacrifice some amount of performance, because increased resource sharing among threads may reduce the rate of retired instructions.

Previous studies have shown that performance degradation occurring when threads or processes run on the same multicore CPU and share resources, such as last-level caches, memory controllers, system request queues, and pre-fetch bandwidth, can reach as much as 200%, relative to running in isolation [6, 10, 13, 15, 22]. Such severe degradation can defeat the benefits of consolidation, leading to violation of customer QoS constraints or simply producing application slowdowns that are deemed unreasonable [8, 17]. In certain cases, the slowdown could be so extreme, that despite saving power we waste *energy*, because the workload takes much longer to complete under consolidation than without it [5].

It is clear that we cannot use workload consolidation without considering its potentially damaging effects. Unfortunately, conventional performance tools, while allowing us to observe events like retired instructions or cache and memory controller accesses, do not tell us how much performance degradation the workload is experiencing. If we have the luxury of knowing in advance the input used by our workload and have access to the hardware on which it will run in the field, we can execute the workload offline with various degrees of consolidation, determine the optimal, and use that setting in the field. Unfortunately, in most cases we do not have advance knowledge of runtime parameters, and so deciding

whether or not to consolidate becomes guesswork.

In this work we propose a solution whose goal is to enable intelligent consolidation decisions in data centers. We develop a model that takes as inputs performance counter values obtained on a consolidated workload (which can be measured inexpensively online) and produces an estimate of how much performance degradation each thread or process is suffering relative to running on the CPU alone, without contention for resources. We do not require running applications in isolation or perturb their execution. Using this estimate, the operator can decide if the degradation is beyond an acceptable threshold, and if it is, roll back the consolidation, distributing the workload to a larger number of CPUs or servers. The model is designed for scenarios where either different applications are consolidated on the same machine, or processes of the same application share the hardware (as commonly happens in a HPC setting).

We create the model using machine learning; to the best of our knowledge, our work is the first to apply machine learning to this incredibly complex problem. We train the model offline, using a set of widely available benchmark programs. To maximize accuracy, the training programs should have properties similar to the workload on which the model will be used, but they need not be the same or overlap. For example, if we are targeting business applications, we could train on SPEC JBB or TPC-W; for scientific workloads, we could train on SPEC CPU. The model is trained for specific hardware, but this needs to be done only once, and can be performed as part of system installation and configuration.

Although machine learning has been used to model system behaviour in the past, it has not been applied to performance degradation on multicore systems. We were compelled to try machine learning, because it could help overcome practical limitations of previous solutions. Previous solutions used analytical modeling, heuristics based on hardware counters, or trial-and-error methods. Analytical modeling is extremely fragile and challenging, because modern CPUs are incredibly complex; moreover, crucial details about the microarchiecture are often unavailable due to intellectual property protection. Heuristic-based models provide only a coarse approximation of performance degradation; typically they can tell us whether the degradation is occurring, but not its magnitude. Indeed, as we show in chapter 4, a cluster scheduler based on a heuristic model can waste energy relative to the scheduler that is based on the more precise model proposed here. Trial-and-error methods require running applications in different co-schedules with other applications [18] or with dummy workloads [8] in an effort to find a co-schedule that minimizes the degradation, but as the

number of cores increases the number of co-schedules grows as well, as does the perturbation imposed on the workload. In considering the limitations of previous approaches, machine learning looked like a promising alternative.

Machine learning shines in its ability to discover complex relationships between a variety of factors and filter out the factors that are not important to the model. This seemed very well suited to our problem, because modern CPUs allow monitoring hundreds of performance events, some of which correlate with sharing-induced degradation, but it is very difficult to filter the spurious events manually. Our idea was to get machine learning to discover the relevant hardware events automatically, so we could apply the same methodology on *any* hardware platform, to automatically create a model for the desired processor without the manual labour of picking the right counters. Indeed, we observed that our model-building methodology seamlessly ported between microarchitectures: we created and evaluated a model on an Intel Xeon and an AMD Opteron processor with equal success.

The key limitation of machine learning is that the model is only as good as the training data. While we can make an effort to train on the workloads similar to those used in the field, it is not always possible. In evaluating the model, we observed that if the input data falls outside the range of the values seen in training, the model becomes limited in its ability to make accurate predictions. While it is possible to re-train the model using the "outlier" workload, it is crucial to *detect* in real-time when the model is about to produce a poor estimate. To that end, we propose a statistical solution, called *confidence predictor*, which detects when the model is about to make a mistake. Upon detecting the "low confidence" signal, the operator can conservatively decide to not consolidate this workload, or re-train the model using the data obtained on the "outlier" workload. The confidence predictor reduces the maximum error by a factor of 2× or 3× depending on the system, while marking roughly 25% of the predictions as "non-confident".

We focus on modeling the degradation resulting from sharing the multicore chip's resources, because they are the most difficult to control in software, unlike CPU cycles, memory space and disk/network bandwidth, which can be controlled by quotas. In fact, in the environment where we chose to evaluate our model, an HPC cluster running scientific workloads, each thread is typically given a dedicated CPU, and a job's memory is sized to fit within the physical memory limits of the machine. Network bandwidth on the cluster interconnect was not a bottleneck in our experiments, but if it were, machine learning could also be used to model contention for that resource.

The contributions of our work are: (1) creating the methodology for modeling performance degradation on multicore systems using machine learning, (2) evaluating the strengths and limitations of the resulting model, (3) designing a confidence predictor that signals when the model is unable to produce an accurate estimate, and (4) demonstrating how the model can be applied to improve performance fidelity and save power in an HPC-like setting.

The rest of the thesis is organized as follows: Chapter 2 describes the methodology for building the model. Chapter 3 evaluates its accuracy. Chapter 4 presents and evaluates a simple scheduler for HPC clusters that uses our model. Chapter 5 discusses related work. Chapter 6 summarizes our findings.

Chapter 2

Methodology

2.1 The Model

Before we describe the methodology for creating the model, we explain how the model would be used in practice. We begin with the assumption that our target application is a single-threaded process. Then we explain how the model would work with multi-threaded processes.

First, the user would train the model following the procedure described below, and using as the training set the applications that most closely resemble those on which the model would be used in the field. Each application is executed alone on the machine and in combination with other applications. We compute how much slower the application runs when co-scheduled with others relative to running alone; this slowdown is the performance degradation. We then train the model to predict performance degradation using hardware counter values obtained on the system when the target application runs with other jobs. The resulting model is thus set up to estimate how much slowdown the application is suffering when space-sharing the CPU with others, without the need to execute the target application alone.

The model estimates the degradation based on per-core and system-wide hardware counter values, and so it is agnostic to whether the cores are running single-threaded processes or threads from the same application. We purposefully do not configure the model to account for positive effects of co-operative sharing; such models are available and, if desired, can be used in conjunction with the proposed model [11].

The goal of the model is not to facilitate contention-aware scheduling of threads within

an application, but to decide whether we need to allocate more hardware in environments where many applications runs on the same physical server. So if we have a multi-threaded application sharing hardware with other processes, we would use the model to estimate, for each thread, the performance degradation that the thread is suffering under resource contention relative to running alone and then average the degradations of all threads to obtain the degradation for the entire application. If the application is deemed to suffer unreasonable performance penalty, we would migrate that application to a less loaded server, to create a less contentious environment. Within each server, a local OS or hypervisor scheduler, ideally one that takes into account both resource contention [22] and co-operative resource sharing [11,20] will decide how to place threads on cores. Other resource-allocation decisions, e.g., regarding CPU and memory quotas, can be applied on top; deciding how to combine allocation of resources of different types is deferred to future work.

In the rest of the chapter we explain how we build the model. As will become clear in Section 2.1.2, our model is a decision tree. A decision tree consists of nodes and branches, where each node is labeled with an attribute (e.g., a hardware counter type in our model) and a threshold for the attribute's value. Based on the thresholds we decide which of the branches emanating from the node to follow. We follow the tree all the way down to a leaf, comparing the measured hardware counter values with the corresponding thresholds assigned to the nodes. The resulting leaf node will give us the predicted degradation for the data point characterized by these hardware counter values. So the goal of building the model is to assign the right attribute thresholds to intermediate tree nodes and predicted degradation values to the leaves, so we arrive at a reasonably accurate prediction of performance degradation. This process consists of three steps: *(1) collection of the training data, (2) attribute selection, (3) model training.*

2.1.1 Collection of the training data

Testing Platform

To confirm portability of our methodology, we built and tested our model on two systems, *Intel* and *AMD* using exactly the same procedure. Refer to Table 2.1 for system parameters. We trained and tested the model only on a single socket, because additional contention from running applications on the second socket did not significantly affect degradation. The (*AMD*) system has a NUMA (non-uniform memory access) architecture, and so we ensured

	Intel: 2-socket "Clovertown"	**AMD: 2-socket "Istanbul"**
Cores per socket	4	6
Shared per socket	Two L2 caches (per pair of cores), front-side bus, pre-fetcher, memory controller	L3 cache (all cores), system request queue, memory controller, data and memory controller pre-fetchers

Table 2.1: Shared resources in the experimental systems

that an application's memory is allocated on the same node as where the application runs. This is how the operating system would typically behave. [1]

Applications and Co-schedules

Since our goal was to evaluate the model on scientific applications typical of HPC clusters, we trained the model on the SPEC CPU2006 suite.

We ran applications in three different types of co-schedules. The *solo run* is when an application runs alone on a system without contention. The solo run was used in calculating the true value of performance degradation. True degradation is needed only to train the model and to evaluate the accuracy of predictions; we do not expect to know it on a production system. In the other two types of co-schedules the application runs in contention, concurrently with others. In the *clean* co-schedule, the primary application (the one whose degradation we predict) runs with several copies of itself (three on the *Intel* system, five on the *AMD* system). In the *random* co-schedule, the primary runs with randomly chosen benchmarks on the remaining cores.

To collect the data for training, we start all applications in the co-schedule at the same time. If any interfering application terminates before the primary is finished, we restart the interfering application, thus keeping the primary in full contention for the entire run. Overall, we recorded over 10 random co-schedules for each of the 27 applications. Together with clean and solo runs, this resulted in over 500 co-schedules.

[1] On most recent NUMA systems with directory-based cache coherence protocol, processes running on separate memory nodes and sourcing data from their local memory node will not noticeably affect each others performance

Recording Performance Events

To record the attributes relevant for modeling degradation, we use the hardware performance counters that can be used to measure events, such as last-level cache misses, the number of bus transactions, etc. (some shown in Table 3.1). On the *Intel* system there are 340 event counters per core, but only four hardware registers for counting them. To be able to record all these events, we had to sample them by switching between different event types.

When we switch between events, we have to make sure that each event is sampled for a substantial period of time, to ensure good sampling accuracy. The need to measure a large number of events puts a lower bound on the interval of execution for which we are able to record all available counters. We set that interval to 5 billion retired instructions, which allowed us to capture the required events without major loss of precision [2] Each 5-billion instruction window is called an *execution instance*. We train the model and produce predictions for execution instances, as opposed to the entire program. The implication is that in production setting, we need to sample event counters for 5 billion instructions (a few seconds of execution time) before we are able to produce an estimate of the degradation. Therefore, the model is best suited for long-running workloads, such as the scientific applications we evaluate in our study.

As we rapidly switch between different types of counters, intermittent system events, such as handling of an interrupt, can introduce unexpected spikes or dips in the measurements. Data containing these variations presents a challenge for a machine learning algorithm, because there is not enough training data to learn the behaviour during these extraneous events. To smooth out their influence, we found it helpful to represent attribute values for each instance as the rolling average of the past five instances.

Calculating performance degradation

The degradation for an instance is obtained by calculating the percent increase in clock cycles needed to complete the fixed instruction window under contention, relative to solo. For example if the co-schedule *A*-*B*-*C*-*D* has 50 execution instances, then the *A* solo run would also have 50 instances, as instances are based upon retired instructions, which are constant regardless of contention. The clock cycles of the ith instance of *A*-*B*-*C*-*D*, denoted

[2] Adding the attribute selection step, described below, enabled us to use a smaller execution interval. We observed, however, that the accuracy of the model was not sensitive to the size of that interval.

$ABCD^i_{clk}$, are compared with the ith instance of A's solo run, denoted as A^i_{clk}. From these, the degradation of A while in contention with B-C-D is computed as:

$$Deg(A^i_{BCD}) = \frac{ABCD^i_{clk} - A^i_{clk}}{A^i_{clk}} * 100\% \tag{2.1}$$

We performed the above calculation on all instances in our data set. After this procedure our dataset contains equally-sized execution instances, each with $340 \times 4 = 1360$ attributes from the event counters[3], and the degradation value.

2.1.2 Attribute Selection and Model Training

Attribute Selection

Before building the model we reduced the number of attributes in the dataset from 340 per core to 19 per core[4]. Attribute selection was performed for three reasons. The first is to eliminate the attributes that were redundant or unrelated to degradation. The second is to reduce the training time from up to several hours (with 340 attributes) to several minutes (with 19 attributes). Third is to allow a new dataset to be recorded with fewer events sets, leading to more accurate recording.

We performed attribute selection using a suite of machine learning algorithms, *Weka*. We tested several attribute selection techniques by creating models for each set of selected attributes and comparing their accuracy. The technique with the lowest error rate was *correlation based feature subset attribute selection (CfsSubset)* [9]. *CfsSubset* sorts the attributes by their correlation to the class attribute (degradation) and to the other attributes in the dataset. Attributes with a high correlation to the class attribute and a low correlation to the other attributes are considered relevant.

Attribute selection yielded 19 attributes for each core, shown in Table 3.1; the same 19 attributes were selected for each core. In order to train the model, we want to distinguish the counters of the *target* core (the one whose degradation we are predicting) from those of the *interfering* cores. To do that, for each event counter we average the values obtained on all interfering cores, for a given execution instance. In summary, before we train the model, we have thousands of execution instances, and for each we have the values of the

[3]This is for the Intel architecture, AMD had 1471 event counters initially recorded

[4]AMD's counters were reduced to 32 core counters and 8 chip counters

Bagged REPTree	**+0.00%**	Linear Regression	+4.45%
Gaussian Process	+1.20%	PLS Classifier	+4.76%
REP Tree	+2.35%	Decision Table	+6.12%
Isotonic Regression	+3.79%	Simple Linear Regression	+8.10%
SVM Reg	+3.84%	Neural Network	+10.60%
SMO Reg	+3.85%	Conjunctive Rule	+11.17%
M5P	+4.14%	Decision Stump	+12.18%
Pace Regression	+4.36%	M5Rules	+17.84%

Table 2.2: Modelling techniques tested and the average error they produced relative to REPTree on a reduced training set. REPTree's average error was 16% in cross-validation.

event counters on the target core, the average of the counters on the interfering cores, and the degradation for the target core.

Model Creation

In the process of building the model we evaluated most modeling algorithms available in *Weka*, listed in Table 2.2. We used on *REPTree* because it yielded the highest accuracy in a variety of test cases, was the fastest model to train, and provided a decision tree which could be investigated to provide a deeper understanding of predictions. REPTree can be used in two modes: as a *regression tree*, where the predicted outcome a real number, and as a *classification tree*, where the predicted outcome is a class, or a range of degradation values in our case. Regression mode produced a higher accuracy than the classification mode, so we use it in our model.

We experimented with several accuracy-improving techniques and found *bagging* to be very effective. Bagging, also referred to as *bootstrap aggregating*, is known to lower the error rate, reduce the variance and help avoid over-fitting. Bagging works by creating m new datasets each populated through sampling from the original dataset, uniformly with replacement. From each of these m data sets a new model is created, providing us with m models. Each model is then used to produce an estimate, and all these estimates are averaged to create the final estimate. When bagging was used, the average error of our REPTree model reduced by over 6%, and the time to train the model and make predictions

remained reasonable.[5]

The REPTree model is well suited for online use, because an estimate can be produced exceptionally quickly (tens of microseconds in our experiments). The tree is represented as a table, and all that is required is a few table look-ups, whose number is proportional to the depth of the tree.

[5]We also attempted to use bagging on the Gaussian Process model, however it crashed upon building due to memory overflows.

Chapter 3

Results

We begin by taking a closer look at the decision tree created for the *Intel* and *AMD* systems (section 3.1). Then, in section 3.2 we present the quantitative evaluation of the model's accuracy, along with the analysis of scenarios producing high errors (section 3.3), and the confidence predictor (section 3.4), a solution for anticipating inaccurate predictions online.

3.1 Analysis of the decision tree

A decision tree is structured as a collection on nodes and branches, where each node contains the attribute used for making the branching decision and the corresponding threshold value. For instance, in our tree for the *Intel* system the attribute showing the number of delayed bus transactions (L2_REJECT_BUSQ:MESI from Table 3.1) is used for branching at the root of the tree as shown in figure 3.1. Instances that generated fewer than roughly 5 delayed bus transactions per thousand instructions (27,792,189 events per 5 billion instruction instance as seen in figure 3.1) follow the right branch, the rest of the instances follow the left branch.

Examining the *Intel* tree[1], we observed that the number of delayed bus transactions (L2_REJECT_BUSQ:MESI) was one of the most important attributes in the decision tree, as it is used for branching decisions at almost every tree level starting from the root and appears as the branching attribute in 10% of all the tree nodes. In contrast, less important events, such as the number of locked data reads from L1 cache appear in fewer than 0.1% of the nodes. The L2_REJECT_BUSQ event indicates that a pending L2 cache request that

[1]We are unable to show the full tree because of its size, but we discuss the salient points.

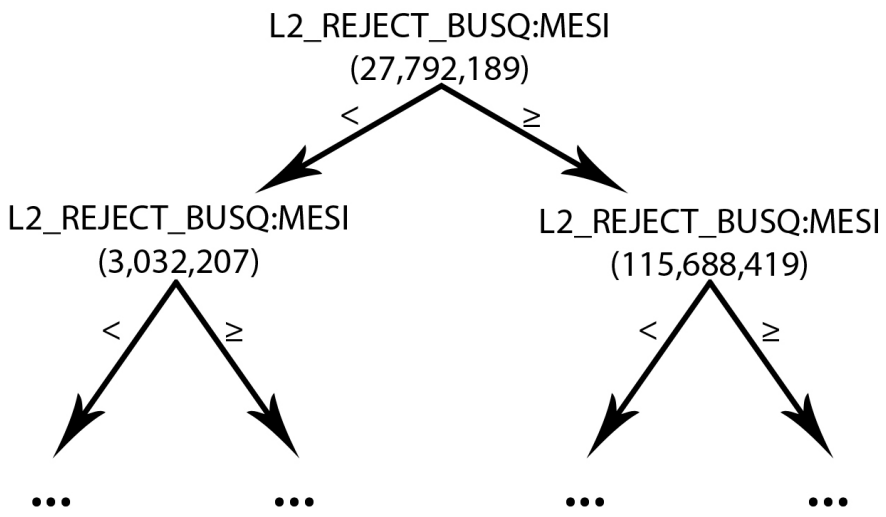

Figure 3.1: The root of the Intel decision tree. The number under the node name represents the value that is used for branching.

requires a bus transaction is delayed from moving to the bus queue, which typically occurs because the bus queue is full. Therefore, frequent occurrence of this event is indicative of front-side bus contention, which was also shown to be one of the key bottleneck resources on the Intel Clovertown processor by Zhuravlev et al. [22]. Other events that are used most frequently in decision making include L1 store misses (L1D_CACHE_ST), L2 cache requests (L2_RQSTS), evicted L2 cache lines that were dirty (L2_M_LINES_OUT) and other events indicative of the bus traffic (BUS_TRANS_INVAL and BUS_TRANS_P).

Somewhat surprisingly, we observed that many of the most significant attributes were related to the write intensity of the workload, e.g., L1D_CACHE_ST and L2_M_LINES_OUT. This could indicate that the underlying system is not able to buffer the writes to the extent that their effect on the memory system is minimal.

The strongest positive correlation with degradation was observed for the following attributes: L2_REJECT_BUSQ (0.87 correlation coefficient), UNHALTED_CORE_CYCLES (0.64), BUS_TRANS_P (0.48), L2_M_LINES_OUT (0.26) and L1D_CACHE_ST (0.22). The strongest negative correlations were observed for INST_RETIRED.STORES (-0.25) and BR_IND_MISSP_EXEC (-0.21).

The decision tree for the AMD system included a larger number of attributes and it

was more difficult to pin-point the key resources responsible for contention. However, some of the most frequently used attributes were the unhalted clock cycles (which essentially indicates the cycle per instruction (CPI) rate of the program), the number of processor cycles the decoder is stalled because the reorder buffer is full, and the number of L2 cache misses that resulted from hardware prefetch requests into the data cache. CPI naturally correlates with degradation, as programs that are more memory-bound (and thus have a higher CPI) are more likely to suffer from resource contention. The number of stall cycles in the reorder buffer indicates that the program is waiting for resources that are unavailable, and could be indicative of memory-system contention. The number of prefetch requests that resulted in accesses to DRAM is a gauge on the memory-system pressure; this agrees with the observation of Zhuravlev et al. that prefetching activity puts significant pressure on the memory system and is largely responsible for performance degradation [22].

3.2 Evaluation of the model's accuracy

We evaluate the model's on both the Intel and AMD architectures, both evaluations use cross-validation. For each primary benchmark, we predict its degradation in the clean and random co-schedules. To produce the estimates, we first remove from the data set *all* execution instances containing this benchmark (either as the primary or interfering). We train the model on the reduced data set and then produce the estimate of the degradation. This way we ensure that the model is not trained on any instances of the application for which it is trying to make predictions. This is the most rigorous validation procedure of all the available options.

Our metric of accuracy, the *error rate*, is the absolute difference between the estimated and the actual measured degradation. For instance, if the measured degradation was 5%, but we predicted 7%, the error would be 2%. Another option was to use a *relative error*, i.e., the percent by which the prediction differs from the true value; in the preceding example, the relative error would be 40%. The downside of relative errors is that they obscure the magnitude and importance of mispredictions. For instance, we will have 100% relative error both if the true degradation is 1% and we predict 2%, and if the true degradation is 100% and we predict 200%. However the mis-prediction in the first case is not significant, while in the second case it is. That is why we feel that the absolute error is a more fair reflection of the model's accuracy. Figure 3.2 shows the cumulative distribution of prediction errors

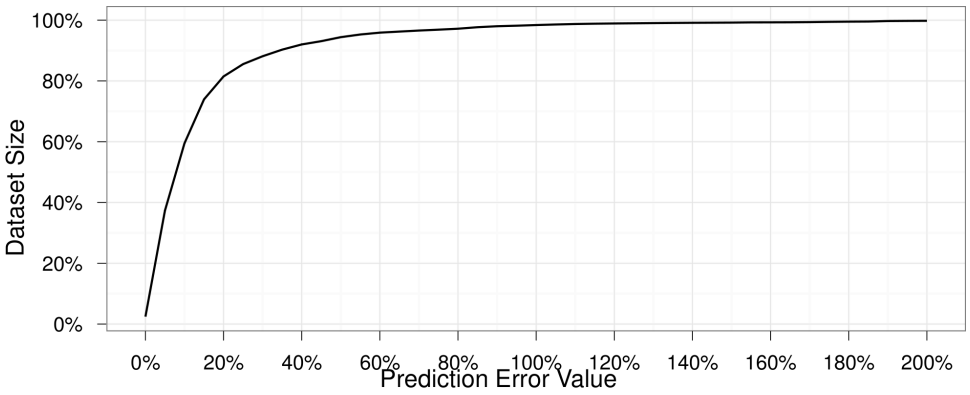

Figure 3.2: Cumulative distribution of errors

produced by our model. For both architectures we observe that 80% of the errors are under 20%. The average error for Intel was 16%, and AMD was 13%.

3.2.1 Intel architecture results

We first evaluate our model's accuracy on the Intel platform.

Figure 3.3 shows the estimated and predicted degradation over time (for all instances) for two benchmarks that are representative of the results that we observed. Each benchmark was run in many co-schedules; however for readability we report on their respective co-schedules that produced the smallest prediction error (Min Error Co.), median error (Median Error Co.) and the highest error (Max Error Co.). In the case of *tonto* we observe that the model is very good at following degradation trends over time; we observed similar behaviour with other benchmarks exhibiting temporal variation in degradation. A full selection of benchmarks run, with their respective min, median, and highest error co-schedules, is attached in Appendix A.

Figure 3.4 summarizes the time series graphs in figure 3.3 and appendix A together into one graph. The x-axis shows the degradation values, the y-axis shows the primary benchmarks. For each benchmark, the dot indicates the true degradation, and the triangle indicates the value predicted by the model. The length of the line connecting the two symbols correlates with the magnitude of the error. The first chart on the left shows the results for the co-schedules that produced the smallest error for each benchmark, the second chart

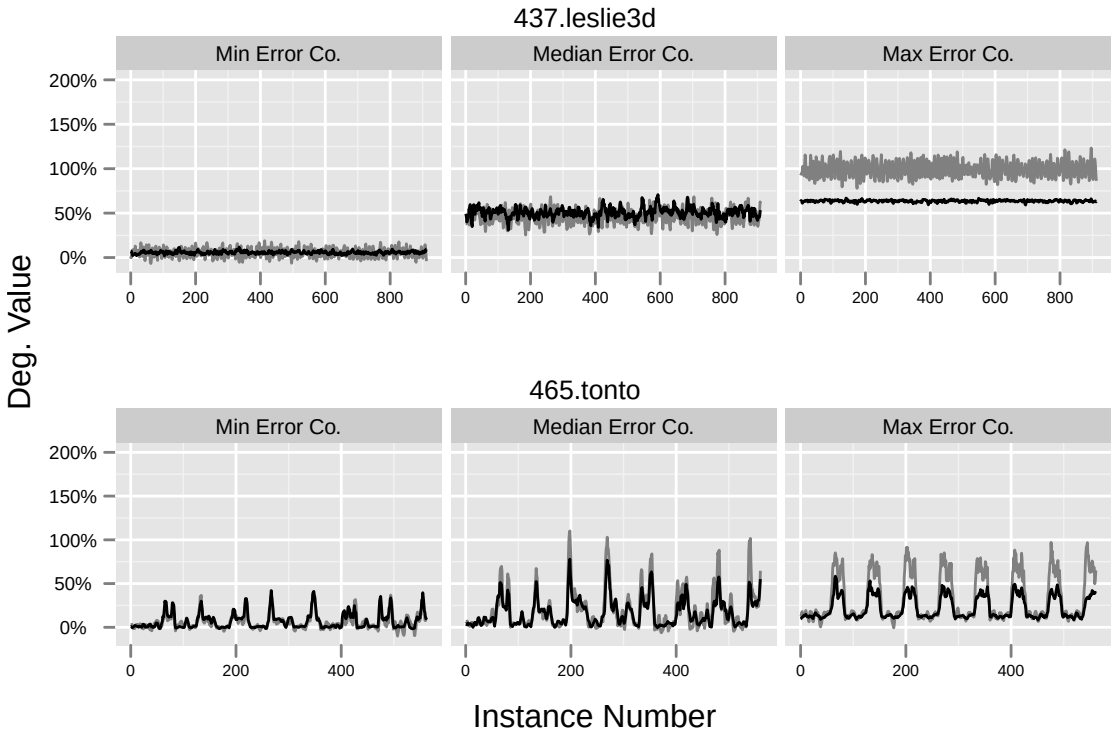

Figure 3.3: Estimated vs. predicted degradation for all instances over time for two selected benchmarks on the **Intel** architecture. We show the co-schedules that produced the lowest, median and the highest errors.

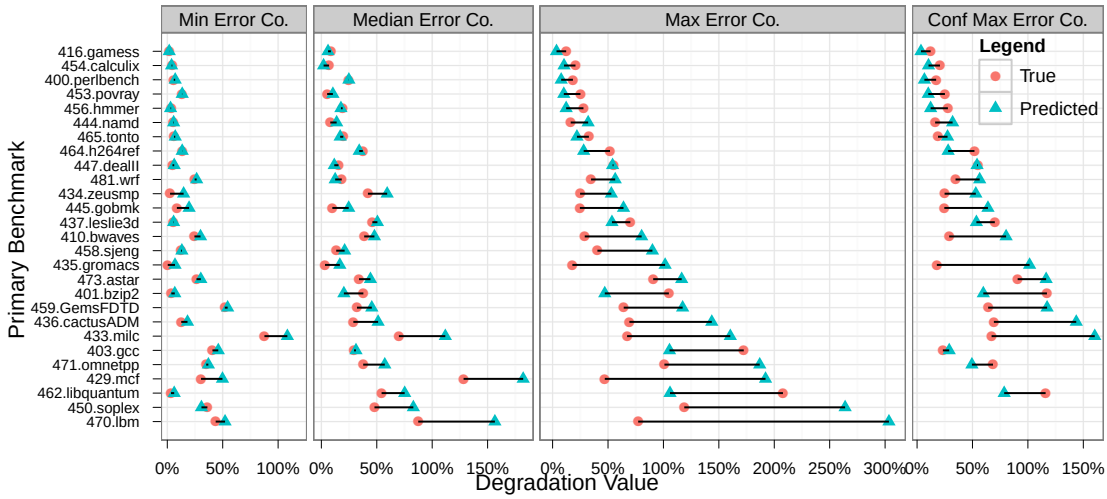

Figure 3.4: Difference between the actual and predicted degradation for the best, median and worst predicted co-schedules for each primary benchmark. The right-most chart shows the max-error co-scheduled when we apply the confidence predictor.

shows the median-error co-schedules, the third chart shows the highest-error co-schedules.

Min- and median-error charts show that, with a few exceptions, the prediction errors are quite small. From the highest-error chart we observe that there are a few large errors for high-degradation benchmarks, such as *lbm*, *soplex*, *libquantum* and *mcf*. As we will demonstrate in the next section, these benchmarks show behaviour that is distinct from the other benchmarks in the training set. Since cross-validation ensures that we do not train on the benchmarks whose degradation we are trying to predict, the model is not trained to recognize these "outliers". The fourth chart in Figure 3.4 shows how the results for highest-error co-schedules improve when we apply the confidence predictor; these data and the confidence predictor will be explained in Section 3.4.

3.2.2 AMD architecture results

Next we evaluate how well our model building methodology transfers between architectures and the model's accuracy on an AMD platform. To create a model for the AMD architecture we simply followed the exact same steps as we did for the Intel architecture: record a training set using SPEC CPU benchmarks, perform attribute selection to determine which event

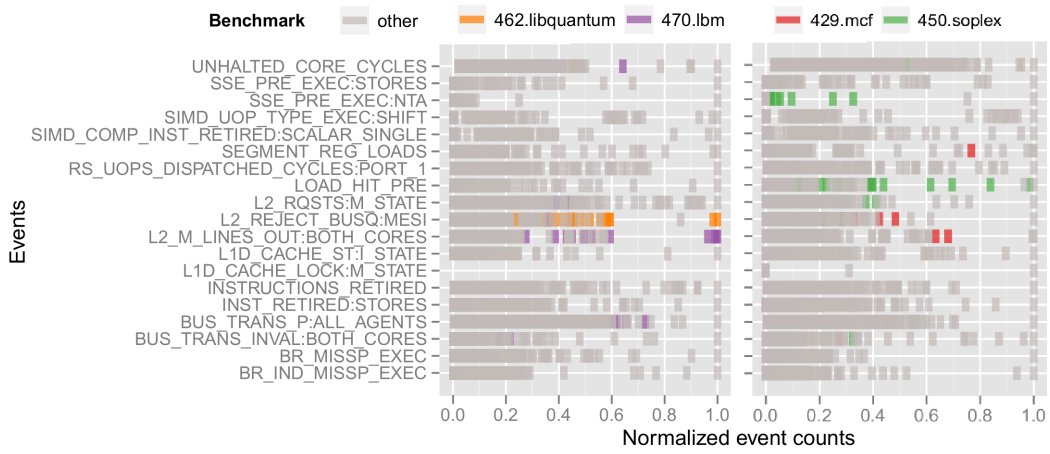

Figure 3.5: Highlighted event counters for *libquantum*, *lbm*, *soplex* and *mcf*. (Must be viewed in colour).

counters are important, and build a REPTree regression model using these attributes. The AMD results have also been cross validated using the same technique as was used for the Intel architecture.

Figure 3.6 shows the time series graphs for the same two benchmarks as was shown in figure 3.4 for the Intel architecture. From this graph we can see that while the applications true degradation values are different between architectures our model is able to make accurate predictions for both. Figure 3.7 summarizes all the time series graphs in both figure 3.7 and in appendix B. When comparing both the Intel and AMD summarized graphs (figures 3.4 and 3.7) we can see that our model performs similarly, regardless of architecture. Note that the same benchmarks, *lbm*, *soplex*, *libquantum* and *mcf* have, as did in the Intel results, the highest errors, and benefit most from the confidence predictor. A full selection of benchmarks run, with their respective min, median, and highest error co-schedules, is attached in Appendix B.

3.3 Uncovering the outliers

With cross-validation, the training set contains absolutely no instances of the application whose degradation we are trying to predict. So, for instance, if we are predicting *lbm*,

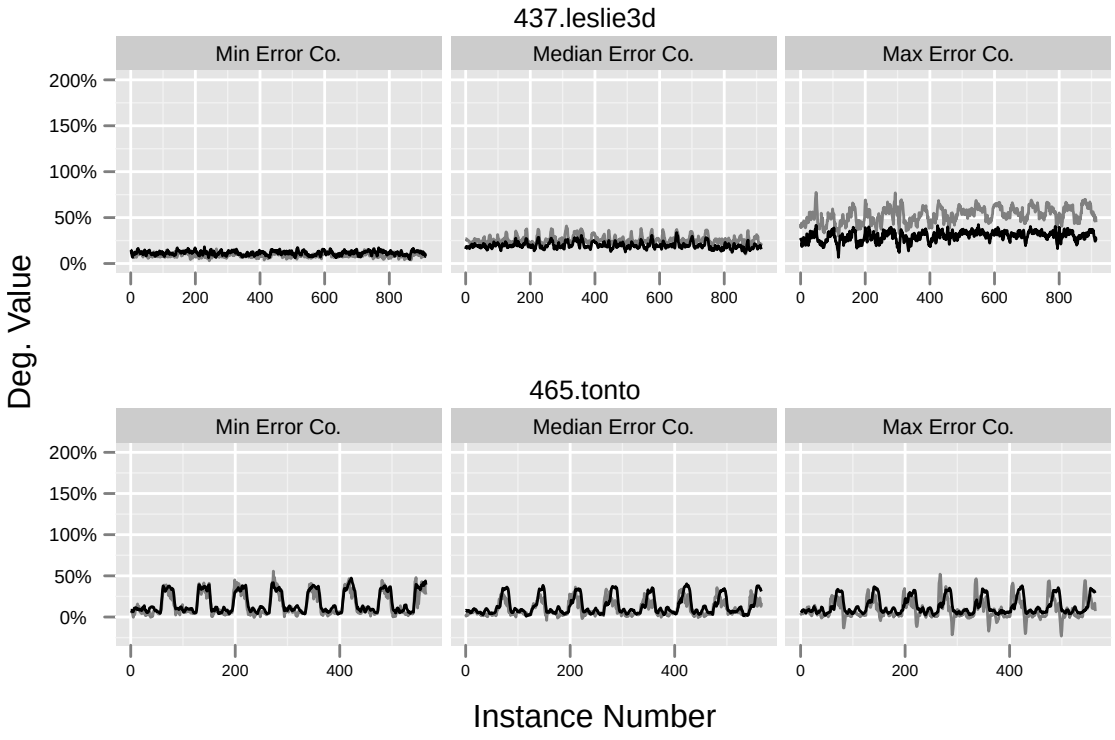

Figure 3.6: Estimated vs. predicted degradation for all instances over time for two selected benchmarks on the **AMD** architecture. We show the co-schedules that produced the lowest, median and the highest errors.

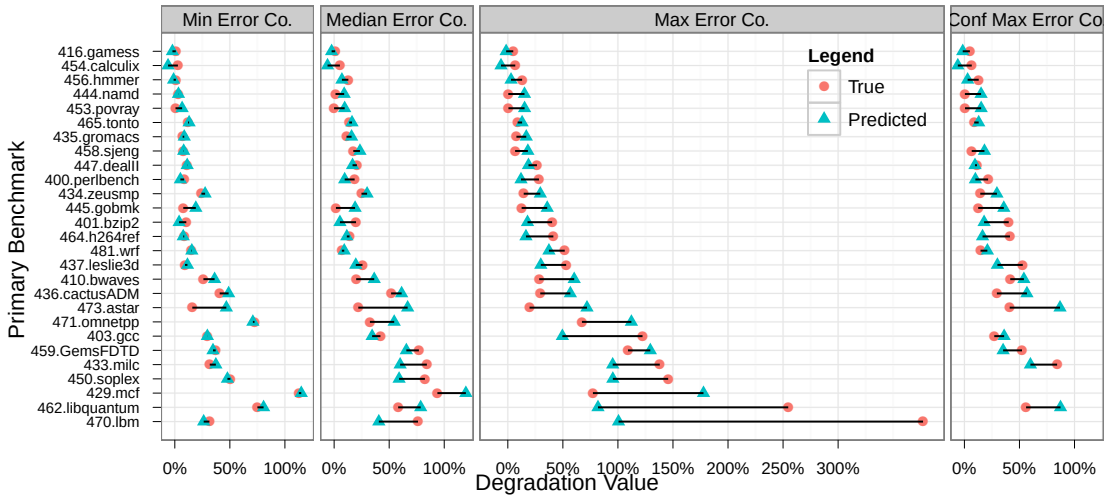

Figure 3.7: Difference between the actual and predicted degradation for the best, median and worst predicted co-schedules for each primary benchmark on the **AMD** architecture. The right-most chart shows the max-error co-scheduled when we apply the confidence predictor.

the model was trained with all event instances involving *lbm* removed. This makes it very difficult for the model to predict this instances as it has never seen them, and must rely on similar instances in the training set which, if lbm performs significantly different to anything found in the training set, can lead to inaccurate predictions. Therefore, if *lbm*'s attributes are very different from the training set, we could use this variation to explain inaccurate predictions and, furthermore, to *anticipate* them. To test this theory, we compare performance attributes of the outliers to those in the rest of the set.

Figure 3.5 compares the event counter values of four benchmarks with the highest errors in the max-error co-schedules (*lbm*, *libquantum*, *mcf* and *soplex*) with those of the other benchmarks for the Intel architecture. Appendix C contains the same graph for the AMD architecture. We begin by analyzing *lbm*. Figure 3.5 clearly shows that *lbm* is an outlier in the attribute L2_M_LINES_OUT. This event count is high when the workload evicts a lot of L2 cache lines in the modified state. When the evicted cache lines are dirty, the memory system has to do more work in handling cache misses, because modified lines must be written back to memory. *Lbm* happens to be a very write-intensive application [16], and since its value of L2_M_LINES_OUT is extremely high, our model extrapolates an extremely

high degradation. In reality, writes, while certainly adding pressure to the memory system, contribute to execution latency only indirectly, since they are handled asynchronously. (The correlation of L2_M_LINES_OUT with performance degradation is only 0.26). As a result, our model greatly overestimates the degradation.

We now look at *libquantum*. In sharp contrast to *lbm*, *libquantum* performs very few writes [16] and is unique in its extreme latency sensitivity. It has very poor cache reuse and as a result spends 100% of execution time in *memory episode*s, where it is waiting for at least one memory request [12]. Indeed, in Figure 3.5 we observe that *libquantum* has an unusually high value of the attribute L2_REJECT_BUSQ:MESI, which occurs when a pending data request from the L2 cache is delayed from moving to the bus queue, and is indicative of long memory episodes. No other benchmark in the SPEC CPU2006 suite has similar behaviour. As a result of its latency-sensitivity, *libquantum* suffers significantly more from contention than other benchmarks, because any increase in memory-system latency has a direct effect on its perforamnce. Since *libquantum* is the only benchmark with this behaviour, the model is unable to capture its extreme latensy-sensitivity and so it consistently underestimates the degradation for *libquantum*.

Looking further at *soplex*, we observe that it has a vastly different prefetching behavior than the rest of the benchmarks. First of all, it has a somewhat higher count of software prefetch events (SSE_PRE_EXEC:NTA) than other benchmarks. But what is particularly interesting is that it has a dramatically high count of LOAD_HIT_PRE events, which counts load operations conflicting with a software prefetch to the same address. For every other benchmark in the suite except one[2], this event count is close to zero. This means that soplex performs extremely effective prefetching, since a large number of loads already have a corresponding prefetch request in flight. Effective prefetching masks memory latency, and so contention for shared resources has a much smaller impact on *soplex*'s performance than one might expect. Since *soplex* is the only benchmark with such property, the model is unable to correctly factor in the effect of successful prefetching and so it typically overestimates *soplex*'s degradation.

Finally, examining the outlier events for *mcf*, we see that, similarly to *lbm* and *libquantum*, it has a large number of L2_REJECT_BUSQ and L2_M_LINES_OUT events. Although when we validate *mcf*, the training data does include *lbm* and *libquantum*, the number of the

[2] *Gobmk* also has a high occurrence of this event, but it is still roughly four times smaller than for *soplex*.

instances that include these benchmarks is very small, relative to the other instances that have much lower counts of the two events. As a result, the model is not strongly trained to make accurate predictions in this case.

The key insight that we gain from this analysis is that the applications responsible for the highest errors can be predictably identified if we analyze how their performance attributes are different from those of other applications. We use this insight to create a method for *anticipating* when the model is likely to produce a high error.

3.4 Confidence Predictor

The confidence predictor decides whether or not the model is likely to make an accurate prediction by comparing the hardware counter attributes of the instance whose degradation we are about to predict with the distribution of attribute values seen in training. If two or more attributes of the to-be-predicted instance are more than two standard deviations away from the mean of the values seen in training, the confidence predictor marks the instance as non-confident and produces a null prediction. There are several ways how the operator can handle null predictions. One possibility is to conservatively label this workload as high-degradation and not to consolidate it – this policy can be used for workloads with strict QoS requirements. Another solution is to add a copy of this workload to a database of training benchmarks, so that the model can be trained on it in the future.

The fourth chart in Figures 3.4 and 3.7 (Intel and AMD respectively) shows the highest-error co-schedules once we introduce the confidence predictor. We observe that the magnitude of errors is substantially reduced. As the trade-off, the predictions for a few benchmarks are not made: this would occur if all instances for that benchmark are marked as non-confident. This is expected: if the behaviour of the benchmark is drastically different from the behaviour seen in training, the model would not be able to confidently predict any of its instances. In this case, it is best to either re-train the model on the benchmark or to conservatively assume a high degradation. Overall, about 25% of the instances are omitted as non-confident.

Figure 3.8 shows the scatter plots of the errors produced by the Intel and AMD models when we make the estimates for *all* instances, regardless of confidence, and when we produce the estimates only for confident instances. The confidence predictor substantially helps to filter out erroneous predictions, reducing the maximum error by about a factor of two.

The average error for all instance on Intel is around 16%, but it is reduced to 10% when the confidence predictor is applied. On our AMD system, the confidence predictor reduces the maximum error by about a factor of 3× and improves the average error from 13% to 10%. Both Intel and AMD confidence predictors mark 25% of their respective instances as non-confident.

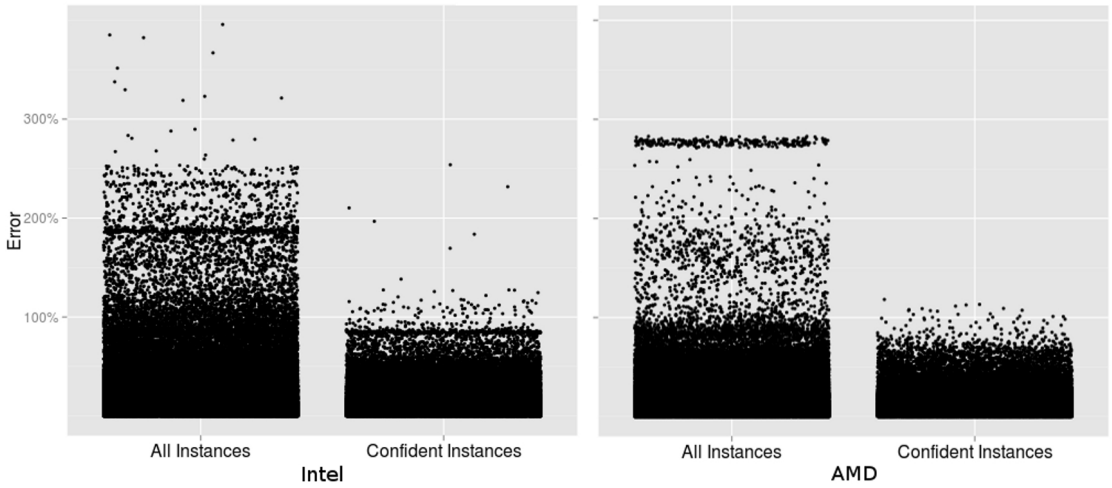

Figure 3.8: Errors for all instances (left) and only confident instances (right) for both Intel and AMD architectures

In summary, we conclude that machine learning is a reasonable method for estimating the complex effect of sharing-induced performance degradation on multicore processors. It is able to produce accurate predictions in the majority of the cases, but when the training set has insufficient diversity we can anticipate high errors in the model by applying the proposed confidence predictor.

Event Name	Event Description
UNHALTED_CORE_CYCLES	Clock cycles elapsed.
INSTRUCTIONS_RETIRED	The number of retired instructions.
RS_UOPS_DISPATCHED_CYCLES:PORT_1	The number of cycles for which micro-ops are dispatched for execution on port 1. Indicative of processor utilization.
SEGMENT_REG_LOADS	Number of segment register loads.
SSE_PRE_EXEC:NTA	This is a software prefetching event. Counts the number of times the SSE prefetch NTA instruction is executed.
SSE_PRE_EXEC:STORES	This is a software prefetching event. Counts the number of times SSE non-temporal store instructions are executed.
L2_M_LINES_OUT:BOTH_CORES	Counts the number of L2 modified cache lines evicted by both cores. Indicative of the pressure on the memory hierarchy.
L2_RQSTS:M_STATE	Counts all completed L2 cache requests, including hardware prefetches. M_STATE counts accesses to cache lines whose content differs from that in in the memory).
L2_REJECT_BUSQ:MESI	Counts event when a pending L2 cache request that requires a bus transaction is delayed from moving to the bus queue. This can happen, for instance, when the bus queue is full.
L1D_CACHE_ST:I_STATE	Counts the number of data writes to cacheable memory that missed the cache.
L1D_CACHE_LOCK:M_STATE	Counts the number of locked data reads in modified state from cacheable memory.
LOAD_HIT_PRE	Counts load operations conflicting with a software prefetch to the same address.
BUS_TRANS_INVAL:BOTH_CORES	Counts invalidate bus transactions for both cores, which can be generated, for instance, when a cache line write misses the L2 cache.
BUS_TRANS_P:ALL_AGENTS	Counts all partial bus transactions.
BR_MISSP_EXEC	Counts the number of mispredicted branch instructions.
BR_IND_MISSP_EXEC	Counts the number of mispredicted indirect branch instructions.
SIMD_UOP_TYPE_EXEC:SHIFT	SIMD packed shift micro-ops executed.
INST_RETIRED:STORES	Counts the number of instructions retired that contain a store operation.
SIMD_COMP_INST_RETIRED	Retired computational Streaming SIMD Extensions (SSE) scalar-single instructions.

Table 3.1: List of the attributes selected by attribute selection

Chapter 4

Use Case for the Model

In this chapter we describe how our model could be used for improving resource-allocation decisions in high-performance computing (HPC) clusters. HPC clusters run scientific applications, many of which are structured as multi-process jobs communicating via a message-passing interface (MPI). By default, cluster scheduling algorithms, such as Maui [19] or Moab [1], will assign a process to every available core on the server, but since many MPI applications are very memory-intensive, they will experience substantial performance degradation when sharing a multicore CPU [4]. The cluster operator may want to spread such jobs across a larger number of nodes, so as to avoid unreasonable performance degradation, but with existing tools it is difficult to decide whether the degradation is high enough to justify using extra hardware.

To demonstrate how the proposed model can address this problem, we prototype a new cluster scheduler that uses the model for scheduling decisions. The proposed scheduler, described later, improves on two baseline cluster scheduling policies: *Best-fit* and *Min-collocation*. *Best-fit*, which is the most commonly used policy, allocates the processes of the same job on all available cores on the node, using additional nodes if needed, but if a single job does not fill all the cores, it fills them with processes of another job. The other baseline policy, *Min-collocation*, attempts to schedule no more than one job per node, as long as there are unused nodes available.

The *Best-fit* policy ensures maximum hardware utilization, but allows contention for multicore resources. *Min-collocation* would produce less resource contention, but will use more hardware (and power). We demonstrate how to find the balance between these extremes with a new *Balanced* scheduler that relies on our model.

The *Balanced* scheduler initially assigns jobs to nodes following the *Best-fit* policy. It then begins monitoring the hardware counter values selected by the model, and estimates the performance degradation for each job. If the degradation is estimated higher than the acceptable threshold, set to 50% in our experiments, the *Balanced* scheduler starts up an additional server and migrates the suffering job to that server. The scheduler is also able to migrate a part of the job by operating on individual containers, but these partial migrations did not occur in our experiments. This proof-of-concept scheduler is simple and does not take into account communication overhead that may occur if the processes of the same job run on several nodes. This is deferred to future research; in this work we show how to avoid contention-induced performance degradation.

The *Balanced* scheduler is implemented as a collection of daemons that run on each node. Within each node, jobs are scheduled by a user-level scheduler *Clavis* [3] that is based on the Distributed Intensity algorithm [4, 22], and assigns threads to cores so as to avoid multicore resource contention. *Clavis* is used as the intra-node scheduling policy under *Best-fit* and *Min-collocation* policies as well.

Our experimental environment mimics an HPC cluster. We do not have exclusive access to the actual cluster where we are able to modify the scheduling algorithm, so instead we used three identical multicore systems connected by the Gigabit Ethernet. We did not have multiple *Intel* systems, so our mini-cluster is comprised of the three *AMD* systems (Table 2.1). Each system has two multicore CPUs with six cores each.

In order to be able to migrate MPI processes from one node to another after the job had begun execution, we place the processes into OpenVZ containers [2], which is a light-weight virtualization option for Linux. OpenVZ produced the lowest overhead compared to Xen and KVM, and offered better reliability than MPI checkpoints. The degradation for the job is estimated by averaging the degradation estimates for the corresponding containers.

We show two experiments demonstrating the benefits of the *Balanced* policy and the underlying model. The first experiment shows that the *Balanced* scheduler is able to respect performance degradation threshold, unlike the *Best-fit* scheduler, while using less energy than the *Min-collocation* scheduler. The second experiment shows that the *Balanced* scheduler that uses our model saves energy relative to the same scheduler that estimates performance degradation using a simple heuristic model proposed in the earlier work [22]. In both experiments we run four MPI jobs from the SPEC MPI suite, each with six processes.

	Experiment 1		
	Node 1	Node 2	Node 3
Best-fit	fds0, tachyon	fds1, fds2	
Min-Collocation	fds0, fds2	tachyon	fds1
Balanced	fds0, tachyon	fds1	fds2
	Experiment 2		
	Node 1	Node 2	Node 3
Balanced (DI model)	fds, tachyon	zeus1	zeus2
Balanced (our model)	fds, tachyon	zeus1, zeus2	

Table 4.1: Job allocation across nodes

We report the running times and the energy consumed to run the workload[1]. The model was trained on SPEC CPU 2006 application; it did not include the MPI applications that we test.

Experiment 1: Improved performance fidelity. In this experiment, we run three copies of the *fds* application and one copy of *tachyon*. *Fds* is memory-intensive, so it would suffer performance degradation under contention, while *tachyon* would not. The *Balanced* scheduler is configured to avoid performance degradation above 50%. Table 4.1 shows how the jobs were assigned to servers under the three algorithms, and Figure 4.1 shows the running time (relative to solo) and energy consumption. The red line indicates the 50% degradation threshold. Migration overhead is always included into the running times that we report.

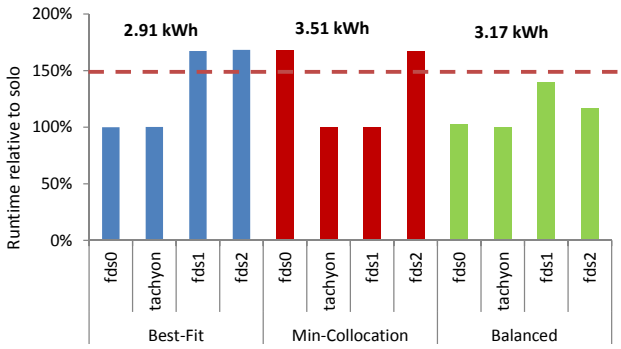

Figure 4.1: Performance and energy consumption during Experiment 1.

[1]Energy was measured using the Dell Remote Access Control interface on our servers.

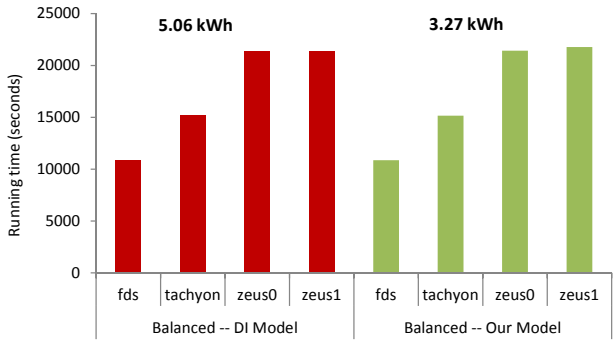

Figure 4.2: Performance and energy consumption during Experiment 2.

Under *Best-fit* and *Min-collocation*, only one copy of *fds* is able to meet the 50% degradation constraint. The other two copies of *fds* suffer roughly 70% performance degradation, because they run together on the same server. Both schedulers assign jobs to nodes according to the order of their arrival. The jobs arrive in the order *fds0, tachyon, fds1, fds2*, so *Best-fit* fills the first node with *fds0* and *tachyon*, then fills the second node with the other two copies of *fds*. *Min-collocation* assigns one job per node, but when *fds2* arrives and all three nodes are filled, it is forced to scheduled *fds2* with *fds0* on Node 1. Although *Min-collocation* has an additional node at its disposal, it is unable to realize that it is better to run a copy of *fds* alone on Node 3 rather than *tachyon*.

Balanced, on the other hand, discovers that two copies of *fds* co-located on the same node will suffer more than 50% performance degradation and migrates one of them to the third node. As a result, it improves performance by about 15% on average (across all applications) relative to *Best-fit* and *Min-collocation*, while using 44% less energy than *Min-collocation*. Even though *Balanced* uses the same number of nodes as *Min-collocation*, it enables the workload to complete quicker, hence smaller energy consumption. *Best-fit* uses 8% less energy than *Balanced*, because it uses fewer nodes, but unlike *Balanced* it does not meet the 50% degradation threshold.

Experiment 2: Improved power efficiency. The purpose of this experiment is to demonstrate the benefit of precise estimates of performance degradation that would be produced by our model, as opposed to coarse estimates that would be produced by heuristic-based models [15,22]. We compare the *Balanced* scheduler that uses our model with the same

scheduler, but that uses the miss-rate based model underlying the Distributed Intensity (DI) algorithm. We refer to this version as *Balanced-DI*. The DI model checks if any co-located jobs have the miss rate greater than one miss per thousand instructions. If that threshold is exceeded, the jobs are deemed "contentious" and the scheduler distributes them to different CPUs, or in our case, different servers. The DI model is considered state-of-the-art, as most software-only contention aware algorithms relied on the models almost identical to DI [4, 13, 15].

We run the following jobs: *fds*, *tachyon* and two copies of *zeus*. Like in the first experiment, the schedulers are configured to avoid the degradation above 50%. Table 4.1 shows how the schedulers assign the jobs to nodes. *Balanced-DI* observes that *zeus* has the miss rate of 25 misses per 1000 instructions, which by far exceeds its thresholds, so it migrates one copy of *zeus* to the third node. However, it turns out that despite its high miss rate, *zeus* experiences only negligible degradation when co-scheduled with another copy (see Figure 4.2). The *Balanced* scheduler that uses our model is able to produce an accurate estimate, so it does not migrate *zeus* to another node, and meets the degradation threshold while using 35% less energy than *Balanced-DI*.

Chapter 5

Related Work

Our work is the first to evaluate machine learning for modeling performance degradation on multicore CPUs. Besides machine learning, there are three major strategies that attacked the same problem: *analytical modeling* (often requiring unconventional hardware), *models based on heuristics*, and *trial-and-error methods*.

Analytical modeling. One of the first models for resource contention on multicore chips was proposed by Chandra et al. [6]. It estimated the increase in the last-level cache (LLC) miss rate resulting from cache contention. Chandra's model required unconventional hardware which in limited cases could be substituted with compiler extensions. The main limitation of this model is that it focused only on caches and did not address other resources, such as memory buses, system request queues, hardware pre-fetchers, etc., contention for which was found to be a crucial factor in performance degradation on modern CPUs [4, 22]. Machine learning models will capture contention in any hardware component as long as this component is represented by relevant performance events.

Eyerman, Hoste and Eeckhout [7] used a semi-manual methodology for modeling CPI stacks. They estimated unknown relationships using regression analysis. However, at the heart of their method is a generic analytical model for the processor. As we explained, we wanted to find a practical method that does not involve any manual model construction, and machine learning answered these needs.

Luque et al. developed a method to precisely count how many extra cycles the thread is wasting, waiting for CPU resources that are occupied as a result of contention [14]. This information can be used directly to estimate the performance degradation that contention is causing. While this is a very promising technique in terms of accuracy, it requires changing

the hardware. Furthermore, this technique, at the time of this writing, addresses only shared caches. Our goal was to design a method that will work on today's hardware and cover all kinds of shared CPU resources.

Models based on heuristics. In recent studies, the last-level cache miss rate was used as a heuristic to predict whether threads or processes sharing a multicore CPU are suffering performance degradation [4, 13, 15, 22]. In that work, the LLC miss rate was used to decide when the threads should be scheduled on separate chips to avoid cache contention. While suitable for coarse-grained scheduling decisions, the miss rate is not sufficient to estimate performance degradation with a greater precision.

Furthermore, relying on a single indicator of performance (the miss-rate) to estimate the effect of sharing multiple resources is a fragile strategy. It may work as long as memory controllers and pre-fetch bandwidth are key contended resources on multicore systems [22], but if the hardware bottlenecks change, the heuristic will stop working. Furthermore, this method does not easily allow integration of other shared resources into the model. Machine learning can adjust to changes in hardware and be extended to model any new resources that emerge as important for contention; therefore, it is a more future-proof method.

Trial-and-error methods. Trial-and-error methods require running the workloads in various combinations (co-schedules) with other workloads [18] or with dummy benchmarks [8]. The goal is to observe how performance degradation changes in different co-schedules and to use that information to create online a machine- and workload-specific model of the degradation. A system called Cuanta is a very elegant solution, relying on a set of "clones", each with a particular cache access pattern. By co-scheduling all clones with a target application, we can find the one that most closely mimics the behaviour of that application. Then, based on a previously constructed degradation matrix and application clones we can predict the degradation for any pair of applications. This approach works well when the number of cores per chip is small, but as the number grows, we would need to run a larger and larger combination of clones concurrently with the application. This is not practical, because the cores are unavailable to run other applications when we use them to run clones. Our machine learning model, on the other hand, requires hardware counter values that can be measured for all cores in parallel, and so the time to perform the on-line measurement does not grow with the number of cores.

Chapter 6

Conclusions

Our study aimed to investigate the effectiveness of machine learning in modeling contention-induced performance degradation: online, on a live workload, and without *a priori* knowledge of applications or the need to run them in isolation. We aimed for a model that seamlessly ports across different systems, and machine learning met this need as it does not rely on microarchitectural knowledge. We found that machine learning can indeed be used to build reasonably accurate models, which estimate degradation within 16% of the true value on average, however inaccurate estimates can occur if the test application is very different from the applications in the training set. Fortunately, these cases can be anticipated by checking how "dissimilar" the test application is from the training set, in terms of its attribute values. Our proposed method, the confidence predictor, successfully anticipates when the model is likely to produce an inaccurate estimate and reduces the maximum error by up to a factor of three and the average degradation error down within 10%.

Appendix A

Intel Co-schedules

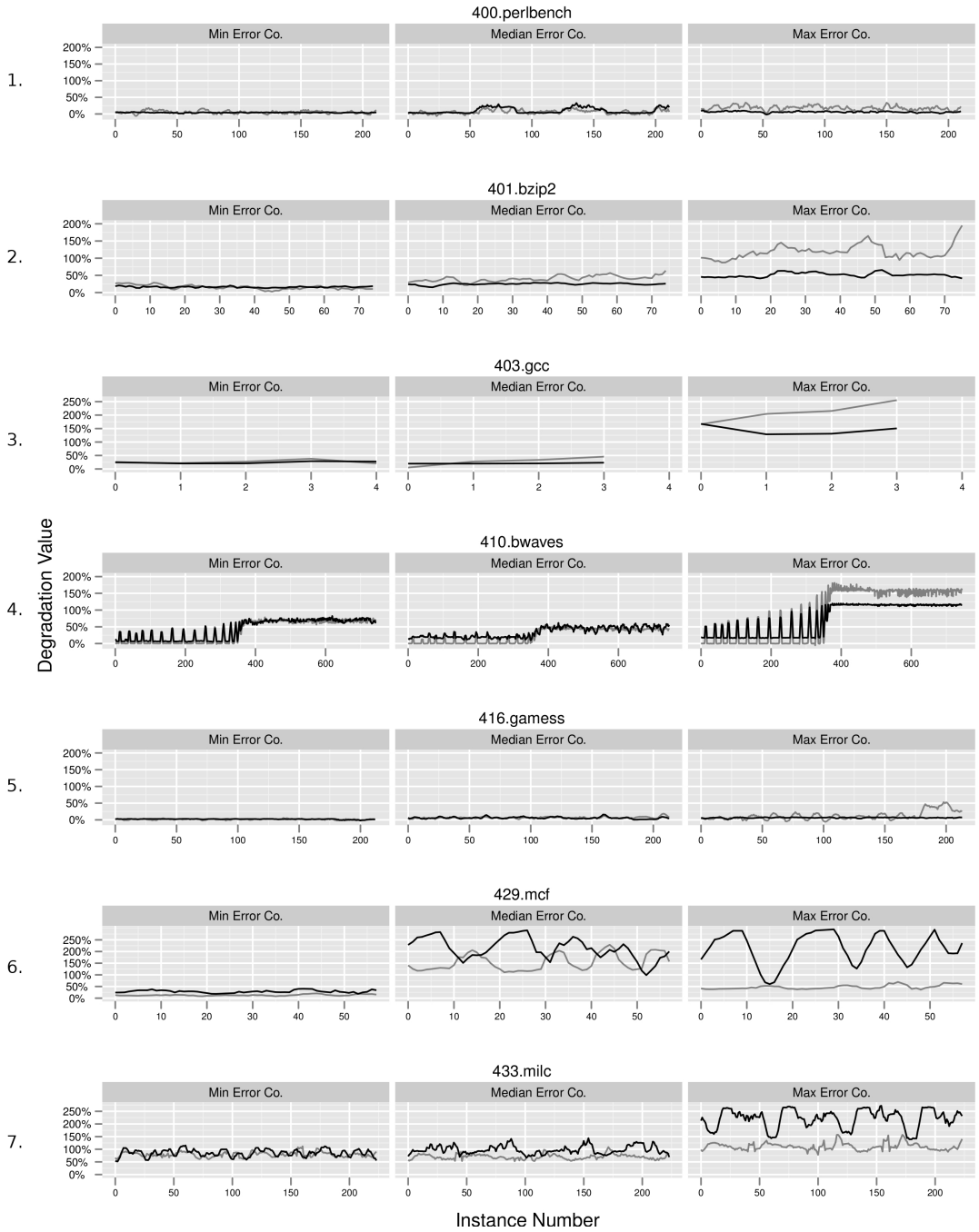

Figure A.1: The min, mean, and max error co-schedules for Intel benchmarks 1-7. Using both clean and random co-schedules.

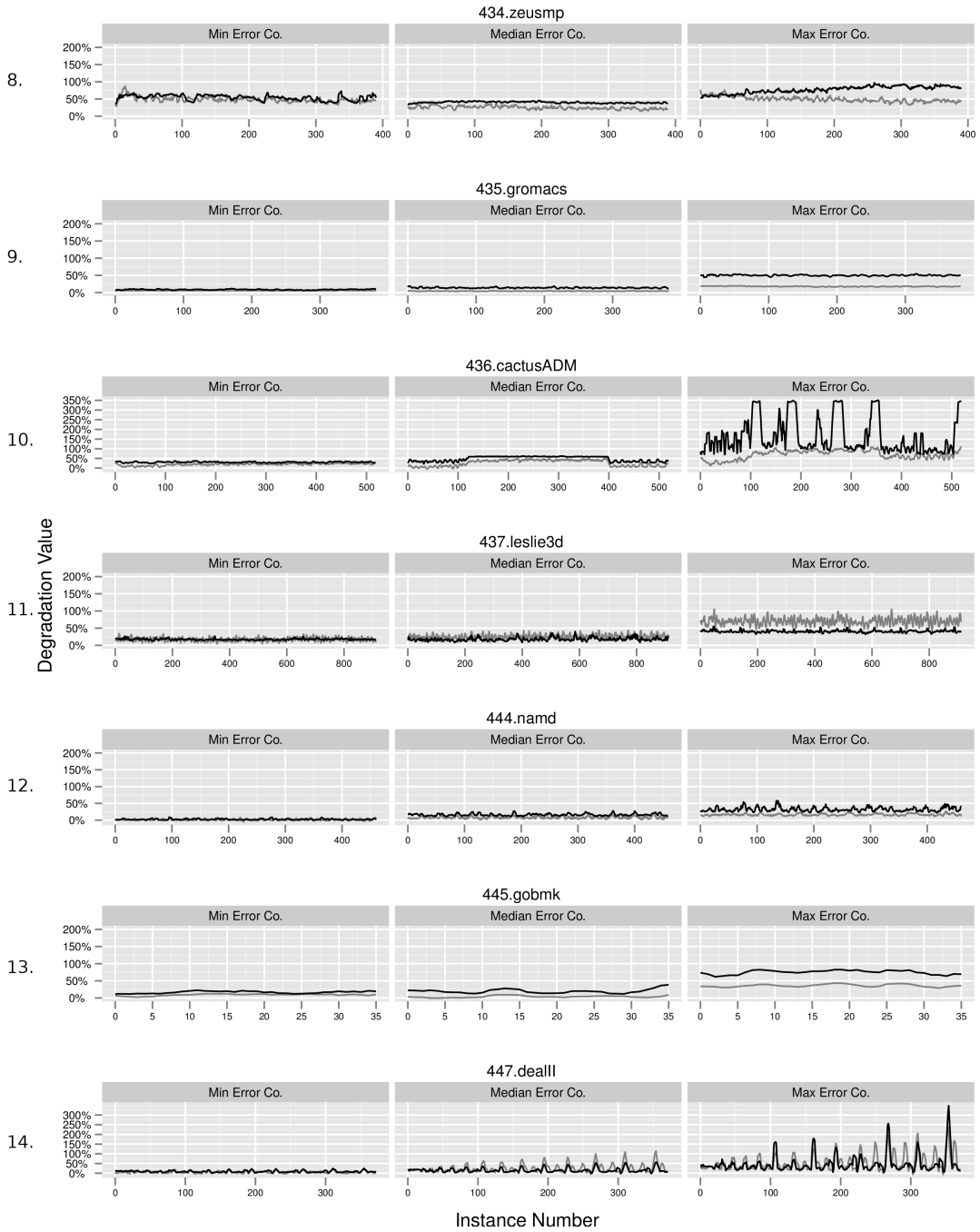

Figure A.2: The min, mean, and max error co-schedules for Intel benchmarks 8-14. Using both clean and random co-schedules.

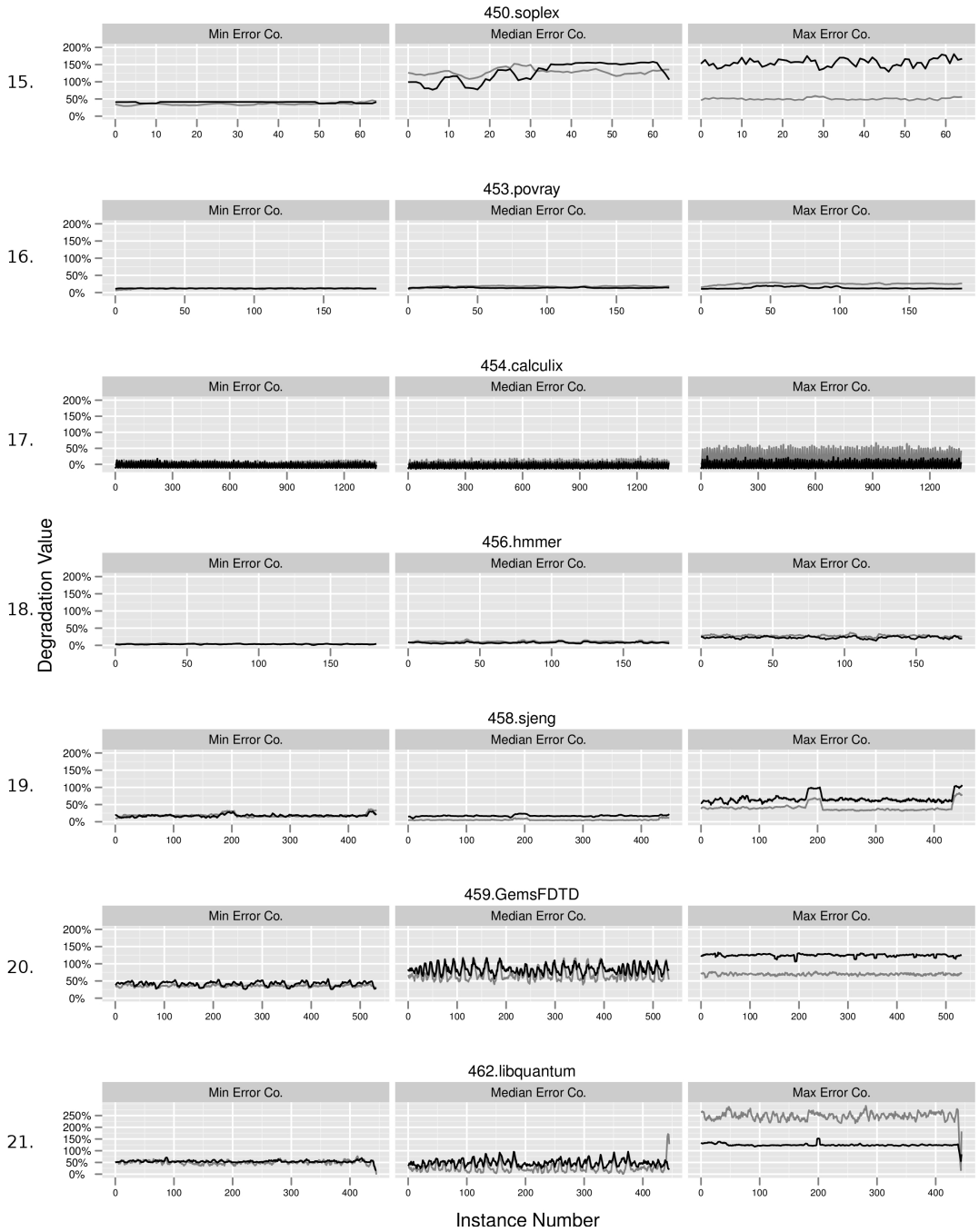

Figure A.3: The min, mean, and max error co-schedules for Intel benchmarks 15-21. Using both clean and random co-schedules.

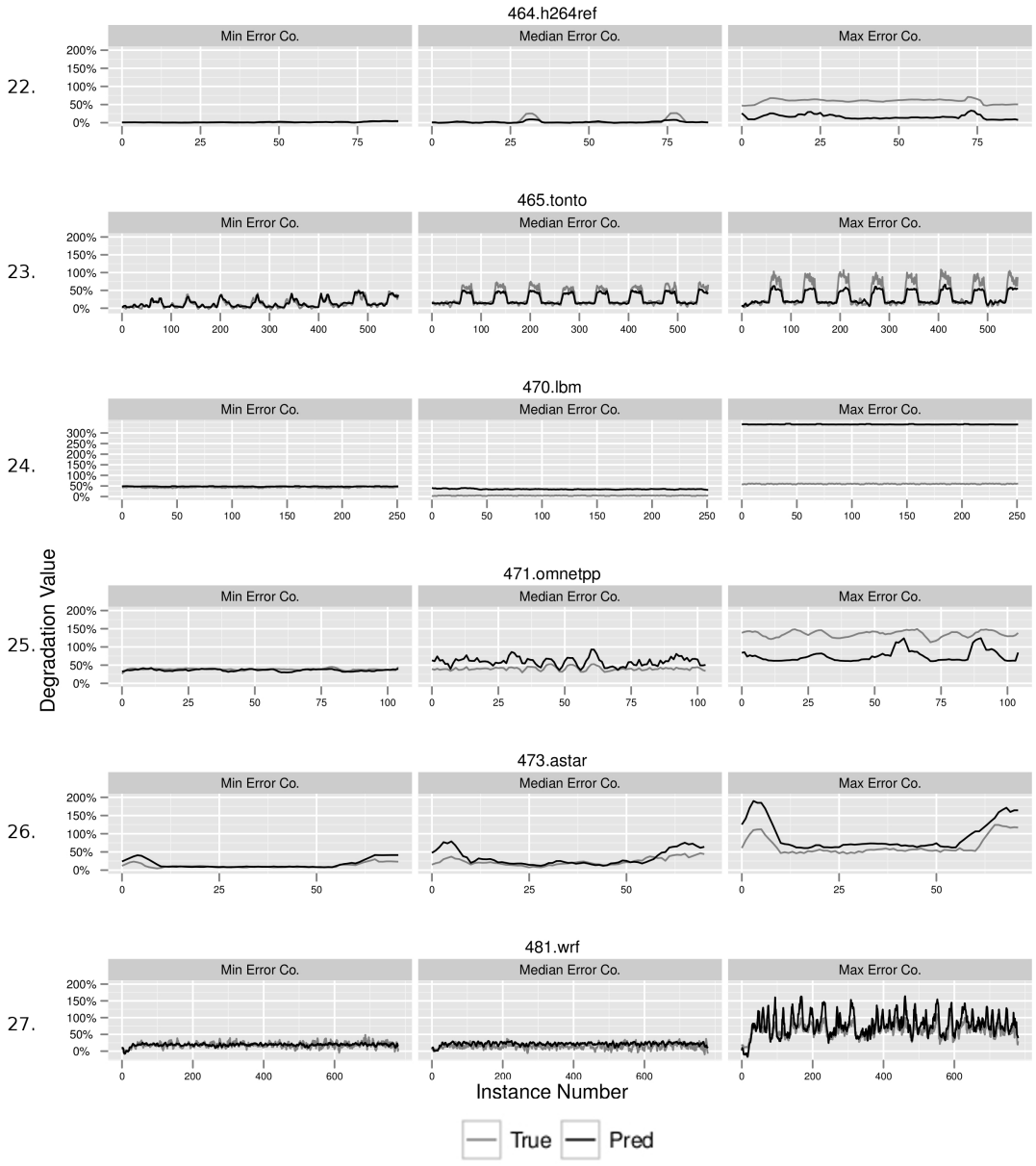

Figure A.4: The min, mean, and max error co-schedules for Intel benchmarks 22-27. Using both clean and random co-schedules.

Appendix B

AMD Co-schedules

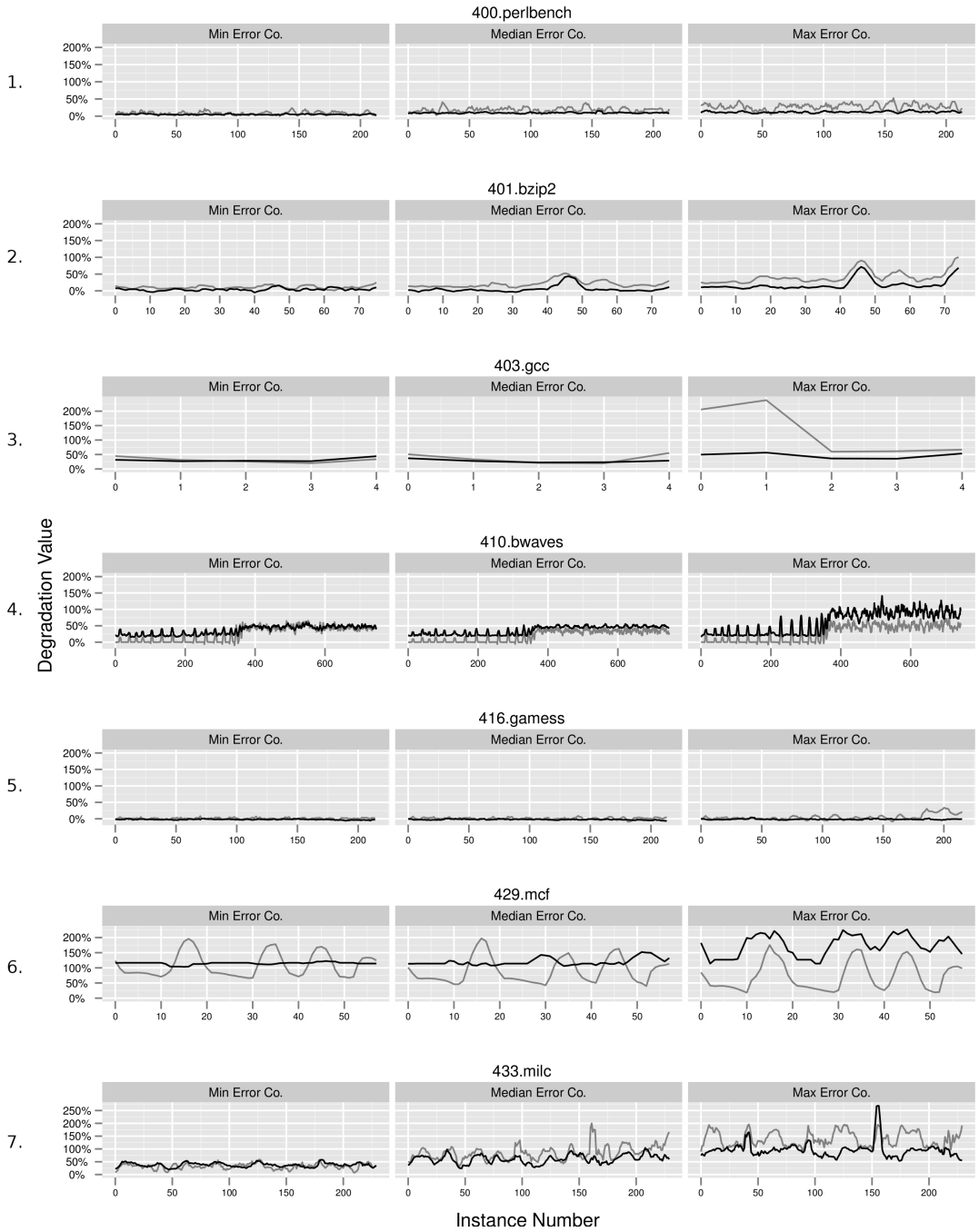

Figure B.1: The min, mean, and max error co-schedules for AMD benchmarks 1-7. Using both clean and random co-schedules.

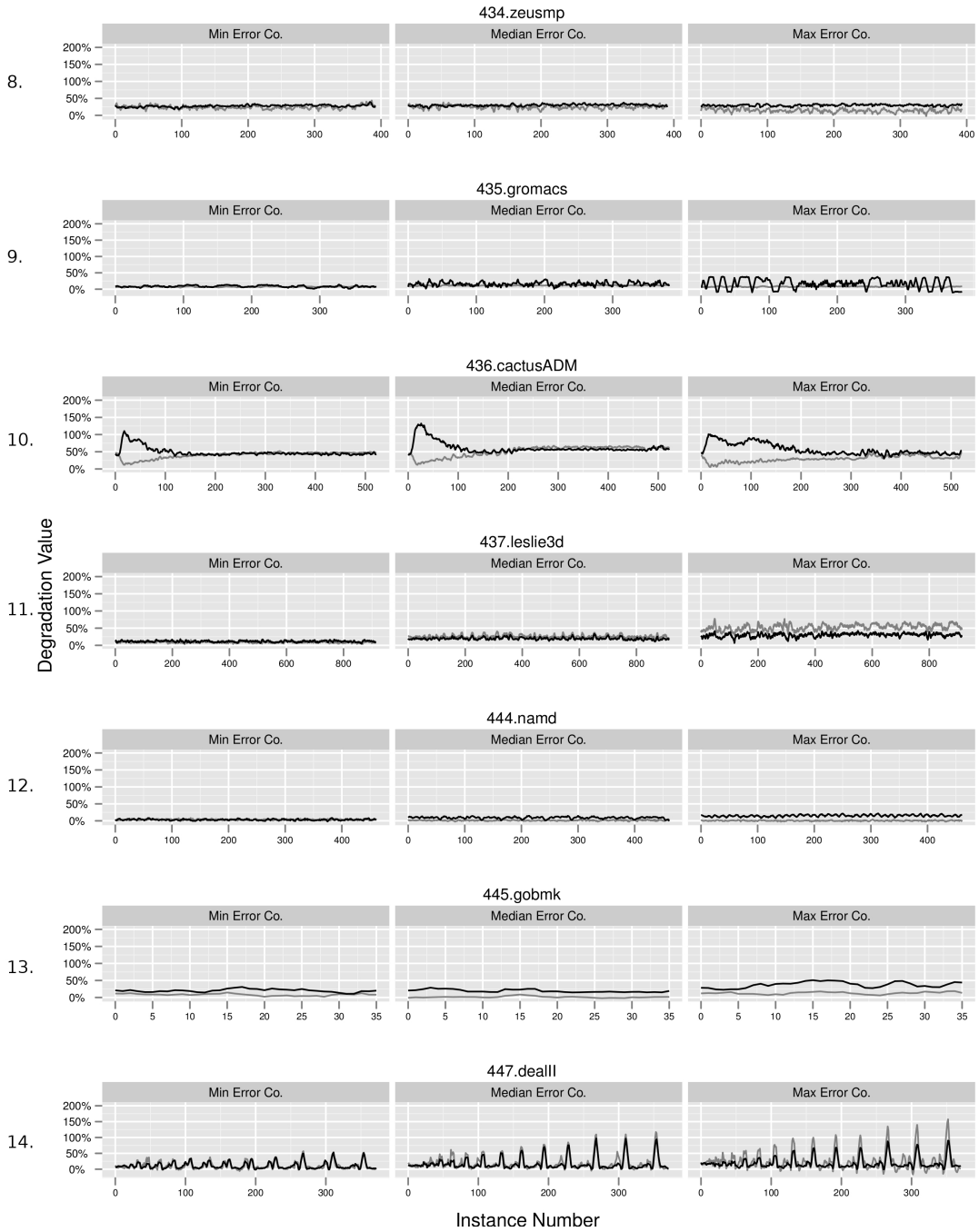

Figure B.2: The min, mean, and max error co-schedules for AMD benchmarks 8-14. Using both clean and random co-schedules.

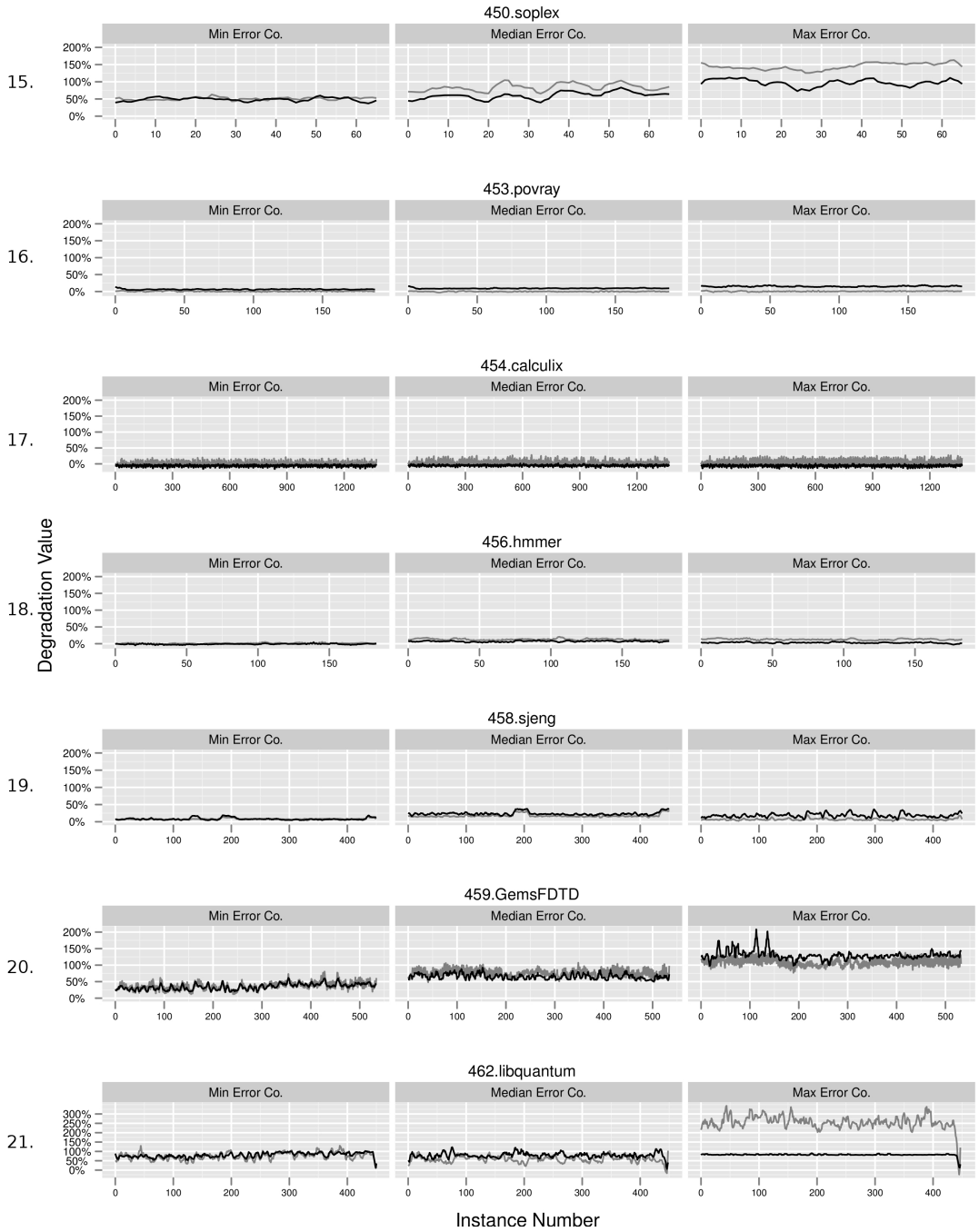

Figure B.3: The min, mean, and max error co-schedules for AMD benchmarks 15-21. Using both clean and random co-schedules.

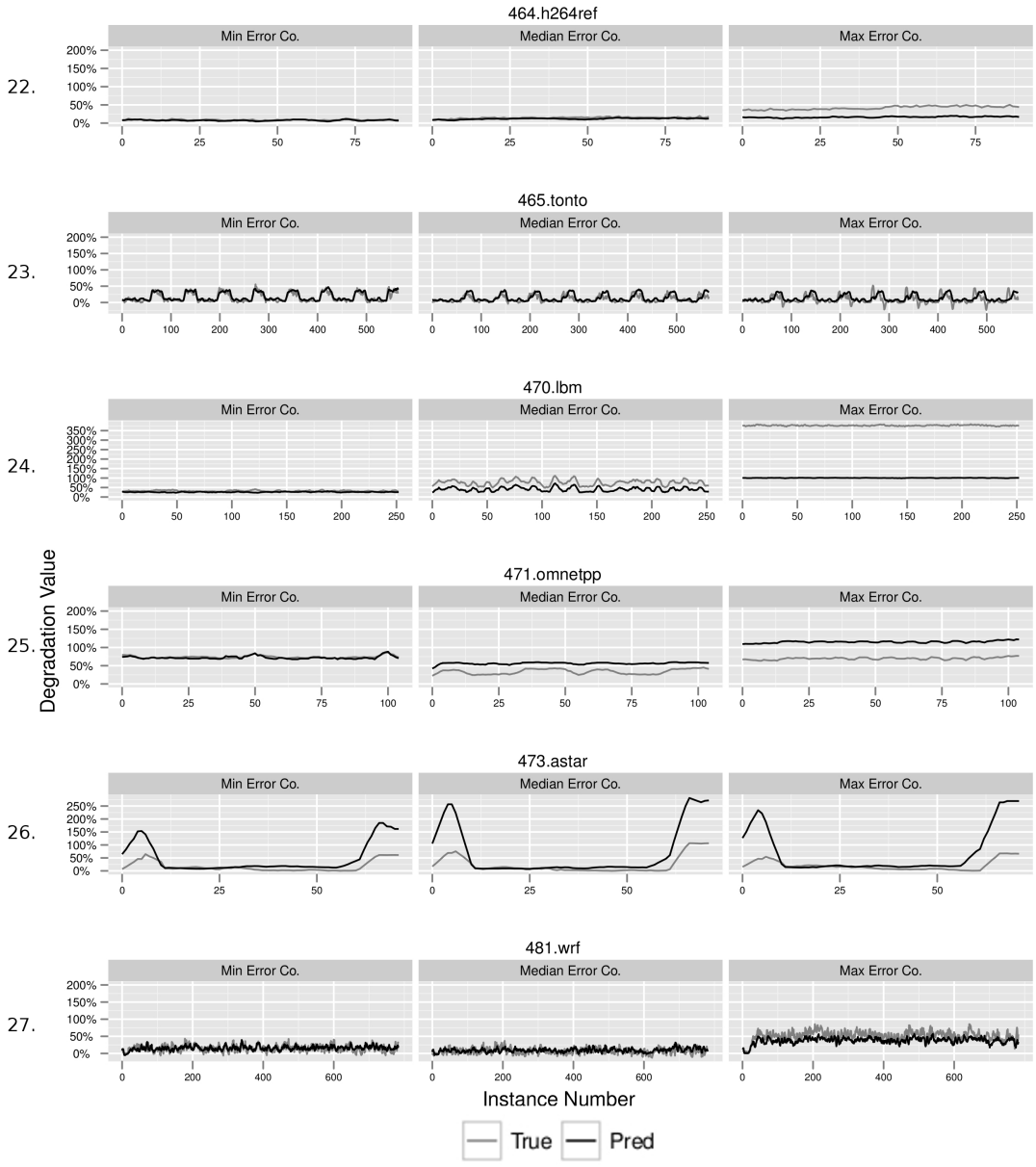

Figure B.4: The min, mean, and max error co-schedules for AMD benchmarks 22-27. Using both clean and random co-schedules.

Appendix C

AMD Highlight event counters


Benchmark
lbm
libquantum
mcf
soplex
other
Counter
SYS_SIZED_COMMANDS:POSTED_WRITE_BYTE
SYS_MEMORY_CONTROLLER_BYPASS:HIGH_PRIORITY
SYS_L3_FILLS_CAUSED_BY_L2_EVICTIONS:CORE_3_SELECT
SYS_L3_CACHE_MISSES:ALL
SYS_GART:APERTURE_HIT_FROM_CPU
SYS_GART:ALL
SYS_DRAM_ACCESSES_PAGE:MISS
SYS_CACHE_BLOCK:ALL
CPU0_SYSTEM_READ_RESPONSES:DATA_ERROR
CPU0_SYSTEM_READ_RESPONSES:ALL
CPU0_SEGMENT_REGISTER_LOADS:GS
CPU0_RETIRED_SSE_OPERATIONS:SINGLE_DIV_OPS
CPU0_RETIRED_SERIALIZING_OPS:X87_BOTTOM_SERIALIZING_UOPS
CPU0_RETIRED_SERIALIZING_OPS:SSE_BOTTOM_EXECUTING_UOPS
CPU0_RETIRED_MOVE_OPS:ALL_OTHER_MOVE_UOPS
CPU0_RETIRED_MMX_AND_FP_INSTRUCTIONS:MMX_AND_3DNOW
CPU0_RETIRED_INSTRUCTIONS
CPU0_RETIRED_FASTPATH_DOUBLE_OP_INSTRUCTIONS:POSITION_2
CPU0_RETIRED_CPUID_INSTRUCTIONS
CPU0_REQUESTS_TO_L2:INSTRUCTIONS
CPU0_QUADWORDS_WRITTEN_TO_SYSTEM:QUADWORD_WRITE_TRANSFER
CPU0_PIPELINE_RESTART_DUE_TO_PROBE_HIT
CPU0_MEMORY_REQUESTS:WRITE_COMBINING
CPU0_LS_BUFFER_2_FULL_CYCLES
CPU0_LOCKED_OPS:CYCLES_WAITING
CPU0_LOCKED_OPS:ALL
CPU0_L2_CACHE_MISS:HW_PREFETCH_FROM_DC
CPU0_INSTRUCTION_CACHE_MISSES
CPU0_INEFFECTIVE_SW_PREFETCHES:SW_PREFETCH_HIT_IN_L2
CPU0_FP_SCHEDULER_CYCLES:ALL
CPU0_FPU_EXCEPTIONS:SSE_AND_X87_MICROTRAPS
CPU0_DR2_BREAKPOINT_MATCHES
CPU0_DISPATCH_STALL_FOR_REORDER_BUFFER_FULL
CPU0_DISPATCH_STALL_FOR_FPU_FULL
CPU0_DISPATCHED_FPU:OPS_STORE
CPU0_DISPATCHED_FPU:OPS_MULTIPLY_PIPE_LOAD_OPS
CPU0_DATA_CACHE_MISSES
CPU0_DATA_CACHE_LINES_EVICTED:ALL
CPU0_CPU_CLK_UNHALTED
0.0
0.2
0.4
0.6
0.8
1.0
Scaled Values

Bibliography

[1] Moab Adaptive HPC Suite. In *http://www.adaptivecomputing.com/resources/docs/*.

[2] OpenVZ: Container-based Virtualization for Linux. *wiki.openvz.org*.

[3] Sergey Blagodurov and Alexandra Fedorova. User-level Scheduling on NUMA Multicore Systems under Linux. In *Proc. of Linux Symposium*, 2011.

[4] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A Case for NUMA-Aware Contention Management on Multicore Systems. In *Proceedings of USENIX Annual Technical Conference*, 2011.

[5] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28:8:1–8:45, December 2010.

[6] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, 2005.

[7] Stijn Eyerman, Kenneth Hoste, and Lieven Eeckhout. Mechanistic-empirical processor performance modeling for constructing cpi stacks on real hardware. In *ISPASS*, 2011.

[8] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[9] Mark A. Hall. Correlation-based Feature Selection for Machine Learning. Master's thesis, University of Waikato, 1999.

[10] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT*, 2008.

[11] Ali Kamali. Sharing Aware Scheduling on Multicore Systems. Master's thesis, Simon Fraser University, 2010.

[12] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.

[13] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28(3):pp. 54–66, 2008.

[14] Carlos Luque, Miquel Moret, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu, and Mateo Valero. ITCA: Inter-task Conflict-Aware CPU Accounting for CMPs. In *PACT*, 2009.

[15] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. In *EuroSys*, 2010.

[16] Asit K. Mishra, Xiangyu Dong, Guangyu Sun, Yuan Xie, Narayanan Vijaykrishnan, and Chita R. Das. Architecting on-chip interconnects for stacked 3d stt-ram caches in cmps. In *ISCA*, 2011.

[17] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, 2010.

[18] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '02, pages 66–76, 2002.

[19] Sourceforge. Maui Scheduler Open Cluster Software. In *http://mauischeduler.sourceforge.net/*.

[20] David Tam, Reza Azimi, and Michael Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of EuroSys*, 2007.

[21] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *Proceedings of PPOPP*, 2010.

[22] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Contention on Multicore Processors via Scheduling. In *Proceedings of ASPLOS*, 2010.