Understanding the Scaling Behavior of the GFDL FMS High-Resolution Atmosphere Model on the Argonne BG/Q Platform

IBM BG/Q: A Platform for Performance Discovery

SAMARA TECHNOLOGY GROUP LLC

11 Chaplin Circle, Boxford, MA 01921-1305 USA
Phone: +1 (978) 352-8389
www.samaratechnologygroup.com

Table of Contents

Understanding the Scaling Behavior of the GFDL FMS High-Resolution Atmosphere Model on the Argonne BG/Q Platform....1
IBM BG/Q: A Platform for Performance Discovery....1
1 Executive Summary....3
2 Introduction....4
2.1 Background....4
2.2 Project History....6
3 Broad Goals....6
3.1 Background....6
3.2 Methodology....9
4 Findings....9
4.1 Scaling Results....9
4.2 Initial Analysis....10
4.3 Detailed Results....12
4.3.1 The New Test Configuration....12
4.3.2 Understanding Performance Through Globally Summed Hardware Counts....14
4.3.3 OpenMP Overheads....17
4.3.4 Performance Drivers....18
4.3.5 Conclusion....21
4.4 Machine Learning....23
5 Summary....24
6 Acknowledgements....24
Appendix A....25
Appendix B....27
Appendix C....30
Appendix D....33

1 Executive Summary

The work of this project applies the IBM BG/Q MPI and thread aware hardware performance tools MPITRACE, HPM and HPMPROF to analyze the loss of scaling performance for the GFDL 3.5km resolution cubed-sphere atmosphere model. Initially it seemed quite reasonable to ascribe scaling loss to communication issues and perhaps load imbalance. But detailed analysis of the data clearly demonstrates that calculations performed in the MPI subdomain halo regions is the primary source for scaling loss. Put another way, more MPI ranks means smaller work regions but more halo points. This produces the classic perimeter to area problem. A secondary factor turns out to be ever increasing OpenMP overheads associated with parallel regions containing very little work.

In separate but related work, we have applied Machine Learning techniques with hardware counter data as input in an attempt to predict performance. Within the scope of project resources and time, some successful results were achieved using non-linear models within a particular MPI layout. But this nascent work misses the role of expanding instruction counts due to the subdomain halos. Left for future work is the development of techniques that can probe across layouts to arrive at the central results concerning increasing instruction count and the loss of scaling performance.

2 Introduction

2.1 Background

Climate modeling, in particular the tantalizing possibility of making projections of future climate that have predictive skill on timescales of many years, is a principal science driver for exascale computing. Success with this effort will stretch the boundaries of computing along multiple axes:

- Resolution, where computing costs scale with the 3rd and 4th powers of problem size for dynamics and physics, respectively and the data archiving costs as 3rd power
- Complexity, as new subsystems of climate processes enter the simulation realm as feed-backs
- Capacity, as we build ensembles of runs to sample uncertainty, both in our knowledge and representation, and inherent in the chaotic system

The predictive understanding of climate change, and the detection and attribution of climate change to anthropogenic and natural components are among the leading human issues of our time. While the global-mean response of climate to anthropogenic forcing seems now to be confirmed with a great deal of confidence, the cutting edge of research and policy questions is now moving to the issue of understanding and predicting climate variability and change on regional scales, with lead times from weeks to decades. Such scales are the ones of most direct relevance to society and decision-makers.

Model resolutions in the Fifth Intergovernmental Panel on Climate Change Assessment Report (2013) are mostly in the 50-100km range for both ocean and atmosphere. A central concern for the next generation of models is to understand natural and forced variability as we make the next leap in resolution. This leap is particularly interesting as fundamental new physics appears in models of both atmosphere and ocean at 25km resolution and higher. We begin to see the direct influence of both ocean eddies and organized atmospheric storm systems (tropical cyclones and mid-latitude fronts).

A key result that may be obtained from the next generation of model resolution is an answer to the "decadal predictability" conundrum, which is principally driven by the ocean state: are there modes of variability of the coupled ocean-atmosphere system that are predictable on timescales of a decade or more; and to what extent is this dependent on ocean resolution? Conversely, we also seek to answer the question of whether the statistics of fine-scale phenomena such as the inter-annual variability in hurricane frequency and intensity is predictable on the basis of free-running Earth System Models.

The resolution, complexity, and capacity achievable on exascale platforms holds the promise to allow us to characterize the "tail" for the probability distributions where a lot of climate risk and difficult policy decisions reside.

The climate modeling community has been actively involved in providing feedback to the exascale design process. The exact contours of that system are yet unknown. But we do know that in one way or form, we are entering a realm of multi-billion-way concurrency in computation. This will pose extraordinarily daunting challenges.

We believe that the path forward toward exascale must address the extreme concurrency required.

In brief, we believe that one way to achieve this consists of:

- Climate system components that exhibit 10^5 concurrency based on the target resolutions
- Assembling the components into coupled systems where O(10) components are concurrently scheduled
- Running in ensembles of appropriate size for sampling outcomes which introduces O(10-100) concurrency
- Leveraging task-parallel work-flows that support another O(10) elements in the entire work-flow (simulation, archival, post-processing and analysis) to be concurrently scheduled.

We underline the fact that in the path outlined above, the exploitation of exascale systems will require a multi-pronged approach, in which scalable methods within components, scalable coupled numerics between components, and scalable work-flows across the complete climate modeling process managing models and data across massive ensembles of coordinated simulations, will all play a role.

Central to the success of the current climate research within NOAA has been GFDL's Flexible Modeling System (FMS). First developed in the Cray T3 era to enable GFDL's scientific transition from vector to scalable, parallel platforms, the past decade plus has been significant development within individual model components. For the atmosphere, these developments include the Cubed Sphere dynamical core and advances in atmospheric chemistry. For the ocean side, there have been multiple advances in bio-geochemistry and vertical mixing. Additionally, the land model has added time dependent vegetation cycles. Over this period, the the fundamental extensibility of the infrastructure has supported some of the most significant achievements in climate research over the past decade. Indeed from its beginnings on the parallel platforms of this century, the FMS modeling infrastructure and climate models supported by them have been extended to the petascale platforms of today with bright prospects for equally impressive achievements through what remains of this era.

For all its impressive achievements in support of climate science, it is clear that the existing modeling frameworks cannot simply be extended to the coming exascale platforms. Aside from the challenges posed by the highly threaded nature of coming compute node architectures, this infrastructure was developed in an era where I/O and memory footprint were essentially free and reliability of individual computational elements was a given. Even in this petascale era, we are finding difficulties with these assumptions. At exascale, we know with virtual certainty that memory and I/O use will need to be carefully controlled.

For almost as long as FMS has been a platform for climate science, it has served as a basis for studying current and future performance. After some initial experimental work on the BG/L located at Princeton University, FMS was running on the Argonne BG/P within 6 months of its arrival. From that point forward, work on the Argonne Blue Gene series of platforms has been a key driver for improvements in memory footprint control and scalability for FMS. These improvements are cornerstone elements supporting the IPCC AR5 production recently completed on Gaea at NCRC utilizing models running on *O*(10K) or more cores in an MPI / OpenMP framework.

The work described in this report represents the performance analysis facet of an Early Science Award to run the GFDL 3.5km resolution, non-hydrostatic model on BG/Q. The performance work is part of an initiative codified in 2011 to develop testbeds for the technologies pointing the way to

exascale capable platforms. Current efforts include increasing model component concurrency[1], studies of concurrent components on Intel Xeon Phi and nVidia GPUs as well as extremes of scaling on the BG/Q platform.

2.2 Project History

The work described in this report is a performance analysis off-shoot of an Early Science Project on the Argonne BG/P system. Titled "Climate-weather modeling studies using a prototype global cloud-system resolving model", the proposal targeted creation of a global atmospheric model capable of directly simulating deep convection and severe storms on a planetary scale. The goal of the project was to develop a software platform capable of predictive understanding of individual tropical storms, as well as the response of global and regional storm statistics to climate change.

With the resources provided the GFDL_esp on the Argonne BG/P and later BG/Q systems, excellent progress has been made towards the scientific goals. Nevertheless the software infrastructure supporting this work encountered significant performance challenges in attempting to scale to a million or more hardware threads. This project is an attempt to identify root causes and possible courses of action to improve model scaling behavior.

3 Broad Goals

Simply stated, the goal of this research is to understand and quantify the elements leading to the scaling behavior of the GFDL 3.5km Resolution Cubed-Sphere Atmosphere model on the Argonne IBM BG/Q (Mira). The Statement of Work reads as follows:

> *... determine the root cause(s) and possible solutions for the scaling behavior of the GFDL FMS 3.5km Cubed Sphere Atmosphere running on the Argonne BG/Q platform.*

3.1 Background

Data gathered prior to the start of the work reported here demonstrate that the strong scaling performance of multiple GFDL 3.5km cubed-sphere atmosphere models deteriorates significantly between 32K and 64K MPI-Ranks. A plot of scaling for the Held-Suarez model is provided below courtesy of Chris Kerr at GFDL (Figure 3.1)

1 In this context, concurrency means the capability to run a model component on its own set of hardware. For example, FMS has long had the capability to run the atmosphere and land on one set of processing cores and the ocean and ice on a different set. GFDL is currently in the process of making the solar radiation a concurrent component as well.

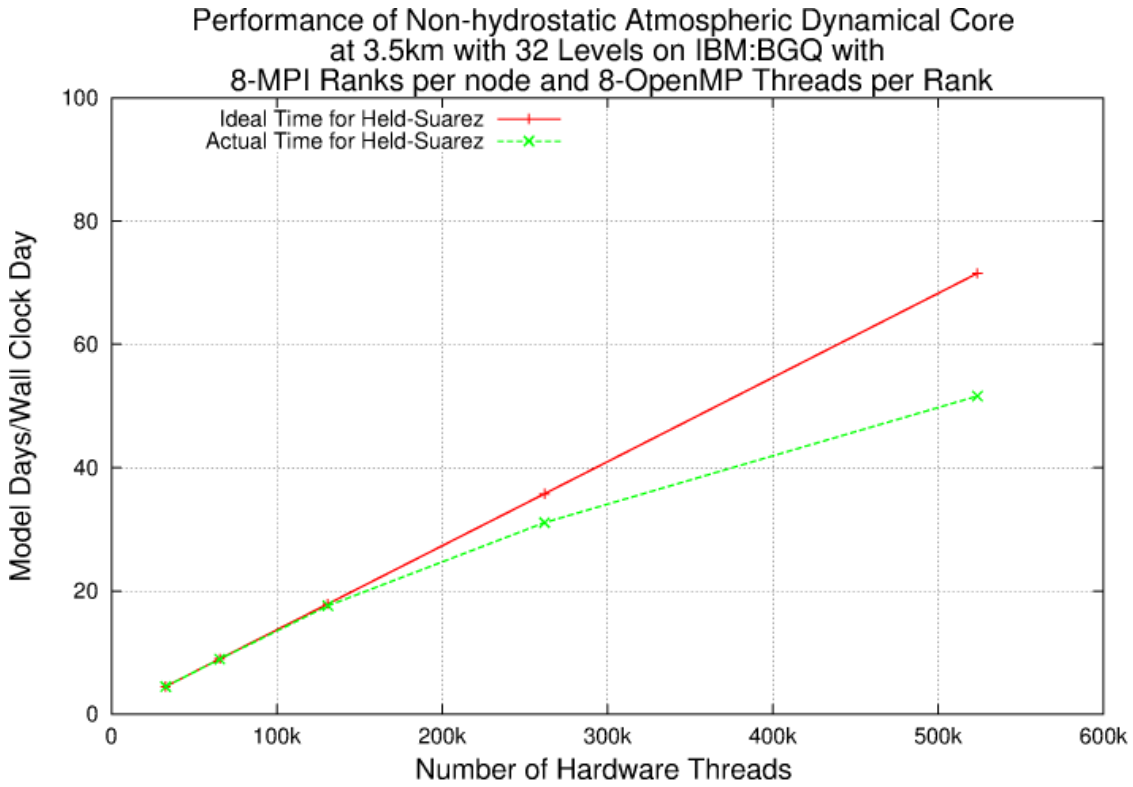


Figure 3.1

The data further show that the communication time is completely dominated by calls to MPI_Wait. Plotting the time spent in MPI against a projection onto the underlying cubed-sphere grid produces fascinating but difficult to interpret patterns. These patterns suggest that sub-classes of processes may behave in correlated and perhaps self-reinforcing patterns any of which may contribute to the scaling limitations. A graphic with the wait time provided by Chris and Bob Walkup at IBM is presented below (Figure 3.2):

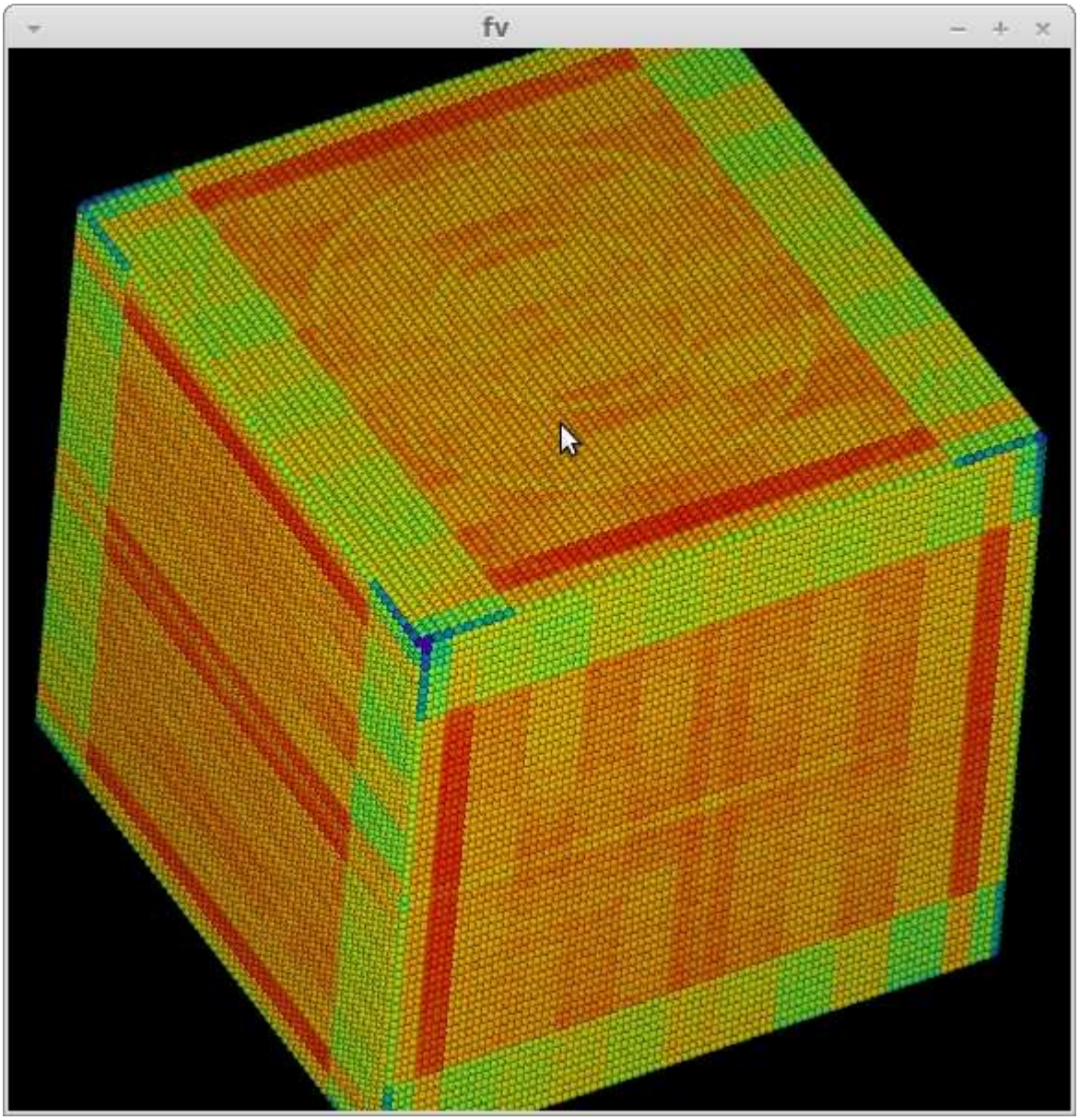


Figure 3.2

Assessment Going Into the Project:

- The MPI communication diagram (largely dominated by MPI_Wait) is actually an aggregate over the entire program run. We need to gain deeper understanding of the communication classes and how they contribute to the aggregate.
- Both communication and computations load imbalance have been examined. While the imbalance in floating-point instructions was determined to be about 5% across the ranks, the wait time varies by ~50%. There are clearly additional factors. What are the root cause(s)?
    - Examine the possible roles of system hardware and software in the communication features.
- Once the underlying mechanisms have been identified, how might the strong scaling performance of the code be improved?

3.2 Methodology

The Argonne BG/Q platform supplies a number of tools for the acquisition of data related to application performance. The project selected

- mpitrace: A library for collecting and providing ascii file output of application message passing characteristics.
- mpihpm_smp: An OpenMP compatible library for collecting and providing ascii file output of a selected set of hardware counter information along with selected metrics derived from the raw counter data.
- hpmprof: An OpenMP compatible library for collecting and providing ascii file output of a user selectable hardware counter at a user selectable frequency profiled by subroutine. The library provides resolution down to subroutine source line number.

In addition, the project employed a number of additional analysis tools

- A database with web front end developed previously by Samara Technology Group to enable text and graphic comparison of hardware counter data across run configurations for scaling analysis.
- Ad-hoc, spreadsheet based analysis of cross configuration performance features.

In an effort to develop novel approaches for the analysis of large, complex application performance related datasets, Samara also teamed with Dr. Haimonti Dutta of the Columbia University Center for Computational Learning in an effort to apply Machine Learning techniques. Results from all of these approaches are documented in this report.

4 Findings

4.1 Scaling Results

Among the first tasks was to repeat the scaling results of Kerr et al. To this end, we measured the wallclock performance of the 3.5km Held-Suarez main loop on 1, 2, 4, 8 and 16 racks of Mira. This excludes the expensive initialization and the small but performance irrelevant restart write. Figure 4.1 depicts Models Days per Computation Wallclock Day on configurations from 1 to 16 racks as measured for this project:

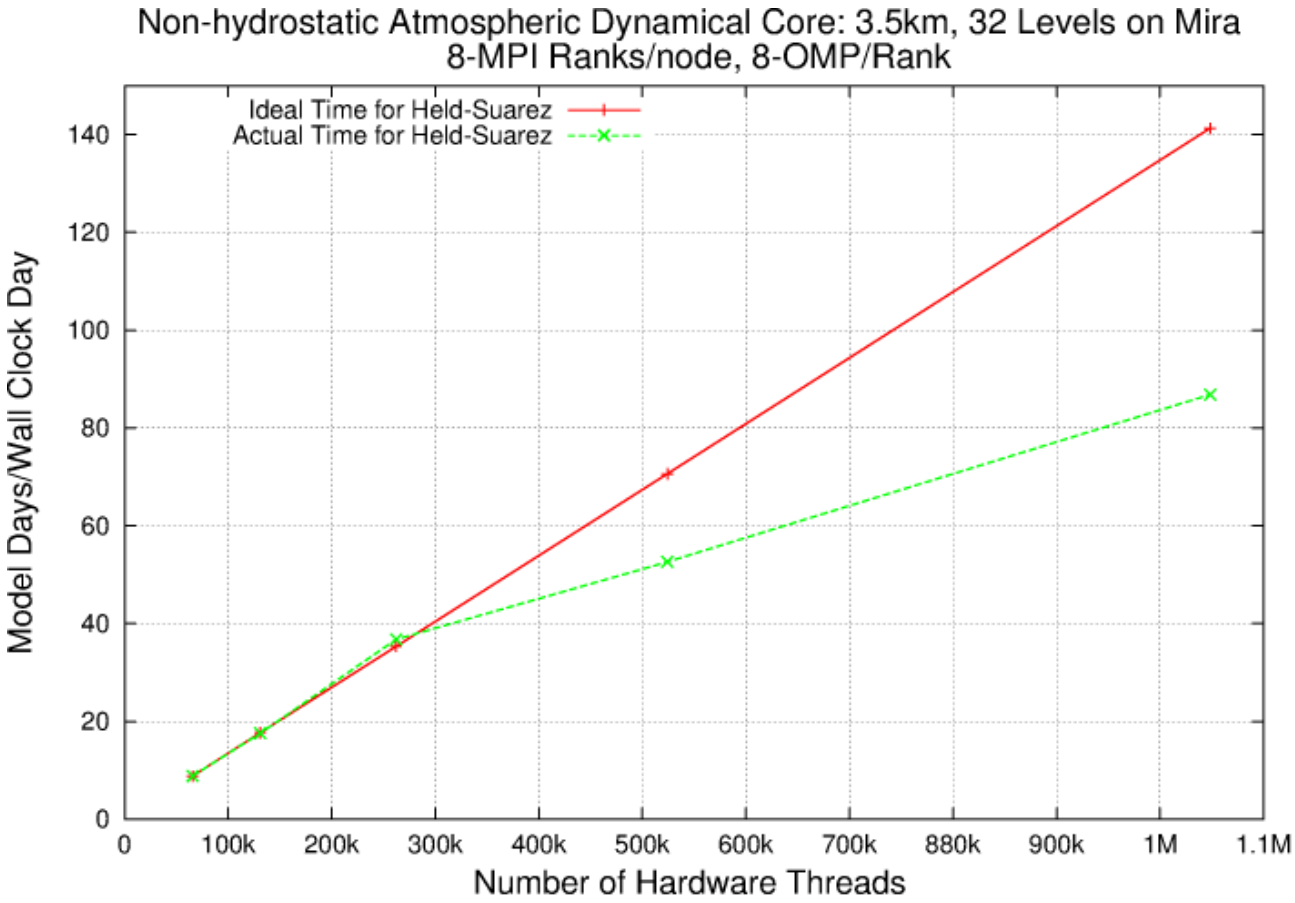


Figure 4.1

There are two particularly interesting features apparent in the data. The first is the principle topic of this work: The loss of scaling performance at extreme scale. But a second feature found with some layout optimization work (*i.e.* finding optimal MxN subdomain decompositions) is the slightly superlinear speedup going from 2 to 4 racks (131K to 262K threads). Follow on analysis made the source of both of these features very clear.

4.2 Initial Analysis

The principle hypothesis concerning the scaling behavior posited issues associated with communication performance. There was a lot of reason to believe this view. Just on general principles, this is the first time the communication infrastructure supporting the model[2] has been run at such scales. As communication performance is so intimately linked with scaling behavior, it is quite reasonable to wonder whether the FMS infrastructure was encountering some previously unknown internal limitation. Certainly earlier work on the Argonne BG/L and P had demonstrated previously unknown application scaling limitations.

At the outset, it was known that the FMS communication library implements a non-blocking communication model, posting non-blocking receives before buffering data and then executing the non-blocking sends. But a key limitation of FMS is that it supports only a single communication in-flight between any MPI two ranks. If a second communication becomes necessary between two

2 The FMS MPP library that contains all information concerning the rank to rank communication topology as well as wrapping the calls to MPI.

MPI ranks, the implementation blocks until the first send/recv pair is satisfied.

It was known that the edges and corners receive special treatment that requires communication in addition to that on the interior of the cube face. Was it possible that these large scales with a processor of substantially different architecture than Intel x86 and a model with over 150K MPI ranks was revealing some sort of complex chaining behavior? As noted in the background material of Section 2.1, the communication patterns as depicted in Figure 3.2 suggest sub-classes of processes that may behave in correlated and perhaps self-reinforcing patterns any of which may contribute to the scaling limitations.

Such basic observations of model scaling and subsequent questions are generally the motivations for performance analysis research. But as becomes clear throughout the rest of this report, initial observations and prior knowledge notwithstanding, one *must* keep an open mind. As the data and analysis of this report clearly demonstrate, *the causes of the scaling deterioration for the model are in fact rooted in phenomena associated with the compute portions of the application; not communication*. Further, approaches that rely on *halo* or *ghost cell* regions[3] (as do virtually all grid based approaches in a distributed, parallel environment) have characteristics that can *induce anti-scaling behavior* (*i.e.* performance components that increase with increasing MPI rank count).

First we examine the base timing data to understand communication behavior. One of the most striking features is that virtually all of the time attributed to MPI is MPI_Wait and MPI_Allreduce. The following Table 4.1 summarizes the Main Loop time and the Communication time associated with the Main Loop:

8 threads		Sim days / day	Main Loop sec	Comm sec	Comm Time Breakdown sec	
	1 Rack	8.8	3254.7	431.0	8.4 / 2.1	ISend/IRecv
					143.0	AllReduce
					277.5	Wait
	4 Racks	36.8	782.9	65.8	7.2 / 1.8	ISend/IRecv
					7.7	AllReduce
					49.0	Wait
	16 Racks	86.8	331.7	58.7	5.4 / 1.2	ISend/IRecv
					7.2	AllReduce
					45.0	Wait

Table 4.1

It is clear from Table 4.1 that the (non-blocking) send and receive operations take only a small portion of the communication time.

The MPI_Wait time is indicative of load imbalance across the MPI ranks. Further while not represented in the table, 88.5% of the Allreduce operations are on a single, double-precision value while the remaining 11.5% are a vector of 32 double precision values. Since the BG/Q communication network is very high bandwidth and low latency, the Allreduce times are also attributable to load imbalance.

But it is not the load imbalance that leads to loss of scaling performance and there is a very simple way to see that the real problem lies with loss of scaling in the computation. Subtracting the Main

3 For more detail on ghost cells, see for example http://parlab.eecs.berkeley.edu/wiki/_media/patterns/ghostcell.pdf.

Loop communication time from the Main Loop itself, we arrive at 2823.7s for a single rack. Simply dividing 2823.7s by 16, we arrive at the perfect scaling value of 176.5s. Calculating the Main Loop compute time as measured for 16 racks produces 273s. Thus, the measured Main Loop time at the scaling extreme is almost 1.55x that of perfect scaling. One or more phenomena are clearly having a profound negative impact on computational component.

At this point in the study, we were asked to update the source code base and specific test problem to bring it in line with the code base being used in a broader set of studies that include Xeon Phi, Xeon Sandy Bridge and at some point, perhaps nVidia GPGPUs. But before leaving the datasets associated with the original code base, a table of some hardware counters as measured by the mpihpm_smp library is provided below:

C2560 NonHydrostatic HS						
	Sim-day /		FP Ops /			DDR
racks	Wall-Day	speedup	Sim-hr	%FXU	IPC/core	Bytes/Cyc
1	8.83	1.0	4.63E+15	60.9	0.77	13.2
4	36.78	4.2	4.92E+15	63.1	0.85	9.5
16	86.84	9.8	5.63E+15	66.7	0.75	6.0

Table 4.2

Among the most striking features of the data is the *increase* in FP operations (over 20%) as one scales out. We will revisit this phenomenon in some detail later in the report. Another interesting feature is that even with the increase in FP operations, the number of non-FP operations (based on the %FXU) is also increasing. Finally, the scaling of the IPC/core is also quite interesting. We can guess that decreased memory traffic is a significant factor increasing the IPC/core going from 1 to 4 racks. What then reduces the IPC/core when scaling from 4 to 16 racks?

4.3 Detailed Results

4.3.1 The New Test Configuration

As mentioned in Section 4.2, we were asked to update the source code base and specific test problem to bring it in line with the code base being used in a broader set of studies. Aside from any source code changes, the test problem was moved from the non-hydrostatic Held-Suarez case to the hydrostatic Held-Suarez case. It is not surprising that the non-hydrostatic case is computationally significantly more expensive. Nevertheless, the performance features of the hydrostatic case are similar enough to remain generally congruent to the original problem.

The results of Section 4.2 also strongly suggest that MPI is not the sole nor even major source of scaling inefficiency. We conclude from this that we are in search of efficiency losses within the computational performance. Thus, it should be sufficient to apply weak-scaling principles and study a test case with a smaller global domain while maintaining the per-rank working set size by appropriate choice of core count and layout.

Referring to the time spent in MPI as depicted by Figure 3.2, some early work also clearly demonstrated that contours of the MPI cost graph were closely associated with the number of

subdomain points assigned to the MPI rank. It should be noted that for the original problem, one of the goals for decompositions used in the analysis was to fill as many of the nodes as possible for a given rack count. Given the 6 faces to the cubed-sphere, the C2560 grid, the empirically determined 8 OpenMP threads per rank and the constraint to maximize residency at a given rack count leads to some interesting decompositions. Trial and error found that configurations closest to square gave the best results. Consequently, the MPI decompositions for the C2560 results reported here were (X-ranks x Y-ranks) 39x35, 78x70 and 156x140 for 1, 4 and 16 racks, respectively[4]. Since 2560 is not an integral multiple in either direction of any of these rank counts, there is substantial variation in subdomain point count. This makes the minimal impact of MPI load imbalance on scaling as analyzed in Section 4.2 all the more interesting.

In searching for a reduced size global grid, we note that differing subdomain point counts is an obvious source of MPI load imbalance. Again referring to the results of Section 4.2, we chose to study a global grid size allowing symmetry between the X and Y decompositions. A factor motivating the choice for the study grid was noting that in some average sense, the subdomain sizes for the C2560 performance tests were approximately 72x72, 36x36 and 18x18 for 1, 4 and 16 racks, respectively. Combining this with the desire to minimize the resources required to run the many data gathering experiments led to the choice of a 288x288 global grid. While not a unique solution, it certainly satisfies the desired constraints. Eight threads per rank (per the original problem) and subdomain sizes of 72x72, 36x36 and 18x18 lead to node counts of 12, 48 and 192, respectively[5].

The Figure 4.2 below depicts the scaling performance of the 288x288 demonstrating that from a scaling perspective, it is a reasonable surrogate:

4 Corresponding to 8190, 32760 and 13140 MPI-ranks.

5 Corresponding to 96, 384 and 1536 MPI-ranks.

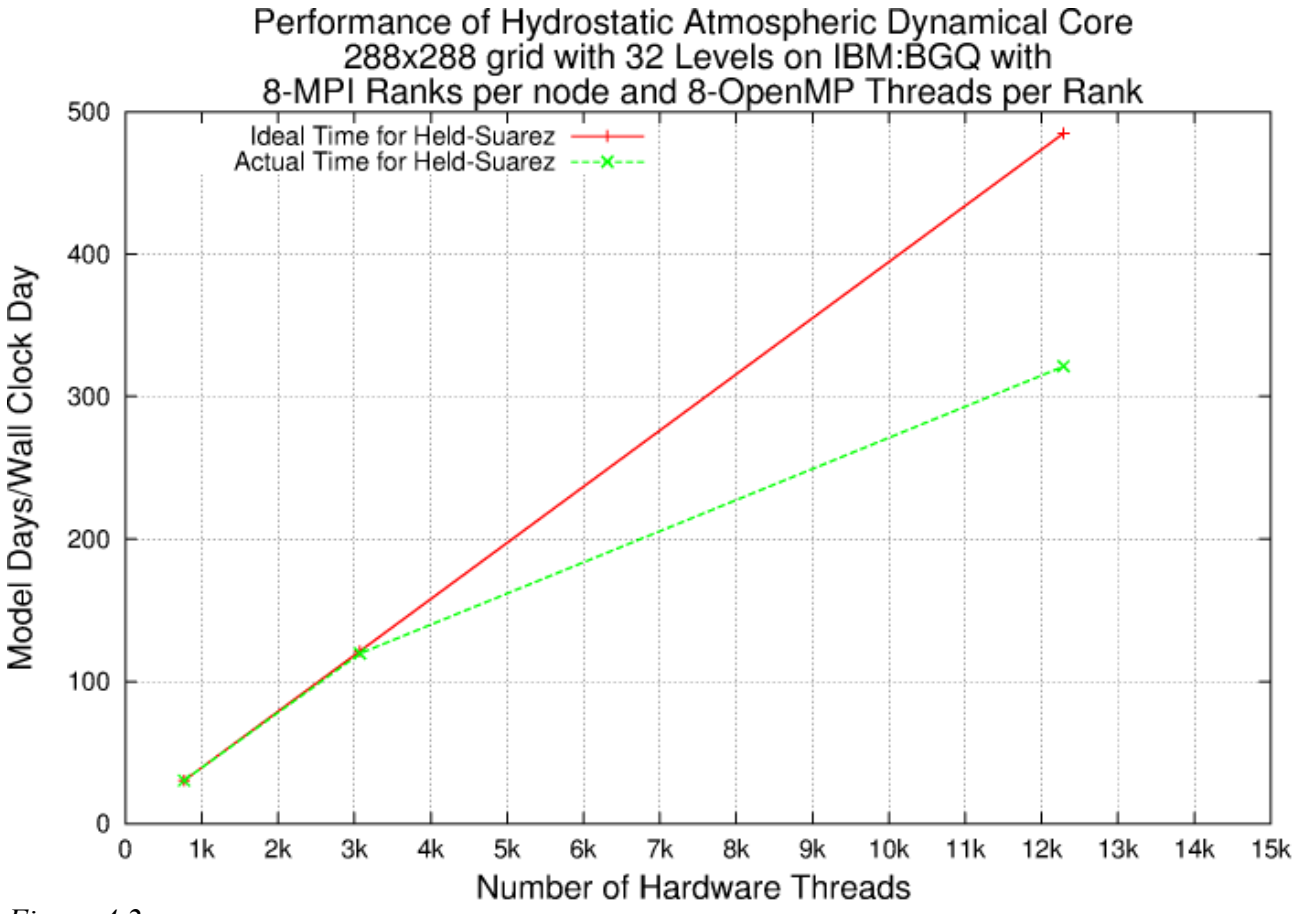


Figure 4.2

In summary, after the initial data gathering outlined in Section 4.2, we were asked to update the code base and change test case for consistency with ongoing performance analysis across multiple, highly threaded architectures. Supported by the results of Section 4.2, we used the principles of weak-scaling to choose a substantially smaller global grid. We maintained a notion of the average subdomain point count. The new test case is hydrostatic while the original case is non-hydrostatic. Still the general shape of the performance features remains congruent to the original problem. It should be noted that we loose the superlinear speedup originally present going from 1 rack to 4 racks. As the effect was small and goal is to understand performance at full scale (*i.e.* equivalent to 16 racks), the loss is inconsequential to the analysis. The details described below bear this out.

4.3.2 Understanding Performance Through Globally Summed Hardware Counts

This report will not cover the hardware counter libraries in any detail. There are excellent descriptions of the tool sets elsewhere[6]. This report will discuss how the data from the hardware counters has been applied.

While the libraries are capable of outputting per MPI rank hardware counter information (and indeed even some data on a per thread basis), the shear volume and complexity of the data makes it essential to try to first determine what drives performance from a global perspective. Towards this end, we (and others) have applied the following techniques successfully.

6 See for example IBM public documentation "IBM System Blue Gene Solution Blue Gene/Q Application Development" as well as /home/walkup/mpi_trace/bgq/MPI_Wrappers_for_BGQ.pdf on Mira and Vesta.

For a simple case, consider a parallel application with a Single Program / Multiple Data, MPI programming paradigm and further, consider a profile of the cycles spent in the various subroutines summed across all ranks for a given run.

Under perfect scaling in this highly simplified example, the global sum of the total program cycle counts across all the ranks is a constant. Further, this is true at the procedure level as well.

But we do not live in such a simple world. First and perhaps most important, the working set size changes as one scales out. Thus the application running on N-MPI ranks is likely to have a very different interaction with the memory hierarchy than when running on 4N or 16N ranks. Nor is the subroutine profile likely to remain constant[7].

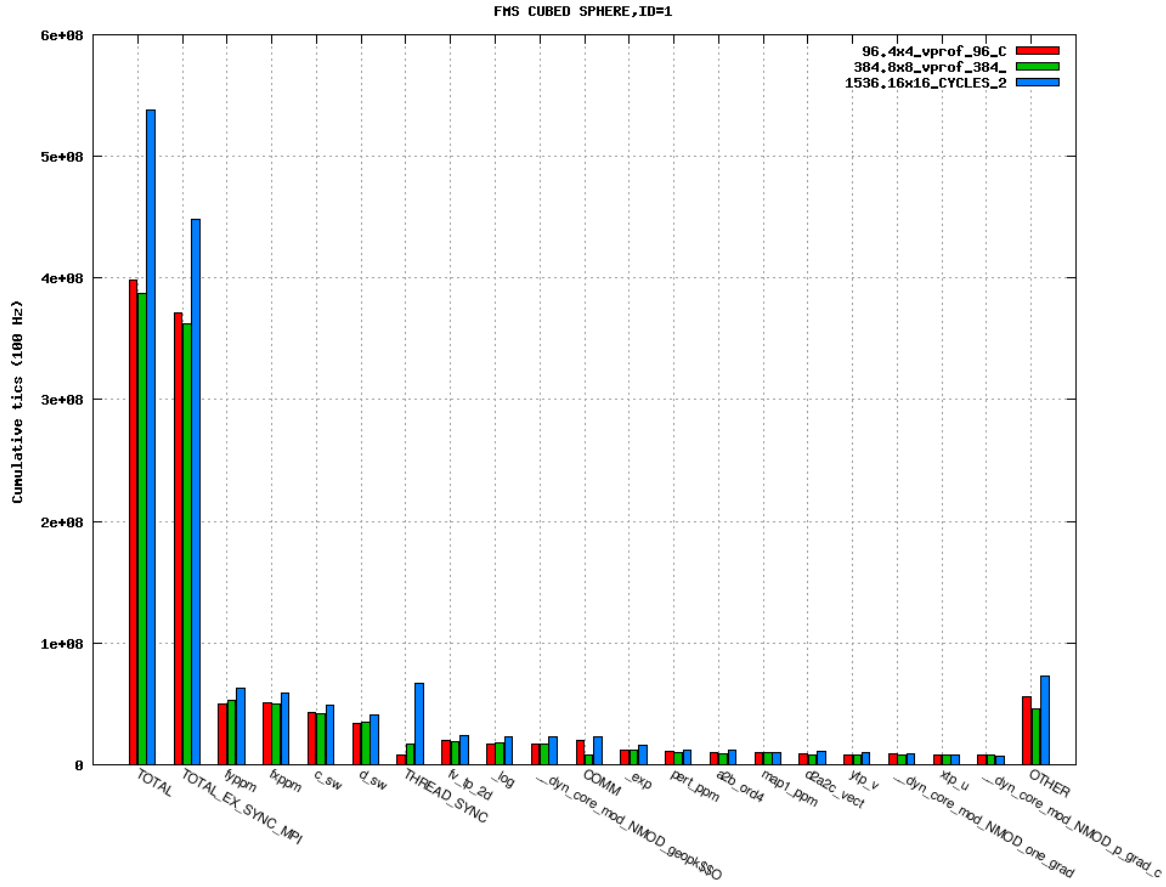


Figure 4.3

The graph in Figure 4.3 depicts the cycle count on a per subroutine basis for the MPI layouts 4x4, 8x8 and 16x16 as summed across *all ranks*[8] for the 288x288 grid. As one can readily assess, things are not constant as one scales out. Further, one can readily pick out that OpenMP thread load imbalance costs increase as one scales out. But is the apparent thread load imbalance sufficient to explain the loss in scaling? It should be noted that the version of HPMPROF that provides the data for this graph is limited to an interrupt for Thread-0. With a total of 8 OpenMP threads for each

7 This is why it is essential to use test cases that mimic if not replicate production problem working set sizes. One can easily *solve the wrong performance problem* with test cases that are too small or over-simplified.

8 For this paper, the terms *cycle* and *cycle count* will refer to the globally summed value unless otherwise explicitly specified. The terms wallclock and runtime all refer to the measured unit of time to solution.

rank, we are getting only a limited view of the thread performance world.

The utility of global counts for all of the hardware counters is as useful as cycles. These other hardware counters such as completed instructions through the main execution unit (XU instructions), the number of completed FP instructions (executed through the auxiliary instruction unit – the AXU – on the BG/Q processor), cache misses, etc all provide information on what's causing the observed performance. Aggregating the values across ranks produces a top-down perspective[9].

Graphs are but one way to gain insight into the global data. The table below summarizes the hardware counter values across layouts for the 288x288 grid with working set sizes equivalent to 1, 4 and 16 racks of the original c2560 grid (96, 384 and 1536 MPI ranks for the 288x288, respectively). Note not only the increase in AXU instructions[10] with increasing core count, but the increase in XU instructions as well.

288x288: All						
Pts per face	72x72	36x36	18x18			
MPI face layout	4x4	8x8	16x16			
Ranks	96	384	1536	384 / 96	1536 / 384	1536 / 96
Total Cycles	4.55E+14	4.65E+14	7.04E+14	1.02	1.51	1.55
Load Miss	1.56E+13	1.55E+13	1.85E+13	1.00	1.19	1.18
Cacheable Loads	2.30E+14	2.60E+14	3.56E+14	1.13	1.37	1.55
L1p Miss	3.92E+12	3.97E+12	6.99E+12	1.01	1.76	1.78
XU Inst	4.93E+14	5.93E+14	9.35E+14	1.20	1.58	1.90
AXU Inst	3.05E+14	3.25E+14	3.66E+14	1.07	1.13	1.20
XU / AXU	1.62	1.83	2.55	1.13	1.40	1.58
FP	3.90E+14	4.15E+14	4.66E+14	1.06	1.12	1.19
L2 Hit	5.17E+14	5.74E+14	7.98E+14	1.11	1.39	1.54
L2 Miss	2.40E+13	2.06E+13	1.73E+13	0.86	0.84	0.72
FPU %	38.19	35.36	28.15			
FXU %	61.81	64.64	71.85			
Bytes/cyc	11.587	8.785	4.738			
IPC	0.88	0.99	0.92			
Tot GF	131.65	547.65	1626.00			
% Loads that hit in						
L1D cache	93.21	94.02	94.81			
L1p Buffer	5.08	4.45	3.22			
L2 Cache	0.40	0.53	1.36			
DDR	1.30	0.99	0.61			

Table 4.3

It is also interesting to note that XU instructions outnumber AXU instructions from 1.6x to 2.6x across the scaling range. Since the BG/Q core is capable of executing an AXU as well as an XU

9 See “Beyond the CPU: Hardware Performance Counter Monitoring on Blue Gene/Q”, McCraw et al for more detail on the BG/Q processor as well as the hardware counter infrastructure.

10 The Auxiliary Execution Unit (AXU) issues the FP instructions (and only FP instructions). The FP count produced by the HPM and HPMPROF libraries is an attempt to apply weights to the actual FP instruction. For example, SIMD instructions producing more results per cycle are weighted accordingly.

instruction with each cycle, it seems clear that the XU instructions will dominate performance[11].

As it turns out, some relatively simple call chains execute most of the total runtime cycles. These call chains are the “D-grid shallow water” (D_SW), the “C-grid shallow water” (C_SW) and the “Geopotential” (GEOPK). These chains are all OpenMP enabled, contain no MPI communication and expend 61%, 61% and 49% of the cycles for the 4x4, 8x8 and 16x16 decomposition, respectively. Time spent in MPI from other parts of the code (again, almost entirely in wait operations) is 4.9%, 2.0% and 4.3% again across the scaling range 4x4, 8x8 and 16x16.

OpenMP overhead and load imbalance are somewhat more difficult to ascertain. Two approaches were applied to the D_SW, C_SW and GEOPK call chains. For the first approach, XU, AXU instruction counts were measured in each call chain using HPM. Recall that HPM aggregates per core counter values across all threads. In separate experiments, HPMPROF was used to collect the data within the main loop for a specific counter. Note that the version of HPMPROF used here records the value for Thread-0 only.

The HPM global count for a particular counter was compared against a Thread-0 global count derived from HPMPROF data.

The key to the analysis is that the HPM data contains all routines in a particular call stack while the approach using HPMPROF simply selects the FMS source routines in the call stack. Thus, HPM data contains all OpenMP overheads while HPMPROF does not. To the extent that the Thread-0 XU and AXU data exemplifies the true average across threads, the HPMPROF counts will be 1/8 the HPM counts. The different approaches yield strikingly congruent results for the D_SW and C_SW call chains while the comparison yields discordant results for GEOPK. Luckily GEOPK comprises only about 7% of total cycles across the scaling range.

Later in the project, a hardware profiling library that sums over the threads (referred to from here forward as HPMPROF_smp) was provided by IBM[12]. While there are several caveats when attempting to perform interrupt driven profiling in a multi-threaded environment, HPMPROF_smp worked extremely well for the calipered experiments being performed. The results from this series of data was consistent with the notion that the D_SW and C_SW call chains contain very low amounts of OpenMP overheads while GEOPK has a rather high percentage of cycles associated with OpenMP. For more detail, see Appendix B.

4.3.3 OpenMP Overheads

While Section 4.3.2 notes that OpenMP overheads are low for the D_SW and C_SW call chains, further work with HPMPROF profiling reveals that cycles spent in OpenMP related functions escalate elsewhere in the application. The following table summarizes the results from HPMPROF (records thread 0 only) for cycles and XU instructions across all MPI ranks. The counts have been binned into categories. The categories FMS and MPP are application code. Here we've specifically broken out the cycles spent in the FMS MPI library wrapper. Virtually all of the MPP cycles are devoted to data movement. Further, the counts attributed to FMS also include runtime math functions such as log and exp. The message passing portion of the counts have been further decomposed into symbols associated with the underlying MPICH implementation and the BG/Q Parallel Active Messaging Interface (PAMI)[13]. The final categories were cycles associated with

11 XU instructions almost undoubtedly dominate performance as they contain the load and store operations as well as address calculation.

12 As with many aspects of this project, we are deeply indebted to Bob Walkup. IBM TJ Watson Research Center for valuable discussions and ad hoc tool tinkering.

13 Note that the PAMI counts also include the Message Unit Programming Interface which were small in number in

OpenMP, the IBM XL compiler components not associated with the OpenMP implementation and miscellaneous.

CYCLES								Count Ratios	
ALL	430062011	435254051	653187860					8x8 /	16x16 /
	4x4	8x8	16x16		4x4	8x8	16x16	4x4	8x8
fms	350846079	351744324	419909026	fms	81.58%	80.81%	64.29%	1.00	1.19
mpp	9755165	20533757	52617483	mpp	2.27%	4.72%	8.06%	2.10	2.56
omp	10147590	22793135	80750098	omp	2.36%	5.24%	12.36%	2.25	3.54
mpi	6499407	7381086	22731636	mpi	1.51%	1.70%	3.48%	1.14	3.08
pami	24289231	18103732	46002166	pami	5.65%	4.16%	7.04%	0.75	2.54
xl	27870178	12348854	22877831	xl	6.48%	2.84%	3.50%	0.44	1.85
misc	654361	2349163	8299620	misc	0.15%	0.54%	1.27%	3.59	3.53

Table 4.4

The OpenMP profile was analyzed in some detail:

omp cycles	4x4	8x8	16x16
SpinWaitTaskSwitchBGQ	58.40%	34.44%	22.51%
xlsmpGetDefaultSLock	16.41%	36.52%	50.70%
pthread_cond_signal	5.80%	5.53%	5.58%
sched_yield	5.21%	4.08%	2.40%
xlsmpParallelDoSetup_TPO	2.66%	3.87%	3.71%

Table 4.5

We can see that OpenMP emerges as a substantial component with unpleasant anti-scaling characteristics and we know that these cycles are not associated with the primary computation call chains. Unfortunately at this point, we have no particularly effective techniques for tracing where these OpenMP cycles are coming from other than to play hide and seek by calipering a particular code section and hoping to find hits. Interesting in the OpenMP profile is the emergence of the xlsmpGetDefaultSLock. This routine receives no counts in any of the D_SW, C_SW and GEOPK call chains. In particular, its absence in GEOPK likely means that whatever effects leading to the increased OpenMP overheads for GEOPK do not explain the more general increase in OpenMP cycles.

4.3.4 Performance Drivers

FP Updates to Halo Values

Referencing Section 4.2, the increase in AXU instruction count as MPI rank count increases is one of the more interesting features of the hardware counter data. While an increase in AXU count seems counter-intuitive, a little thought provides an obvious mechanism. One of the MPI scaling phenomena is that the ratio of halo point count to points interior to the subdomain increases. Put another way, only the calculations and updates on values in the subdomain interior (the so called *compute domain*) move the simulation forward. Calculations leading to updates in the halo region are redundant. While redundant calculations are often preferable to communication, it is a fact that they do not in and of themselves produce forward progress for the simulation.

The OpenMP implementation in the FMS cubed-sphere dynamical core has the following structure:

any event.

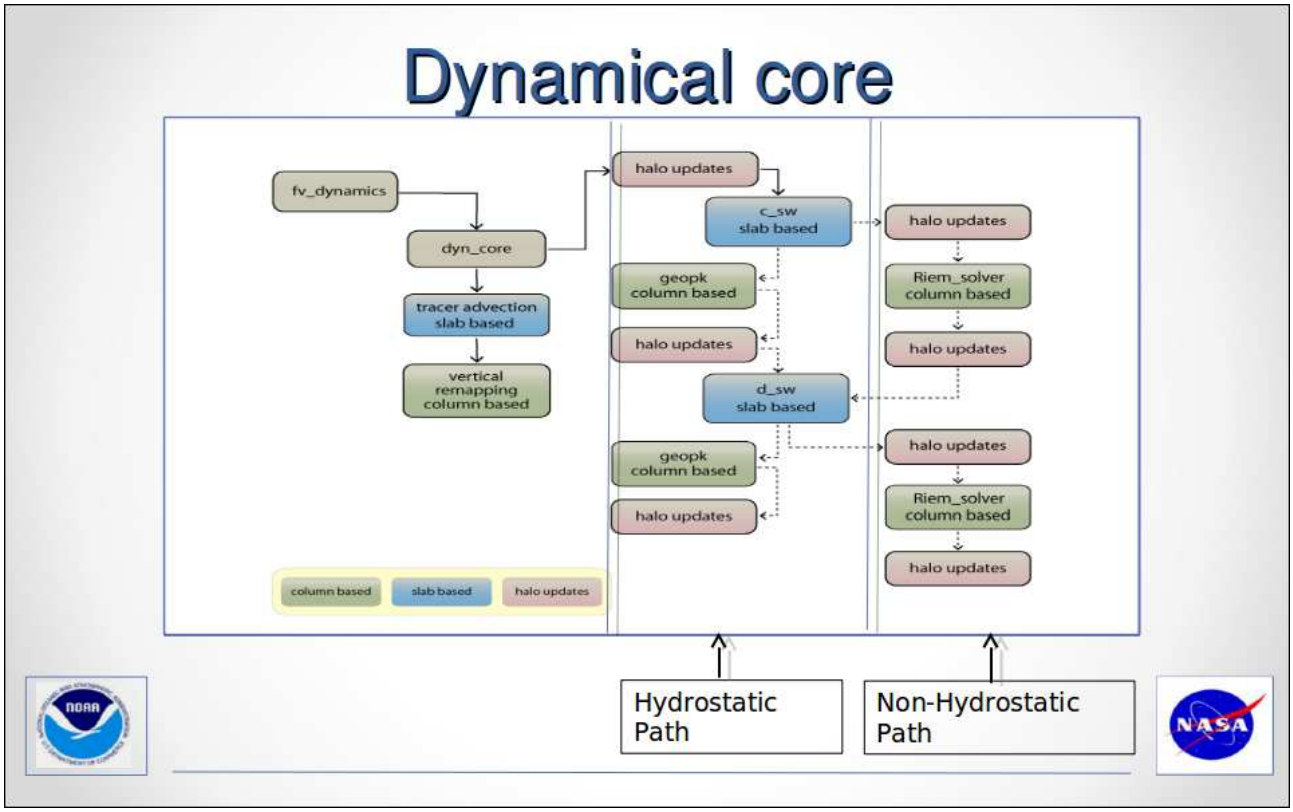


Figure 4.4

As depicted by the Figure 4.4 shaded sections, many of the calculations are across a "slab" (*i.e.* the X and Y directions) with OpenMP along the Z axis (NZ=32 for this case). The numeric operators used by the cubed sphere require up to 3 points in each of these spacial directions and thus each subdomain requires a halo of size 3. A simple calculation shows how quickly the halo points dominate any derived quantities that need values in the halo region. Consider the reduced grid with 288x288 points per face. Per the discussion of Section 4.3.1, 4x4, 8x8 and 16x16 MPI-ranks per face mimics the C2560 on 1, 4 and 16 racks, respectively. In turn this produces subdomain sizes of 72x72, 36x36 and 18x18. Including a 3-point halo, the so-called *data (sub)domain* for each rank is 78x78, 42x42 and 24x24, respectively. Thus for a decomposition of 4x4, the halo region is about 15% of any computation that updates the entire data domain. On the other hand, the halo region is almost 44% of the data domain at 16x16 ranks per face. More importantly, the total number of grid points to compute (*i.e.* the point count summed across all data domains) escalates from 97.3K pts per face at 4x4 to 147.5K pts per face for any computation that includes calculating values for the entire halo region. The increase in the total number points for any computation updating the entire data domain across the range representing 1 to 16 racks is over 50%!

Luckily relatively few computations call for operations that update the halo region and then, generally not the entire region. Nevertheless, the phenomenon is clearly discernible in the hardware counter data. The following is a sample taken with HPMPROF using a kernel based on a prominent subroutine in the run time profile (fxppm). For this section of code, the number of levels (NK=32) is the OpenMP axis and the loop is farther up in the call chain. The test is setup to run the call to fxppm 16 times more iterations for a subdomain size of 18x18 than 72x72. This mimics the (idealized) conservation of FP operation count across the global domain.

```
do j=js,je
  do i=is-2,ie+2
    xt = 0.25*(q(i+1,j) - q(i-1,j))
    dm1(i) = sign(min(abs(xt), max(q(i-1,j), q(i,j), q(i+1,j)) - q(i,j), &
                  q(i,j) - min(q(i-1,j), q(i,j), q(i+1,j))), xt)
  enddo
.
.
.
```

For this case, js and je are simply the J limits of the compute subdomain (*i.e.* no Y halo). HPMPROF (setup to profile FP operations) hits some component of this loop 13088 and 15150 times for the 4x4 and 16x16 decompositions, respectively. Simple division shows us there are approximately 1.16 times more FP operations for 16x16 than 4x4[14]. On the other hand, the inner loop updates 2 of the halo points on each side in the X direction.

$$(22*18)*(16^2) / (76*72)*(4^2) = 1.16$$

The agreement with theory is startling and may seem like good luck as much as anything else since the data is acquired by statistical sampling. Still this loop represents one of the most expensive elements of the subroutine and so should build the best statistics. Counts from other, less expensive loops reflect the tendency described here, but not the detailed values of theory.

Unlike HPMPROF which profiles code based on a particular hardware counter overflow, the HPM tool simply captures the values of particular counters between start and stop points. HPM applied to the fxppm kernel verifies the bloat in FP operations as one scales out and this is consistent with the halo spanning loop structures of fxppm.

Beyond Floating Point Operations

While FP operations (AXU instructions) are a measure of the computational work being done, it must be remembered that a substantial set of operations must be performed in order to get the FP operands into place for the computation. Thus given the nature of the RISC instruction set, it is not surprising that non-FP operations (XU instructions on BG/Q) outnumber FP operations by a rather large margin. Referring to the tables in Appendix A, we see that only for the computationally densest part of the dynamical core (the D_SW call chain) is there relative parity between XU and AXU instruction counts.

What are the drivers for the XU operations? Since the growth of XU instructions outpaces the growth of AXU (FP) instructions, one might wonder whether the XU instructions are associated with some breakdown in OpenMP efficiency, with the dycore computations in the call chain or with something as yet unknown. The HPM and HPMPROF_smp tools allow us to answer this question in some detail.

We can gain a top-down perspective from the XU instruction counts decomposed as with cycles in Table 4.4

14 A similar experiment was performed with AXU instructions and produced similar results.

XU

								Count Ratios	
ALL	70672412	73963029	119300459					8x8 /	16x16 /
	4x4	8x8	16x16		4x4	8x8	16x16	4x4	8x8
fms	53518822	58295094	72234995	fms	75.73%	78.82%	60.55%	1.09	1.24
mpp	1310255	3754271	11990508	mpp	1.85%	5.08%	10.05%	2.87	3.19
omp	1075382	3467290	14656405	omp	1.52%	4.69%	12.29%	3.22	4.23
mpi	2128409	1214499	3278641	mpi	3.01%	1.64%	2.75%	0.57	2.70
pami	7448656	4632589	11403841	pami	10.54%	6.26%	9.56%	0.62	2.46
xl	5119204	2306978	4561084	xl	7.24%	3.12%	3.82%	0.45	1.98
misc	71684	292308	1174985	misc	0.10%	0.40%	0.98%	4.08	4.02

Table 4.6

Among the first things to note is that the shape of the categorized XU counts is similar to the shape of the cycle counts. It seems reasonable to assert that the major source of increase in XU and hence cycle count is driven by the increased count of halo points as manifested in the fms and mpp categories. There is also a substantial component that emerges from OpenMP (specifically xlsmpGetDefaultSLock). Previous analysis has demonstrated that these instructions are not associated with the main computational components D_SW, C_SW and GEOPK.

As concerns the main computational components, we can demonstrate that the HPM and HPMPROF_smp tools are giving consistent results. Subtracting the load operations from the XU count, one can begin to deconstruct the types of operations being performed. Using HPM calipered around the D_SW, C_SW and GEOPK regions, we obtained the counter data in Appendix C which provides a breakdown of the XU instructions in terms of loads, stores, branches and miscellaneous "other" (integer arithmetic mostly likely address calculation, bit manipulations, register moves, etc). As a check, we also ran HPMPROF_smp over the D_SW region for branches and stores and projected a global count for 1 simulation day. A comparison of values derived from the two approaches is provided in the table below.

D_SW				
	HPMPROF_smp		HPM	
	16x16	8x8	16x16	8x8
BR	4.44E+13	3.63E+13	4.58E+13	3.71E+13
ST	4.65E+13	3.88E+13	4.53E+13	3.82E+13

The results are within about 3% of each other.

4.3.5 Conclusion

We have used various hardware counters summed across all MPI ranks as probe into the resource utilization characteristics that drive the scaling behavior of the GFDL FMS 3.5km High-Resolution Atmosphere model on the Argonne BG/Q platform. Early in the process we determined that MPI performance per se plays no role in the scaling characteristics. Based on this result, we applied the principle of weak scaling to study a much coarser resolution grid on layouts that preserve the per rank working size.

The hardware counter data and analyses clearly demonstrate that calculations for derived variables associated with the halo region are inherently non-scalable since the ratio of halo to compute domain points grows ever larger with MPI rank count. Performance enhancement through cache effect induced memory footprint reduction via scaling can offset instruction count growth as it does

in the transition from the 4x4 decomposition to the 8x8 for the 288x288 grid (equivalent to the 1 to 4 rack transition for the original C2560 case). But such transitions are limited in number and effect leaving the behavior seen from the 8x8 to the 16x16 decomposition much more likely. The "Halo Effect" mechanism drives over 70% of the XU instructions and hence cycles at full scale.

The emergence of substantial cycles associated with OpenMP overheads plays a secondary role in performance at scale. We have demonstrated that these OpenMP cycles are not associated with D_SW and C_SW and that their link to GEOPK is completely insufficient to be the root cause of the count expansion. On the other hand, the proportion of cycles associated with these three components has diminished from 60% of total cycles for the 4x4 and 8x8 decompositions to 50% for 16x16. This fall mirrors the rise of the OpenMP overhead cycles like from the remaining 50% of FMS.

Perhaps the pivotal thing to understand is the distinction between processor performance verses scaling performance. Instructions per cycle (IPC) is the key metric for processor performance and the BG/Q processor core is capable of concurrently executing an AXU and an XU instruction per cycle[15]. So far as concurrency of the AXU and XU is concerned, it turns out that even within the most compute dense portion of the code (the parallel region around D_SW), the IPC is only 1.12 to 1.18. Over the entire code, the IPC ranges from about 0.9-1.0.

With respect to instructions executed on the *computational domain* (the portion that actually moves the simulation forward), we note that regardless of number, the potentially expensive[16] cache misses get distributed over increasing numbers of MPI ranks and so their net effect on wallclock time should diminish linearly[17]. And this is true of the per rank cost of all operations on the computational domain and this fact is fundamental to the notion of scaling.

This leads to the assertion that the loss of performance can only come from increased instruction count and reduced OpenMP efficiency. And indeed, the cycle count data in Table 4.4 contains the smoking gun:

fms+mpp:
96: 3.6 x 10^8 cycles
1536: 4.7 x 10^8 cycles

omp:
96: 1.0 x 10^7 cycles
1536: 8.1 x 10^7 cycles

Thus the excess cycles due to the FMS application code (fms+mpp) is 1.1 x 10^8 while openMP increases by 7.1 x 10^7. This makes FMS about 51% of the total cycle expansion of 2.2 x 10^8 while openMP is about 32%.

Finally, the load imbalance as manifested in the time attributed to MPI is a very complex phenomenon. The transition to improved memory hierarchy utilization 4x4 → 8x8 seems to offset the negative scaling effects of increased workload. But at a minimum, there is special treatment of grid points on the corners and edges and there are likely other factors associated with increased

15 With certain restrictions on the nature of the instruction. For example, the AXU and XU must be from different h/w threads.

16 Potentially expensive because quick switching from a stalled thread to one ready to run can provide very effective latency hiding.

17 Of course the hardware counter data tell us that we're getting generally better cache behavior as we scale out.

overall point count. Without another performance boost, the cycles associated with load imbalance for 16x16 are headed towards 3X those for the 8x8 configuration (though still only about 17% of the total). Nevertheless, it accounts for only a minor portion of the scaling loss.

4.4 Machine Learning

This section outlines[18] the results of applying Machine Learning techniques (specifically linear and non-linear regression modeling) to the problem of performance projection. The HPMPROF hardware counter data acquired during the project was used as training input for the regression model.

The goal of the exercise was to predict the total cycles (*i.e.* the sum across all ranks) for a given layout based on the remainder of the hardware counter data. Both linear and cubic spline models were attempted. For those familiar with the complex changes in factors driving performance over a large scaling range, it is not surprising that the non-linear models were far better than linear models at performance prediction, especially at the largest rank counts.

The layout set analyzed was more extensive than that presented in this paper so far[19]. One of the reasons for this was to provide sufficient input training data. In fact, part of the work examined the trade-off between the size of the training set and the predictive power of the resulting model.

One of the interesting aspects that HPMPROF provided was the ability to associate counts with particular classes of activities. For example, the model differentiated between XU instructions associated with model computation verses those associated with thread sync operations.

The results of the approaches attempted here are encouraging. For layouts with smaller numbers of MPI ranks (4x4, 4x8, 8x8), the primary performance predictor was the number of FP operations (*i.e.* the work to be done). At higher scales, factors associated with thread synchronization operations (XU instructions and events producing dependency stall hardware counts) become an increasingly dominant predictor. Thus, major results of regression model analysis are consistent with the detailed findings presented in the previous sections

What's missing from this nascent work is what turns out to be the central result of the project:

The majority of the loss of scaling comes from the ever increasing total instruction count

But this is not really surprising. The time and resources available allowed for the development of performance models *within* a given MPI layout. This is the obvious first step. But it remains the case that any approach capable of arriving at the central result must at need probe relationships across the layout set. And in developing the techniques, somehow one must incorporate the notion that an increase in total count of some factors (*e.g.* XU instructions) is a negative for performance. Is this a set of constraints or some kind of damping? Whatever it is, one must formulate a way to incorporate it into the regression model.

While this is not the first attempt to apply these types of techniques to application performance[20], this may be the first attempt to apply Machine Learning to such a large, complex application.

18 For full details, see Appendix D.
19 4x4, 4x8, 4x8, 8x8, 8x16, 16x8 and 16x16
20 See for example reference LB06 in the full report

5 Summary

The work of this project has applied the IBM BG/Q MPI and thread aware hardware performance tools MPITRACE, HPM and HPMPROF to analyze the loss of scaling performance for the GFDL 3.5km resolution cubed-sphere atmosphere model. Initially it seemed quite reasonable to ascribe scaling loss to communication issues and perhaps load imbalance. But detailed analysis of the data seems to clearly demonstrate that the major factors are:

- The ever increasing total count of instructions associated with calculations in the subdomain halos
- Increasing relative overheads of the OpenMP regions that had little work in them in the first place

It is interesting to note that the issue with the halo calculations places fundamental limits on strong scaling for applications that perform such operations. On the other hand, the issue with the OpenMP overheads begs a couple of questions left unanswered by the current work:

- What portion of the overhead is thread count dependent?
- If there is such a dependency, might it improve performance to reduce the thread count for these low work regions as one scales out?
- If there is no thread count dependency in the overhead, might it be better to turn off the parallel region altogether?

In separate but related work, we have applied regression modeling with hardware counter data as input in an attempt to predict performance. Within the scope of project resources and time, some successful results were achieved with non-linear models within a particular model layout. But this nascent work misses the role of expanding instruction counts due to the subdomain halos. Left for future work is the development of techniques that can probe across layouts to arrive at the central results concerning increasing instruction count and the loss of scaling performance.

6 Acknowledgements

We would like to thank the Argonne Leadership Computing Facility and the Early Science Program for the funding and computation resources supporting this work.

We would also like to thank Bob Walkup of the IBM TJ Watson Research Center for numerous conversations. His depth of knowledge concerning the details of the IBM BG/Q hardware and software infrastructure as well as his willingness to provide ad-hoc modifications to the performance toolset to support the data gathering were truly assets to the project.

Appendix A

C_SW

	4x4	8x8	16x16			
	96	384	1536	384 / 96	1536 / 384	1536 / 96
Total Cycles	6.18E+13	5.92E+13	7.09E+13	0.96	1.20	1.15
Load Miss	2.70E+12	2.68E+12	2.93E+12	0.99	1.09	1.08
Cacheable Loads	2.66E+13	2.88E+13	3.39E+13	1.08	1.17	1.27
L1p Miss	5.53E+11	4.40E+11	5.80E+11	0.80	1.32	1.05
XU Inst	6.08E+13	6.74E+13	8.23E+13	1.11	1.22	1.35
AXU Inst	2.61E+13	2.74E+13	3.03E+13	1.05	1.10	1.16
XU / AXU	2.33	2.46	2.72	1.06	1.11	1.17
FP	3.54E+13	3.71E+13	4.07E+13	1.05	1.10	1.15
L2 Hit	6.32E+13	7.13E+13	8.47E+13	1.13	1.19	1.34
L2 Miss	3.70E+12	2.89E+12	1.72E+12	0.78	0.60	0.47
FPU %	30.05	28.93	26.89			
FXU %	69.95	71.07	73.11			
Bytes/cyc	12.608	8.968	5.625			
IPC	0.69	0.78	0.77			
Tot GF	85.70	373.95	1369.00			
% Loads that hit in						
L1D cache	89.84	90.70	91.36			
L1p Buffer	8.07	7.78	6.93			
L2 Cache	0.34	0.27	1.08			
DDR	1.74	1.25	0.64			

D_SW

	4x4	8x8	16x16			
	96	384	1536	384 / 96	1536 / 384	1536 / 96
Total Cycles	1.84E+14	1.89E+14	2.26E+14	1.03	1.19	1.23
Load Miss	8.62E+12	8.36E+12	9.52E+12	0.97	1.14	1.10
Cacheable Loads	1.07E+14	1.17E+14	1.38E+14	1.09	1.18	1.29
L1p Miss	1.43E+12	1.14E+12	1.91E+12	0.80	1.68	1.34
XU Inst	2.19E+14	2.46E+14	3.05E+14	1.12	1.24	1.39
AXU Inst	2.03E+14	2.18E+14	2.48E+14	1.07	1.14	1.22
XU / AXU	1.08	1.13	1.23	1.05	1.09	1.14
FP	2.42E+14	2.59E+14	2.95E+14	1.07	1.14	1.22
L2 Hit	2.63E+14	2.87E+14	3.48E+14	1.09	1.21	1.32
L2 Miss	1.20E+13	1.17E+13	8.69E+12	0.98	0.74	0.72
FPU %	48.09	46.95	44.83			
FXU %	51.91	53.05	55.17			
Bytes/cyc	13.857	12.024	7.674			
IPC	1.12	1.19	1.18			
Tot GF	196.68	813.51	3094.00			
% Loads that hit in						
L1D cache	91.94	92.84	93.12			
L1p Buffer	6.72	6.19	5.50			
L2 Cache	0.00	0.00	0.60			
DDR	1.34	0.98	0.79			

GEOPK

	4x4	8x8	16x16			
	96	384	1536	384 / 96	1536 / 384	1536 / 96
Total Cycles	3.07E+13	3.33E+13	4.87E+13	1.09	1.46	1.59
Load Miss	3.85E+11	4.25E+11	5.75E+11	1.10	1.35	1.49
Cacheable Loads	2.70E+13	2.91E+13	3.80E+13	1.07	1.31	1.41
L1p Miss	1.19E+11	2.01E+11	4.47E+11	1.69	2.23	3.75
XU Inst	5.16E+13	5.58E+13	7.85E+13	1.08	1.41	1.52
AXU Inst	2.05E+13	2.20E+13	2.53E+13	1.08	1.15	1.23
XU / AXU	2.52	2.53	3.11	1.01	1.23	1.23
FP	3.40E+13	3.65E+13	4.19E+13	1.07	1.15	1.23
L2 Hit	3.50E+13	4.14E+13	5.45E+13	1.18	1.31	1.56
L2 Miss	1.49E+12	1.26E+12	1.33E+12	0.84	1.06	0.89
FPU %	28.42	28.32	24.34			
FXU %	71.58	71.68	75.66			
Bytes/cyc	10.04	6.94	4.43			
IPC	1.16	1.15	1.01			
Tot GF	168.51	663.39	2001.00			
% Loads that hit in						
L1D cache	98.57	98.54	98.49			
L1p Buffer	0.99	0.77	0.34			
L2 Cache	0.00	0.15	0.74			
DDR	0.44	0.54	0.44			

Appendix B

First, a little about HPMPROF vs HPM. HPMPROF is a profiler that references the program counter at a specified sampling rate based on the overflow of a particular counter and so we are able to pick out individual subroutines (indeed the sample can be mapped to specific lines within a subroutine). HPM on the other hand simply reads the counter between a program start and stop point and thus measures everything in between.

In particular, the HPMPROF data selected out here excludes everything that is not an FMS source code routine while the HPM data includes all runtime library calls (math, openMP, etc). Further, HPM was configured to sum a particular counter value across all threads associated with an MPI-rank. In contrast, the version of HPMPROF used here records the value for Thread-0 only.

For HPM, separately named calipers were placed around the D_SW, C_SW and GEOPK call chains. The runtime configuration produced an average value (as well as min and max) across all ranks. The HPM average for a particular counter was used to arrive at a global total[21].

In contrast, the HPMPROF data provides a given counter broken down by subroutine. Global counts (for Thread-0) were calculated by summing across the per rank files.

The key observation for the analysis is that to the extent that the Thread-0 XU and AXU data exemplifies the true average across threads, the HPMPROF counts will be 1/8 the HPM counts. The different approaches yield strikingly congruent results for the D_SW and C_SW call chains while the comparison yields discordant results for GEOPK (which contains only the subroutine geopk). Luckily GEOPK comprises only about 7% of total cycles across the scaling range.

The following tables contain the HPMPROF data for a particular call chain on the left and the HPM data for that chain on the right.

C_SW
FP

	4x4	8x8	16x16
c_sw	2754274	2860094	3066215
d2a2c_vect	993311	1080560	1262994
divergence_corner	653389	677691	730424
Total*10^6	4.40E+12	4.62E+12	5.06E+12
			hpm/(8*hprof)

hpm over C_SW call chain

	4x4	8x8	16x16
	3.54E+13	*3.71E+13*	*4.07E+13*
	1.00	**1.00**	**1.00**

XU

	4x4	8x8	16x16
c_sw	5453249	5902115	6838298
d2a2c_vect	1185742	1398483	1852352
divergence_corner	855872	935429	1121178
Total*10^6	7.49E+12	8.24E+12	9.81E+12
			hpm/(8*hprof)

hpm over C_SW call chain

	4x4	8x8	16x16
	6.08E+13	*6.74E+13*	*8.23E+13*
	1.01	**1.02**	**1.05**

21 Simply GTot = NMPI * Avg

D_SW
FP

		4x4	8x8	16x16			
fyppm		9328545	10058426	11608209			
fxppm		9300661	10031083	11584917			
d_sw		3975705	4152380	4534689			
fv_tp_2d		3815718	4079314	4603914			
pert_ppm		2162231	2309012	2615015			
ytp_v		1371861	1446332	1587308	***hpm over D_SW call chain***		
xtp_u		1370466	1441689	1590149	4x4	8x8	16x16
	Total*10^6	3.13E+13	3.35E+13	3.81E+13	*2.42E+14*	*2.59E+14*	*2.95E+14*
				hpm/(8*hprof)	**0.96**	**0.97**	**0.97**

XU

		4x4	8x8	16x16			
fyppm		8776815	9639592	11738500			
fxppm		7030601	7905075	9829224			
d_sw		5443156	6129611	7602831			
fv_tp_2d		3069435	3371721	4099711			
pert_ppm		1192443	1310706	1604727			
ytp_v		1383861	1515908	1788126	***hpm over D_SW call chain***		
xtp_u		1055107	1167702	1361353	4x4	8x8	16x16
	Total*10^6	2.80E+13	3.10E+13	3.80E+13	*2.19E+14*	*2.46E+14*	*3.05E+14*
				hpm/(8*hprof)	**0.98**	**0.99**	**1.00**

GEOPK
FP

		4x4	8x8	16x16	***hpm over GEOPK call chain***		
geopk		2474017	2265588	2565790	4x4	8x8	16x16
	Total*10^6	2.47E+12	2.27E+12	2.57E+12	*1.93E+13*	*2.09E+13*	*2.46E+13*
				hpm/(8*hprof)	**0.97**	**1.16**	**1.20**

XU

		4x4	8x8	16x16	***hpm over GEOPK call chain***		
geopk		2241672	2310471	3069864	4x4	8x8	16x16
	Total*10^6	2.24E+12	2.31E+12	3.07E+12	*2.71E+13*	*2.95E+13*	*4.20E+13*
				hpm/(8*hprof)	**1.51**	**1.60**	**1.71**

Later in the project, we were supplied a version of the profiling library that attempts to sum across the thread team rather than rely on Thread-0. While there are some substantial caveats when attempting to perform interrupt driven profiling in a multi-threaded environment, HPMPROF_smp worked extremely well for the calipered experiments being performed. The results from this series of data focusing on cycle counts was consistent with the notion that the D_SW and C_SW call chains contain very low amounts of OpenMP overheads while GEOPK has a rather high percentage of cycles associated with OpenMP.

There results for the 16x16 and 8x8 MPI configurations are presented below. Since the counts can be picked out by subroutine, there was some ambiguity as to how account some of the routines that were not either FMS source or IBM math libraries (e.g. log, tanh, etc). In particular, OpenMP implementations encapsulate the parallel region in a subroutine call. For IBM XLF, these routines show up with the name $$OL$$[22]. The data below present the total counts sampled, those that OpenMP related except the $$OL$$ routine and then a similar calculation where counts associated with the $$OL$$ are tabulated as OpenMP overhead. These results are consistent with the HPMPROF vs HPM analysis.

HPMProf_SMP Cycles

16x16	Total	OMP	% OMP	OMP+OL	(OMP+OL)%
D_SW	590349	2596	0.44%	6746	1.14%
C_SW	178398	2483	1.39%	3130	1.75%
GEOPK	112648	7519	6.67%	47124	41.83%

8x8	Total	OMP	% OMP	OMP+OL	(OMP+OL)%
D_SW	968407	2230	0.23%	7704	0.80%
C_SW	297617	1460	0.49%	1731	0.58%
GEOPK	166730	5772	3.46%	64728	38.82%

22 The IBM XL compiler's OpenMP "outliner" which packages a code section into a thread function.

Appendix C

The following table contains the data provided by HPM calipers around each of the code regions D_SW, C_SW and GEOPK. HPM is run with the non-default settings HPM_THREADS=1 and HPM_GROUP=5. Setting the number of threads per core to 1 gives that thread access to all 24 of the core's event registers while HPM_GROUP 5 is setup to collect a detailed accounting of all XU instructions. The output provides the global average for each core and the values for the 16x16 and 8x8 decompositions are provided in the table below:

16x16	d_sw	c_sw	geopk	8x8	d_sw	c_sw	geopk
FP LD	2.11E+10	4.83E+09	5.12E+09	FP LD	7.38E+10	1.76E+10	1.61E+10
FP ST	6.84E+09	1.66E+09	1.39E+09	FP ST	2.39E+10	5.91E+09	4.39E+09
Quad LD	5.76E+06	0.00E+00	6.91E+05	Quad LD	1.45E+07	0.00E+00	2.59E+06
Quad ST	2.23E+07	0.00E+00	6.91E+05	Quad ST	8.03E+07	0.00E+00	2.59E+06
Bit Manip	6.83E+08	3.12E+08	8.08E+08	Bit Manip	1.17E+09	8.75E+08	2.32E+09
BR Cond	6.96E+09	1.67E+09	1.36E+09	BR Cond	2.25E+10	5.41E+09	3.43E+09
BR Uncond	4.87E+08	3.26E+08	2.43E+08	BR Uncond	1.59E+09	1.14E+09	7.31E+08
Cache Invalid	0.00E+00	0.00E+00	0.00E+00	Cache Invalid	0.00E+00	0.00E+00	0.00E+00
Cache ST	0.00E+00	0.00E+00	0.00E+00	Cache ST	0.00E+00	0.00E+00	0.00E+00
Cache Touch	3.28E+05	2.47E+05	4.69E+05	Cache Touch	9.14E+05	3.81E+05	5.42E+05
INT Arith	9.40E+09	2.89E+09	9.44E+08	INT Arith	2.96E+10	9.31E+09	2.34E+09
Cmp Inst	7.21E+08	3.76E+08	6.02E+08	Cmp Inst	1.40E+09	9.87E+08	1.38E+09
INT Div	5.76E+04	5.76E+04	1.15E+05	INT Div	5.76E+04	5.76E+04	1.15E+05
LOG Inst	6.58E+08	2.79E+08	4.35E+08	LOG Inst	1.21E+09	7.28E+08	1.20E+09
Reg mv	4.96E+08	9.63E+07	5.63E+07	Reg mv	9.24E+08	1.91E+08	8.21E+07
INT Mult	5.57E+08	1.84E+08	1.14E+08	INT Mult	9.96E+08	3.36E+08	1.98E+08
Interrupt	5.86E+05	3.09E+05	7.17E+05	Interrupt	1.77E+06	5.87E+05	8.19E+05
LD	1.62E+09	6.83E+08	1.23E+09	LD	2.98E+09	1.13E+09	2.86E+09
ST	5.16E+08	1.60E+08	4.35E+08	ST	9.24E+08	2.43E+08	1.18E+09
LD / ST & Res	6.44E+05	4.23E+05	9.67E+05	LD / ST & Res	1.27E+06	6.77E+05	1.13E+06
Context Sync	3.48E+06	2.01E+06	4.62E+06	Context Sync	8.86E+06	3.53E+06	5.44E+06
Sngl Thd/core Avg	5.01E+10	1.35E+10	1.28E+10	Sngl Thd/core Avg	1.61E+11	4.39E+10	3.62E+10
Global Total (proj)	3.08E+14	8.27E+13	7.83E+13	Global Total (proj)	2.47E+14	6.74E+13	5.56E+13

Table C.1

The Global Total is derived by multiplying the Single Thread / core average by the number of ranks. Further, in order to compare the Grand Total with the HPM results in Appendix A, one must also multiply by the 4 threads per core as is done in the table above.

To aid the analysis, we break the list into instruction types:

Cacheable LDs

16x16	d_sw	c_sw	geopk	8x8	d_sw	c_sw	geopk
FP LD	2.11E+10	4.83E+09	5.12E+09	FP LD	7.38E+10	1.76E+10	1.61E+10
Quad LD	5.76E+06	0.00E+00	6.91E+05	Quad LD	1.45E+07	0.00E+00	2.59E+06
LD	1.62E+09	6.83E+08	1.23E+09	LD	2.98E+09	1.13E+09	2.86E+09
Sngl Thd/core Avg	2.27E+10	5.51E+09	6.36E+09	Sngl Thd/core Avg	7.68E+10	1.87E+10	1.89E+10
Global Total (proj)	1.40E+14	3.39E+13	3.91E+13	Global Total (proj)	1.18E+14	2.88E+13	2.90E+13

Table C.2

Stores

16x16	d_sw	c_sw	geopk	8x8	d_sw	c_sw	geopk
FP ST	6.84E+09	1.66E+09	1.39E+09	FP ST	2.39E+10	5.91E+09	4.39E+09
Quad ST	2.23E+07	0.00E+00	6.91E+05	Quad ST	8.03E+07	0.00E+00	2.59E+06
ST	5.16E+08	1.60E+08	4.35E+08	ST	9.24E+08	2.43E+08	1.18E+09
Sngl Thd/core Avg	7.38E+09	1.82E+09	1.83E+09	Sngl Thd/core Avg	2.49E+10	6.15E+09	5.58E+09
Global Total (proj)	4.53E+13	1.12E+13	1.12E+13	Global Total (proj)	3.82E+13	9.45E+12	8.56E+12

Table C.3

Branch

16x16	d_sw	c_sw	geopk	8x8	d_sw	c_sw	geopk
BR Cond	6.96E+09	1.67E+09	1.36E+09	BR Cond	2.25E+10	5.41E+09	3.43E+09
BR Uncond	4.87E+08	3.26E+08	2.43E+08	BR Uncond	1.59E+09	1.14E+09	7.31E+08
Sngl Thd/core Avg	7.45E+09	1.99E+09	1.60E+09	Sngl Thd/core Avg	2.41E+10	6.55E+09	4.16E+09
Global Total (proj)	4.58E+13	1.22E+13	9.83E+12	Global Total (proj)	3.71E+13	1.01E+13	6.39E+12

Table C.4

Other

16x16	d_sw	c_sw	geopk	8x8	d_sw	c_sw	geopk
Bit Manip	6.83E+08	3.12E+08	8.08E+08	Bit Manip	1.17E+09	8.75E+08	2.32E+09
Cache Touch	3.28E+05	2.47E+05	4.69E+05	Cache Touch	9.14E+05	3.81E+05	5.42E+05
INT Arith	9.40E+09	2.89E+09	9.44E+08	INT Arith	2.96E+10	9.31E+09	2.34E+09
Cmp Inst	7.21E+08	3.76E+08	6.02E+08	Cmp Inst	1.40E+09	9.87E+08	1.38E+09
INT Div	5.76E+04	5.76E+04	1.15E+05	INT Div	5.76E+04	5.76E+04	1.15E+05
LOG Inst	6.58E+08	2.79E+08	4.35E+08	LOG Inst	1.21E+09	7.28E+08	1.20E+09
Reg mv	4.96E+08	9.63E+07	5.63E+07	Reg mv	9.24E+08	1.91E+08	8.21E+07
INT Mult	5.57E+08	1.84E+08	1.14E+08	INT Mult	9.96E+08	3.36E+08	1.98E+08
Interrupt	5.86E+05	3.09E+05	7.17E+05	Interrupt	1.77E+06	5.87E+05	8.19E+05
Context Sync	3.48E+06	2.01E+06	4.62E+06	Context Sync	8.86E+06	3.53E+06	5.44E+06
Sngl Thd/core Avg	1.25E+10	4.14E+09	2.97E+09	Sngl Thd/core Avg	3.53E+10	1.24E+10	7.53E+09
Global Total (proj)	7.69E+13	2.55E+13	1.82E+13	Global Total (proj)	5.42E+13	1.91E+13	1.16E+13

Table C.5

Note that both the XU Grand Total in Table C.1 and Cacheable LDs in Table C.2 are in very good agreement with the values in Appendix A.

Appendix D

Regression Modeling for Application Performance Prediction

April 23, 2014

Abstract

We perform regression modeling to obtain estimates of the *number of cycles* (used as a proxy for application performance). A linear regression is performed in which the response (number of cycles) is a weighted sum of predictor variables[1] (such as L1P misses, AXU instructions, XU instructions, FP operations, etc.) plus some random noise. Since *linear* relationships may not always be adequate to model the relationship between the predictors and response, non-linear modeling using cubic splines [Wah90] have also been performed. Both the linear and non-linear models are evaluated on blind test data using the R-squared and Mean Square Error (MSE) metric. Our results indicate that non-linear models are much better (higher R-squared) in capturing the characteristics of application performance modeling.

1 Regression Theory

1.1 Linear Models

The linear regression model is used extensively in the statistics and machine learning community. We describe the model here: Let $X_i = (X_{i1}, X_{i2}, \cdots, X_{iP}), 1 \leq i \leq N$ denote real valued random input vectors and $Y_i \in \mathcal{R}, 1 \leq i \leq N$ the real valued output corresponding to X_i. The goal is to predict the output vector Y using the model:

$$\hat{Y}_i = \beta_0 + \sum_{j=1}^{P} X_j \beta_j \tag{1}$$

where $\beta^T = (\beta_0, \beta_1, \cdots, \beta_P)$ denotes the set of regression coefficients. The term β_0 is also known as the intercept or *bias* in machine learning literature. Equation 1 is often written as an inner product by including the bias term β_0 in the coefficient vector and adding the constant variable 1 in X [HTF] as follows:

$$\hat{Y} = X\beta \tag{2}$$

In the $(P+1)$-dimensional input-ouput space, $(X_i, \hat{Y}_i)$ represents a hyperplane. If the constant is included in X_i then the hyperplane includes the origin and is a subspace; if not, it is an affine set cutting the Y-axis at β_0. The most popular method of fitting a linear model to the data available for training is to use the method of least squares. The coefficients β are picked by minimizing the Residual Sum of Squares (RSS):

$$RSS(\beta) = \sum_{i=1}^{N} (Y_i - \beta_0 - \sum_{j=1}^{P} X_{ij}\beta_j)^2 \tag{3}$$

$RSS(\beta)$ can be minimized by solving a system of $P+1$-partial derivatives of $RSS(\beta)$ with respect to $\beta_j, j \in [0, p]$. The solutions of this system are the estimates of coefficients in Equation 1.

[1] Also interchangeably called features or attributes.

1.2 Non-Linear Models

The assumption that the response behaves linearly with the predictors is somewhat restrictive. Relaxing this assumption, predictors suspected of having a non-linear correlation with the response are allowed to undergo polynomial transformations. However, polynomials have peaks and valleys and good fits in one region may adversely affect the fit in another region [LB06].

Spline functions are considered a better alternative in modeling non-linearity. These are piecewise polynomials used in curve fitting. A function is broken down into intervals defining multiple different continuous polynomials with endpoints called *knots*. The number of knots can vary but more knots typically correlate to better fits. An order-M spline with knots $\xi_j, j = 1, \cdots, K$ is a piecewise polynomial of order M and has continuous derivatives upto order $M-2$. A cubic spline has $M = 4$. In practice, the most widely used orders are $M = 1, 2, \text{and}, 4$ [HTF].

1.3 Evaluation

1.3.1 Assessing Fit

The coefficient of determination, denoted by R^2 is typically used to indicate how well the data fits the model. In case of linear regression, if an intercept is included, then R^2 is simply the square of the sample correlation coefficient between the outcomes and their predicted values. Let $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ represent the mean of the observed data. The "variability" of the data is measured by different metrics: (a) Total sum of squares $SST = \sum_i (y_i - \bar{y})^2$. (b) Residual sum of squares $SSE = \sum_i (y_i - f_i)^2$. R^2 is defined as follows:

$$R^2 = 1 - \frac{SSE}{SST} \tag{4}$$

An R^2 of 1 indicates that the regression line perfectly fits the data; however, a value too close to $R^2 = 1$ may also indicate over-fitting.

1.3.2 Generalization Performance

The generalization performance of a learning model relates to its predictive ability on an independent test set. Assessment of this performance is of consequence since it is likely to guide the choice of the model, parameter space and measure of the quality of the model. We distinguish between the notions of training error and test error: (a) *Training error:* is the average loss over the training sample. Training Error $= \frac{1}{N}\sum_{i=1}^{N} L(Y_i, \hat{Y}_i)$ where L is the loss function. (b) *Test error* also called *Generalization error* is the prediction error over an independent test sample. Test Error $= E[L(Y_i, \hat{Y}_i)|\mathcal{T}]$, where $\mathcal{T}$ is a fixed training set. Thus test error is always determined with reference to a specific training set, such as $\mathcal{T}$.

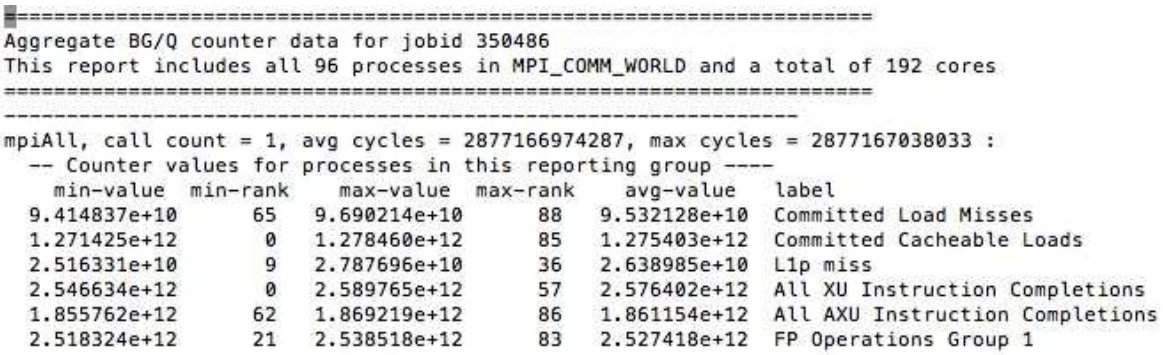
2 Data collected for performance prediction

Performance data was collected using two tools – Hardware Performance Monitor (HPM) and HPMPROF – that were installed on the Blue Gene system. The tools are part of the standard IBM toolset. We used custom shell and Python scripts to parse and massage the output to make it in a form suitable for input into PerfBrowser, a proprietary software for visualizing performance data. PerfBrowser makes it easy to compare and visualize data across multiple experiments.

2.0.3 HPM and HPMPROF

HPM is an IBM tool for gathering hardware performance metrics, such as cache misses, on IBM architectures. It is part of the IBM High Performance computing toolkit. The tool has negligible runtime overhead. It requires the program to be linked with HPM libraries. HPM also supports start/stop sentinels to measure performance metrics in specific code regions. We used HPM to gather metrics both for the program, and the main compute region.

HPM can run in two scopes: node and process. In the node mode, data is collected at a per-node basis, while in the process mode, data is collected for each process (MPI rank). The environment variable



```
=================================================================
Aggregate BG/Q counter data for jobid 350486
This report includes all 96 processes in MPI_COMM_WORLD and a total of 192 cores
=================================================================
-----------------------------------------------------------------
mpiAll, call count = 1, avg cycles = 2877166974287, max cycles = 2877167038033 :
  -- Counter values for processes in this reporting group ----
    min-value  min-rank    max-value  max-rank    avg-value   label
  9.414837e+10       65   9.690214e+10       88   9.532128e+10  Committed Load Misses
  1.271425e+12        0   1.278460e+12       85   1.275403e+12  Committed Cacheable Loads
  2.516331e+10        9   2.787696e+10       36   2.638985e+10  L1p miss
  2.546634e+12        0   2.589765e+12       57   2.576402e+12  All XU Instruction Completions
  1.855762e+12       62   1.869219e+12       86   1.861154e+12  All AXU Instruction Completions
  2.518324e+12       21   2.538518e+12       83   2.527418e+12  FP Operations Group 1
```

Figure 1: Sample HPM output for the 4x4 layout

HPM_SCOPE is used to select the mode. We used the process mode. While collecting data in a process mode, care has to be taken while interpreting metrics that used shared counters, such as L2 misses. A2 counters, are easy to interpret as the specific to each core. HPM can select a combination of performance counters based on HPM_GROUP. For our experiments, we primarily used HPM_GROUP zero.

Another HPM environment setting that is worth mentioning is $SAVE\_ALL\_TASKS$. By default, HPM presents a final summary by aggregating data across tasks. The summary is concise, but loses the data variation across ranks. We used HPM, both with and without the variable to collect data for individual tasks and job statistics (See Figure 1).

HPMPROF allows statistical profiling of a performance event, such L1-D misses. A performance threshold and a performance event is selected for a run. A counter is incremented on each occurrence of the event. Once the threshold is reached, an overflow is said to occur, and the counter is reset, and a counter associated with a program counter value is incremented. Assuming enough overflows, profile generated attributes the performance event to specific code sections. We used hpmprof to profile total cycles, L1-misses, instructions executed, FP operations and dependency stalls. We also used the $SAVE\_ALL\_TASKS$ environment variable to save per-rank data to detect variability across ranks. Similar to HPM, HPMPROF requires the program to be linked with HPMPROF libraries (See Figure 2).

PerfBrowser is a cloud HPC performance data visualization software. It's hosted and maintained by Samara Technology Group LLC. Once performance data is collected using a supported performance tool, it's uploaded to PerfBrowser. PerfBrowser allows comparison of performance metrics across varying process counts and experiments. It produces both textual diffs and plots, making it easy to spot reasons for reduced scaling and performance. As part of this project, we extended PerfBrowser to import HPM and HPMPROF output formats (See Figures 3, 4, 5). In the data import plugin, we added a feature to allow aggregating performance metrics in thread synchronization and MPI calls. This made it easier to abstract out performance data that was not attributable to useful work done by the program.

We collected performance data for a variety of layouts after reducing the problem to an equivalent smaller problem size. For each layout, we used 8 threads per rank, and 8 ranks per node. We had also tried other thread and rank counts per node, but determined the 8thread/rank and 8ranks/node test case as one that yielded the fastest time to solution [Why?]. To make the data collection systematic, we wrote scripts to iterate over the layout parameters. We used the following layouts: 4x4, 4x8, 8x4, 8x8, 8x16, 16x8 and 16x16. [Insert script source for data collection?]

Table 1 describes the features generated for each layout which are then used for learning regression models. The total number of samples generated for each configuration is listed in Table 2. This is further split into train-test pairs by randomly sampling 70% percent for training [The Appendix provides a justification of how the percentages for splitting the data were obtained].

```
Using executable file : ./fms_base.x.hpmprof, histogram file : ./hpm_histogram.367422.0.
Got a total of 450117 hits at 17831 program-counter locations.
HPM sampling using event = 239, threshold = 1000000.

#########################
Function-level profile:
#########################
    tics   function-name
-------------------------
   55768  fyppm
   45794  fxppm
   34546  _log
   19619  d_sw
   18837  c_sw
   18751  fv_tp_2d
   16106  _exp
   10513  _int_free
   10229  a2b_ord4
    9946  _int_malloc
    9860  sim_solver
    9789  PAMI_Context_trylock_advancev
    9036  MPIR_Grequest_progress_poke
    8593  __libc_malloc
    8241  map1_ppm
    8218  simplest_solver
    7839  __libc_free
    7244  MPIR_Wait_impl
    5582  __dyn_core_mod_NMOD_split_p_grad$$OL$$23
    5454  cs_profile
    5243  ytp_v
    4738  pert_ppm
    4400  d2a2c_vect
```

Figure 2: Sample HPMPROF output for INST_XU_ALL.

3 The Regression Models

Linear Models: Table 4 presents the six linear regression models obtained for each layout. Interestingly, the R^2 values for layouts $4 \times 4, 4 \times 8, 8 \times 8$ are substantially higher (≥ 0.73) than the other layouts. This may be attributed to the fact that past 8×8, the loss of scaling is driven by the operations performed in the halo region. These operations are the same types of operations performed in the computational region and so the factors driving their performance are similar. But the halo computations are in fact redundant and do not in and of themselves move the solution forward. They are, however, preferable to communicating the values from the native MPI rank since communication has its own expenses. The summary of performance prediction error from the models on training data[2] is shown in Table 6.

The residuals versus fitted values and Quantile-Quantile (Q-Q) plots of standardized residuals for two layouts 8X8 and 16X16 are shown in Figure 6. The quantile plot is used to determine if the residuals are close to being normally distributed; the theoretical line that the data should fall on if they were normally distributed is also plotted for comparison.

Cubic Spline Models: A different class of models called *Generalized Additive Models* [HT86] replaces the linear function in Equation 1 by an additive function $\sum_{j=1}^{P} s_j(X_j)$. A local scoring algorithm is used for estimating the $s_j()$'s and a spline smoother[3] is used for our problem. The mean of each variable is used to define the knots. Table 6 summarizes the models and their performance. Most notably, the R^2 values are much higher (≥ 0.9) for all the six layouts. For the 16X16 layout, the highest coefficient is observed for the L1D.Misses which play a significant role in performance prediction. Figure 7 shows the residuals vs fitted

[2]In all of the experiments here, the training data is approximately 70% of the instances randomly chosen. The R^2 and MSE values are estimated on the remaining 30% of the data which acts as 'blind' test data. This choice is justified by the fact that not much variability in performance is noted using more (90%) or less (50%) data for training as reported in the Appendix.

[3]Note that it is also possible to use other smoothers such as local average estimates or kernels for scoring.

Feature	Description
Committed.Load.Misses	L1 D-cache load misses
Committed.Cacheable.Loads	Total loads
L1p.miss	Loads that missed the L1P buffer
All.XU.Instruction.Completions	All execution unit instructions (arithmetic, FP)
All.AXU.Instruction.Completions	
FP.Operations.Group.1	Specific class of FP instructions (on BG/Q)
Dep.Stalls	Stalls cycles due to some dependency
DS.THREAD.SYNC	Dep. stalls in a thread sync function
DS.COMM	Dep. stalls in an communication function
DS.THREAD.SYNC.COMM	Dep. stalls in a thread sync/comm. function
FP.Grp1	FP operations Group 1
FP.THREAD.SYNC	Floating Point operations in a thread sync function
FP.COMM	FP operations in a communication function
FP.THREAD.SYNC.COMM	FP in a thread sync/comm function
INST.XU.ALL	All XU Instruction completions
XU.THREAD.SYNC	XU instructions in a thread sync function
XU.COMM	XU instructions in a communication function
XU.THREAD.SYNC.COMM	XU instruction in a thread sync/comm function
L1D.Misses	L1 D-Cache misses
L1D.THREAD.SYNC	L1 D-cache misses in a thread sync function
L1D.COMM	L1 D-cache miss in a communication function
L1D.THREAD.SYNC.COMM	L1 D-cache miss in a thread sync/comm. function
L1P.Misses	L1P misses
L1P.THREAD.SYNC	L1P misses in a thread sync function
L1P.COMM	L1P misses in a thread sync function
L1P.THREAD.SYNC.COMM	L1P misses in a thread sync or comm. function
L2.trunc	L2 misses per node

Table 1: Features used for building Regression Models.

values and Quantile-Quantile plots for cubic spline models from two layouts (8X8 and 16X16).
Discussion: In the modeling tasks above, regression analysis has been used to produce an equation that will predict a dependent variable (number of cycles) using one or more independent variables (such as,L1 Misses, FP Operations, etc.). For example, the equation for predicting the performance of the 4×4 layout using a linear model is as follows:

$$
\begin{aligned}
\text{Number of cycles} &= 4.71\text{e+}12 - 2.17\text{e-}03(Committed.Load.Misses) - 2.63\text{e-}04(Committed.Cacheable.Loads) \\
&- 1.197\text{e-}03(L1p.miss) + 1.22\text{e-}04(All.XU.Instruction.Completions) \\
&+ 7.18\text{e-}05(All.AXU.Instruction.Completions) - 6.22\text{e-}06(FP.Operations.Group1) \\
&- 1.90\text{e+}01(Dep.Stalls) + 1.14\text{e+}02(DS.Thread.Sync) + 1.38\text{e+}02(DS.Comm) \\
&- 6.36\text{e+}02(FP.Grp1) + 5.87\text{e+}05(FP.Thread.Sync) + 2.14\text{e+}02(FP.Comm) \\
&- 6.96\text{e+}02(Inst.XU.All) + 3.06\text{e+}02(XU.Thread.Sync) + 5.48\text{e+}02(XU.Comm) \\
&+ 1.85\text{e+}04(L1D.Misses) + 1.80\text{e+}04(L1D.Thread.Sync) - 1.56\text{e+}04(L1D.Comm) \\
&+ 1.04\text{e+}04(L1P.Misses) + 2.49\text{e+}04(L1P.Thread.Sync) - 4.83\text{e+}03(L1P.Comm) \\
&+ 2.11\text{e-}04(L2Misses)
\end{aligned}
$$

The sign (positive or negative) of a coefficient plays a crucial role in interpretation of the models. For example, in the above equation, the number of cycles is predicted to increase by 7.18e-05 when All.AXU.Instruction.Completions variable goes up by 1 (holding all other variables constant), decrease 6.96e+02 when Inst.XU.All variable goes

Configuration	No. of Samples
4X4	96
4X8	192
8X8	384
8X16	768
16X8	768
16X16	1536

Table 2: No. of samples generated for each configuration.

Layout	Min	1st Quartile	Median	3rd Quartile	Max
4X4	-1978884	-440959	0	554486	1788585
4X8	-5180186	-438508	-7788	441921	2373970
8X8	-5306584	-359207	16723	392478	1681010
8X16	-10971711	-268247	19408	290437	2306172
16X8	-0.008021	-0.001059	-0.000138	0.00079	0.125156
16X16	-0.091776	-0.999322	0.000006	0.000436	0.003262

Table 3: Summary of performance prediction error in different layouts.

up by 1 (holding all other variables constant), and is 4.71e+12 when all the independent variables take a value 0. In addition, the coefficient with the highest positive value (other than the intercept), FP.Thread.Sync in the above model may be interpreted to be the most significant variable when constructing the model. Sorting variables by the coefficients therefore gives some notion of their relative importance in the model. Based on the above criteria, the top 5 drivers of performance in the different layouts are shown i n Table 3.

In regression problems, $f(X) = E(Y|X)$ will typically be non-linear and non-additive in X, but representing $f(X)$ as a linear problem is often a necessary approximation. To go beyond linearity, the vector inputs X is typically augmented or replaced with additional variables which are transformations of X and then linear models are used in this new space of derived input features.

Let $h_m(X) : R^P \rightarrow R$ be the m^{th} transformation of X – some examples of $h_m(X)$ include $h_m(X) = X_m, m = 1, 2 \cdots, p$ (which recovers the original linear model) and $h_m(X) = X_j^2$ or $h_m(X) = X_i X_j$ which allows inclusion of polynomial terms. The regression model that can be learnt is then $f(X) = \sum_{m=1}^{M} \beta_m h_m(X)$ which is often called a *linear basis expansion* of X.

Piecewise polynomials or *splines* can be used as transformations of X. A piecewise polynomial function is obtained by dividing the domain of X into contiguous intervals, and representing each interval by a separate polynomial. In our experiments, we have adapted this technique and each variable is represented by a *cubic* spline. The domain of each variable is split into two parts (min(X), mean(X)) and (mean(X), max(X)) and each of these parts are represented by a cubic polynomial also called a *natural cubic spline*. The mean acts as a *knot*[4]. Thus, for all the models reported in Table 6, the number of variables used for prediction is twice the number of variables used in the linear regression models. In this new space of variables, a linear model is learnt. Looking at the coefficients of this linear model, still conveys important information about the drivers of performance. For instance, the following are the top drivers of performance in the six layouts in the cubic spline models:

- 4X4: FP.Grp1
- 4X8: FP.Thread.Sync
- 8X8: FP.Thread.Sync
- 8X16: FP.Thread.Sync.Comm, DS.Thread.Sync and XU.Thread.Sync

[4]The choice of where a knot should be placed and how many knots should be included is a design consideration.

	Model 4X4	Model 4X8	Model 8X8	Model 8X16	Model 16X8	Model 16X16
(Intercept)	4.712e+12***	2.341e+12***	1.191e+12***	6.898e+11***	6.962e+11***	4.304e+11***
Committed.Load.Misses	-2.179e-03*	1.189e-03*	-5.569e-04	-1.498e-03***	2.468e-12	1.469e-13
Committed.Cacheable.Loads	-2.636e-04*	-4.892e-04***	-7.079e-04***	-4.901e-05	1.280e-12*	-5.184e-13
L1p.miss	-1.197e-03	-2.533e-03**	-2.701e-03***	-6.810e-05	-4.285e-12	2.635e-12
All.XU.Instruction.Completions	1.222e-04***	9.978e-05***	2.392e-04***	3.387e-06	-3.902e-13	1.822e-13
All.AXU.Instruction.Completions	7.180e-05	1.520e-04*	1.147e-04	2.246e-04**	-1.397e-12*	1.013e-13
FP.Operations.Group.1	-6.228e-06	-1.709e-04	-5.233e-05	-1.991e-04**	1.096e-12*	-3.436e-13
Dep.Stalls	-1.900e+01	1.416e+02**	-4.793e+01	-1.623e+02**	2.007e-07	4.121e-08
DS.Thread.Sync	1.142e+02	-1.003e+02	-2.797e+02***	-1.500e+02	5.750e-07	-8.095e-07**
DS.Comm	1.381e+02	3.258e+01	-5.848e+02	2.392e+02*	2.411e-07	1.109e-07
FP.Grp1	-6.360e+02**	8.650e+01	-2.448e+02	-1.311e+01	-3.578e-07	1.480e-06*
FP.Thread.Sync	5.879e+05	4.277e+05*	-8.476e+04	-6.057e+04	-	-
FP.Comm	2.145e+02	-2.476e+04	-1.828e+04	-1.022e+04	3.125e-04***	-4.930e-05
Inst.XU.All	-6.964e+02***	-8.812e+02***	-1.601e+03***	-1.587e+03***	6.137e-07	-7.628e-07
XU.Thread.Sync	3.062e+02	6.631e+02**	1.377e+03***	1.759e+03***	3.652e-07	5.955e-07
XU.Comm	5.482e+02***	9.925e+02***	2.114e+03***	1.687e+03***	-9.636e-07	8.929e-07
L1D.Misses	1.856e+04***	7.652e+02	1.341e+04***	3.309e+04***	-2.032e-05	-2.315e-06
L1D.Thread.Sync	1.804e+04	7.462e+03	-6.810e+03	-5.532e+03	-2.626e-05	-4.516e-06
L1D.Comm	-1.564e+04	3.322e+02	-6.314e+03	-8.872e+03***	-1.159e-06	-2.306e-05*
L1P.Misses	1.049e+04*	9.987e+03**	1.943e+04***	4.312e+03	-4.351e-05	-1.026e-05
L1P.Thread.Sync	2.495e+04	-3.736e+03	1.352e+04**	-3.172e+03	4.612e-06	4.778e-06
L1P.Comm	-4.826e+03	-1.293e+03	4.204e+02	-1.462e+03	3.400e-06	1.400e-05
L2.trunc	2.106e-04	-1.459e-03***	-2.438e-04	1.137e-04	-7.810e-14	-5.104e-13
R^2	0.89	0.74	0.73	0.54	0.49	0.50
Adj. R^2	0.85	0.69	0.71	0.52	0.48	0.49
MSE	1.09e+12	5.22e+11	3.39e+11	3.45e+11	2.72e+12	1.36e+12

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Table 4: Linear Regression Models for layouts $4 \times 4, 4 \times 8, 8 \times 8, 8 \times 16, 16 \times 8, 16 \times 16$.

- 16X8: DS.Thread.Sync and DS.Comm
- 16X16: DS.Thread.Sync and XU.Thread.Sync

It therefore appears that in smaller layouts, 4X4, 4X8, and 8X8 performance prediction is dominated by FP operations while for larger layouts 8X16, 16X8 and 16X16 DS.Thread.Sync and XU.Thread.Sync appears to play a major role.

References

[HT86] T. Hastie and R. Tibshirani. Generalized additive models. *Statistical Science*, 1(3):297–318, October 1986.

[HTF] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*.

[LB06] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ASPLOS*, October 2006.

[Wah90] Grace Wahba. *Spline models for observational data*, volume 59 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1990.

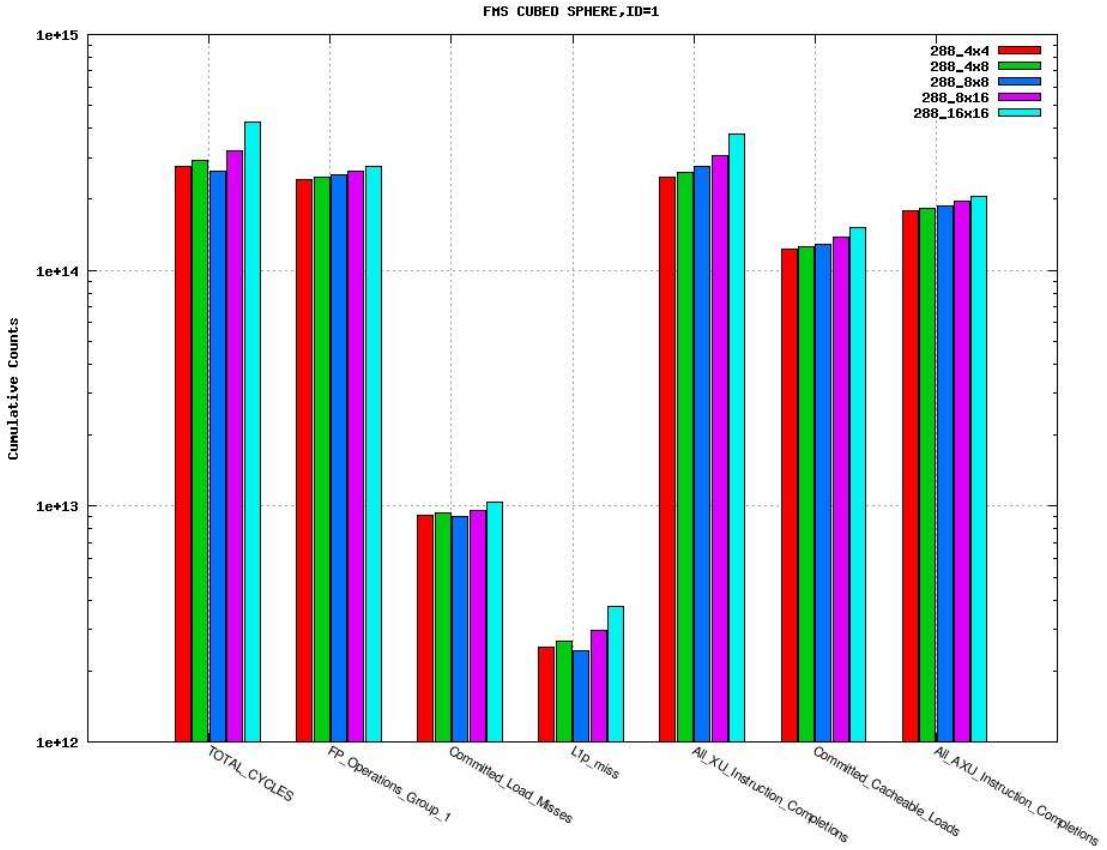
4X4	4X8	8X8	8X16	16X8	16X16
FP.Thread.Sync	FP.Thread.Sync	L1P.Misses	L1D.Misses	FP.Comm	L1P.Comm
L1P.Thread.Sync	L1P.Misses	L1P.Thread.Sync	L1P. Misses	L1P.Thread.Sync	L1P.Thread.Sync
L1D.Misses	L1D.Thread.Sync	L1D.Misses	XU.Thread.Sync	L1P.Comm	Fp.Thread.Sync
L1D.Thread.Sync	XU.Comm	XU.Comm	XU.Comm	Inst.XU.All	XU.Comm
L1P.Misses	L1D.Misses	XU.Thread.Sync	FP.Grp1	DS.Thread.Sync	XU.Thread.Sync

Table 5: Top 5 drivers of performance in linear models for different layouts.

Metric	288_4x4	288_4x8	288_8x8	288_8x16	288_16x16	Variation
TOTAL_CYCLES	2.762080e+14	2.923663e+14	2.622261e+14	3.221857e+14	4.254120e+14	54%
FP_Operations_Group_1	2.425824e+14	2.476320e+14	2.535324e+14	2.642905e+14	2.761441e+14	13%
Committed_Load_Misses	9.148772e+12	9.409766e+12	9.023647e+12	9.629414e+12	1.037172e+13	13%
L1p_miss	2.530735e+12	2.686103e+12	2.448708e+12	2.968453e+12	3.738543e+12	47%
All_XU_Instruction_Completions	2.468684e+14	2.601189e+14	2.747967e+14	3.079855e+14	3.800983e+14	53%
Committed_Cacheable_Loads	1.223120e+14	1.266037e+14	1.290622e+14	1.381257e+14	1.525533e+14	24%
All_AXU_Instruction_Completions	1.786303e+14	1.831141e+14	1.878025e+14	1.970650e+14	2.069407e+14	15%

The table shows the rank-aggregate count for each metric.
The Variation column shows the % increase in the rank-aggregate count
for a metric between the least and the most NPE runs.

(a) HPM DIFF



(b) HPM PLOT

Figure 3: Perfbrowser with HPM

function	96.4x4_vprof_96_C	192.4x8_vprof_192_	384.8x8_vprof_384_	1536.16x16_CYCLES_2	Variation	%TotalDelta
TOTAL	398315361 [100.0%]	391261369 [100.0%]	387654035 [100.0%]	538600851 [100.0%]	35%	100.0%
TOTAL_EX_SYNC_MPI	371194869 [93.2%]	356426141 [91.1%]	362774693 [93.6%]	448542379 [83.3%]	20%	55.1%
fyppm	50040254 [12.6%]	50643188 [12.9%]	53224369 [13.7%]	63067148 [11.7%]	26%	9.3%
fxppm	51117956 [12.8%]	50418078 [12.9%]	49901747 [12.9%]	58463746 [10.9%]	14%	5.2%
c_sw	42861078 [10.8%]	41288799 [10.6%]	42086222 [10.9%]	48629971 [9.0%]	13%	4.1%
d_sw	34314638 [8.6%]	32464357 [8.3%]	34794147 [9.0%]	41060848 [7.6%]	19%	4.8%
THREAD_SYNC	7585504 [1.9%]	10655296 [2.7%]	17074307 [4.4%]	66855832 [12.4%]	781%	42.2%
fv_tp_2d	20296151 [5.1%]	18834279 [4.8%]	19388184 [5.0%]	23638477 [4.4%]	16%	2.4%
_log	16908533 [4.2%]	16989125 [4.3%]	17877853 [4.6%]	23032580 [4.3%]	36%	4.4%
COMM	19534988 [4.9%]	24179932 [6.2%]	7805035 [2.0%]	23202640 [4.3%]	18%	2.6%
__dyn_core_mod_NMOD_geop	16586242 [4.2%]	15211133 [3.9%]	17076317 [4.4%]	22899741 [4.3%]	38%	4.5%
_exp	11633534 [2.9%]	11640939 [3.0%]	12295732 [3.2%]	15839350 [2.9%]	36%	3.0%
pert_ppm	10608409 [2.7%]	10376060 [2.7%]	10378855 [2.7%]	11799883 [2.2%]	11%	0.8%
map1_ppm	9901591 [2.5%]	10189996 [2.6%]	9504553 [2.5%]	10271035 [1.9%]	3%	0.3%
a2b_ord4	9784926 [2.5%]	8488610 [2.2%]	9262420 [2.4%]	11789303 [2.2%]	20%	1.4%
d2a2c_vect	8802566 [2.2%]	8421665 [2.2%]	8408947 [2.2%]	11022418 [2.0%]	25%	1.6%
ytp_v	8282650 [2.1%]	8027941 [2.1%]	8441306 [2.2%]	9515899 [1.8%]	14%	0.9%
__dyn_core_mod_NMOD_one_	8617982 [2.2%]	8380892 [2.1%]	8293706 [2.1%]	8485906 [1.6%]	-2%	-0.1%
xtp_u	7942983 [2.0%]	7858679 [2.0%]	7658236 [2.0%]	8351125 [1.6%]	5%	0.3%
__dyn_core_mod_NMOD_p_gr	7815876 [2.0%]	7929114 [2.0%]	7766018 [2.0%]	7353206 [1.4%]	-6%	-0.3%
cs_profile	5691399 [1.4%]	5834610 [1.5%]	5748846 [1.5%]	7459620 [1.4%]	31%	1.3%
divergence_corner	6426282 [1.6%]	5772331 [1.5%]	5866266 [1.5%]	6393158 [1.2%]	-1%	-0.0%
__libc_malloc	9229719 [2.3%]	7839573 [2.0%]	2187264 [0.6%]	648193 [0.1%]	-93%	-6.1%
__libc_free	8547132 [2.1%]	7233658 [1.8%]	1925210 [0.5%]	98300 [0.0%]	-99%	-6.0%
map1_q2	3665103 [0.9%]	3659999 [0.9%]	3387499 [0.9%]	3796593 [0.7%]	3%	0.1%
__mpp_domains_mod_NMOD_m	2323041 [0.6%]	1756657 [0.4%]	3732764 [1.0%]	4516934 [0.8%]	94%	1.6%
__fv_mapz_mod_NMOD_lagra	2847005 [0.7%]	2742946 [0.7%]	2855092 [0.7%]	3336197 [0.6%]	17%	0.3%
__mpp_domains_mod_NMOD_m	2178642 [0.5%]	1339291 [0.3%]	2599372 [0.7%]	5177098 [1.0%]	137%	2.1%
prt_maxmin	2703546 [0.7%]	2707513 [0.7%]	2803410 [0.7%]	2870867 [0.5%]	6%	0.1%
memset	0 [0.0%]	0 [0.0%]	1741442 [0.4%]	6166560 [1.1%]	inf%	4.4%
mpp_complete_update_doma	0 [0.0%]	0 [0.0%]	1570647 [0.4%]	5873639 [1.1%]	inf%	4.2%
_int_free	3691127 [0.9%]	3303889 [0.8%]	114963 [0.0%]	2186 [0.0%]	-100%	-2.6%
mpp_start_update_domain2	0 [0.0%]	0 [0.0%]	793828 [0.2%]	4888267 [0.9%]	inf%	3.5%
__mpp_domains_mod_NMOD_m	0 [0.0%]	0 [0.0%]	1573004 [0.4%]	3798555 [0.7%]	inf%	2.7%
mpp_complete_update_doma	0 [0.0%]	0 [0.0%]	423141 [0.1%]	4939694 [0.9%]	inf%	3.5%
_int_malloc	2497973 [0.6%]	2405153 [0.6%]	114847 [0.0%]	2278 [0.0%]	-100%	-1.8%
__mpp_domains_mod_NMOD_m	0 [0.0%]	0 [0.0%]	1282343 [0.3%]	3372021 [0.6%]	inf%	2.4%
mpp_do_update_r8_3dv	9406 [0.0%]	50464 [0.0%]	1660456 [0.4%]	2932368 [0.5%]	31075%	2.1%
mpp_start_update_domain2	0 [0.0%]	0 [0.0%]	95818 [0.0%]	4500505 [0.8%]	inf%	3.2%
__dyn_core_mod_NMOD_adv_	1612204 [0.4%]	1514589 [0.4%]	1390562 [0.4%]	0 [0.0%]	-100%	-1.1%
__dyn_core_mod_NMOD_dyn_	1323300 [0.3%]	1321135 [0.3%]	1583111 [0.4%]	174863 [0.0%]	-87%	-0.8%
range_check	1354819 [0.3%]	1331570 [0.3%]	1410661 [0.4%]	0 [0.0%]	-100%	-1.0%
c2l_ord4	1209737 [0.3%]	424393 [0.1%]	1396393 [0.4%]	39665 [0.0%]	-97%	-0.8%
AssignWI	0 [0.0%]	0 [0.0%]	0 [0.0%]	2241669 [0.4%]	inf%	1.6%
a2b_ord2	301795 [0.1%]	0 [0.0%]	58051 [0.0%]	0 [0.0%]	-100%	-0.2%
_qsin	0 [0.0%]	0 [0.0%]	76329 [0.0%]	73057 [0.0%]	inf%	0.1%
__write_nocancel	10533 [0.0%]	9484 [0.0%]	9261 [0.0%]	13887 [0.0%]	31%	0.0%
_xldfrct	14680 [0.0%]	8326 [0.0%]	5175 [0.0%]	1880 [0.0%]	-88%	-0.0%
fv_dynamics	12336 [0.0%]	7705 [0.0%]	4044 [0.0%]	0 [0.0%]	-100%	-0.0%
__dyn_core_mod_NMOD_init	19092 [0.0%]	0 [0.0%]	0 [0.0%]	0 [0.0%]	-100%	-0.0%
_xldexpn	10629 [0.0%]	0 [0.0%]	3060 [0.0%]	0 [0.0%]	-100%	-0.0%
_WordCpy	0 [0.0%]	0 [0.0%]	0 [0.0%]	3689 [0.0%]	inf%	0.0%
__dyn_core_mod_NMOD_dyn_	0 [0.0%]	0 [0.0%]	3222 [0.0%]	0 [0.0%]	inf%	0.0%

Figure 4: Perfbrowser with HPMPROF-DIFF

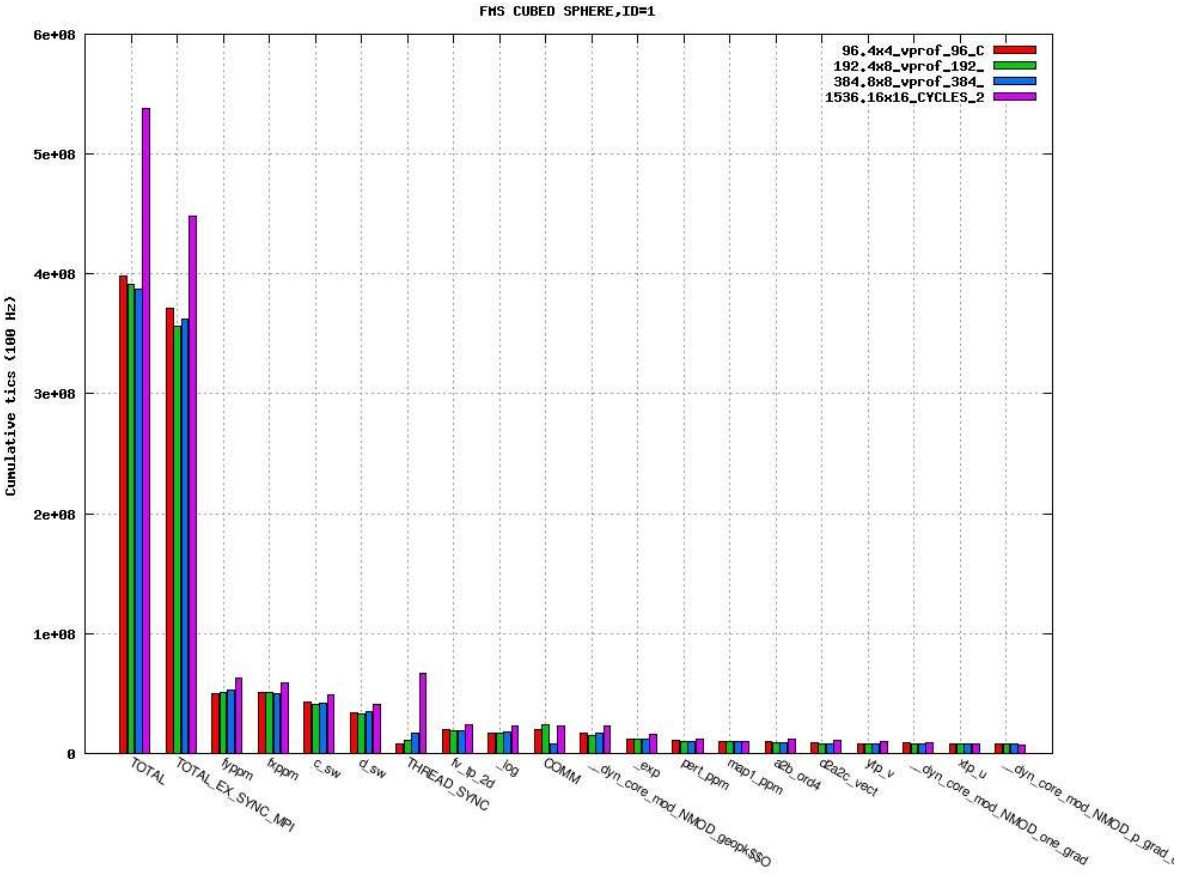
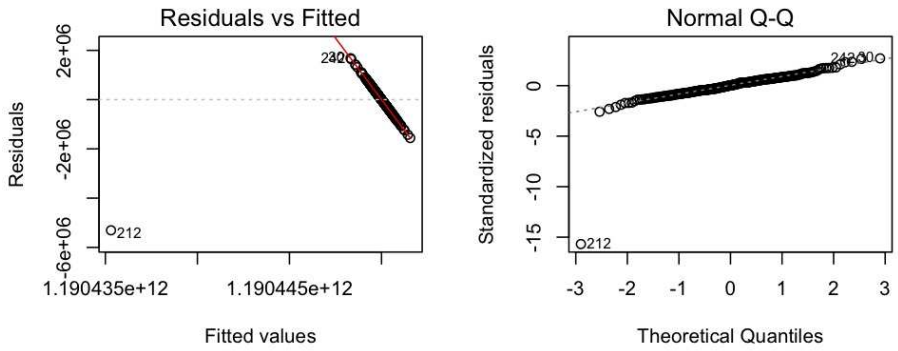
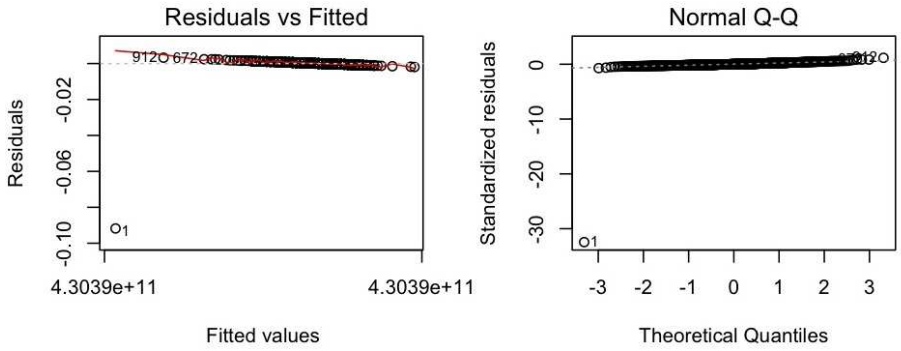


Figure 5: Perfbrowser with HPMPROF-PLOT

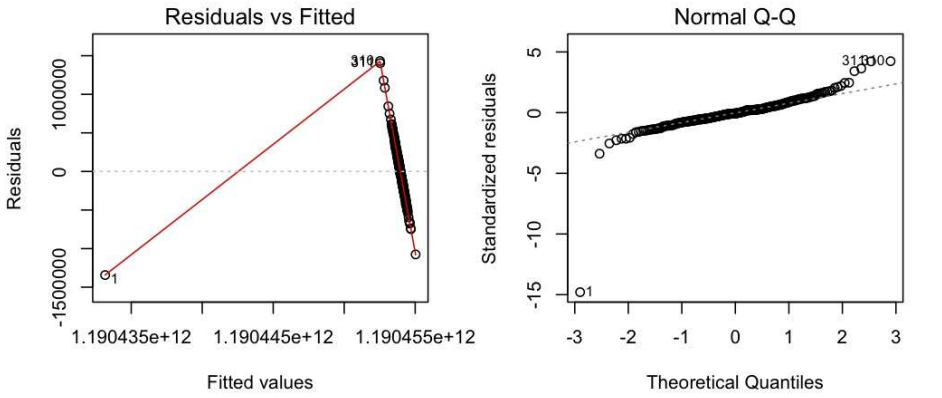


(a) 8×8

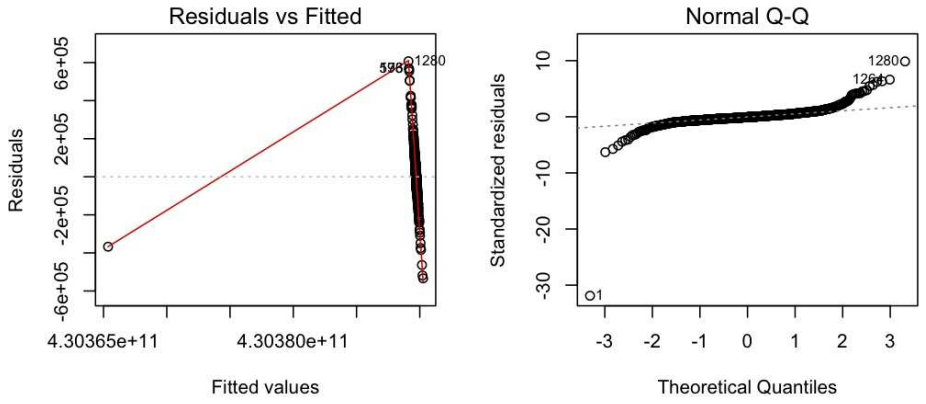


(b) 16×16

Figure 6: Residuals vs Fitted and Q-Q plots for two layouts using linear regression models.



(a) 8×8



(b) 16×16

Figure 7: Residuals vs Fitted and Q-Q plots for two layouts using cubic spline models.

	Model 4X4	Model 4X8	Model 8X8	Model 8X16	Model 16X8	Model 16X16
(Intercept)	4.712e+12	2.341e+12	1.190e+12	6.896e+11	6.962e+11	4.304e+11
Committed.Load.Misses.Knot1	1.113e+07	1.976e+06	1.194e+05	-5.000e+05	3.526e+05	3.482e+05
Committed.Load.Misses.Knot2	6.992e+06	1.500e+06	4.914e+05	-6.052e+05	-1.116e+05	2.312e+05
Committed.Cacheable.Loads.Knot1	-3.158e+07	-6.511e+06	-8.193e+06	-6.052e+05	-2.890e+06	-5.362e+05
Committed.Cacheable.Loads.Knot2	-1.492e+07	-2.946e+06	-5.166e+06	-1.317e+06	-2.301e+06	-1.830e+06
L1p.Miss.Knot1	-8.153e+06	-2.474e+06	-1.468e+06	2.696e+05	-1.353e+06	-1.127e+05
L1p.Miss.Knot2	-6.253e+06	-3.158e+06	-2.262e+06	-3.068e+05	-1.970e+06	-9.648e+04
All.XU.Instruction.Completions.Knot1	2.553e+07	4.415e+06	6.219e+06	-2.694e+05	1.528e+06	-2.606e+06
All.XU.Instruction.Completions.Knot2	9.544e+06	1.917e+06	5.191e+06	1.077e+06	2.139e+06	3.924e+04
All.AXU.Instruction.Completions.Knot1	2.071e+07	-1.007e+06	-7.069e+05	2.276e+06	1.056e+06	-1.516e+05
All.AXU.Instruction.Completions.Knot2	4.760e+06	-2.659e+06	-1.129e+05	1.392e+06	-8.800e+05	4.090e+05
FP.Operations.Knot1	-3.290e+07	-2.574e+06	3.396e+05	-3.271e+06	-1.234e+06	1.728e+05
FP.Operations.Knot2	-9.968e+06	1.732e+06	-3.832e+05	-1.484e+06	1.861e+06	-5.913e+05
Dep.Stalls.Knot1	2.103e+07	8.889e+06	1.217e+06	1.860e+05	-8.539e+06	-1.243e+06
Dep.Stalls.Knot2	1.410e+06	1.855e+06	1.087e+05	-4.715e+05	-4.260e+06	-9.074e+05
DS.THREAD.SYNC.Knot1	-3.410e+05	-1.505e+06	-1.991e+04	1.866e+07	2.161e+07	3.564e+07
DS.THREAD.SYNC.Knot2	-4.154e+06	-4.045e+04	-1.991e+04	3.045e+06	8.077e+06	6.303e+06
DS.COMM.Knot1	4.065e+06	4.272e+06	-1.181e+07	1.490e+07	1.275e+07	6.972e+05
DS.COMM.Knot2	-9.955e+05	-9.929e+05	-2.824e+06	1.869e+06	5.571e+06	5.553e+05
DS.THREAD.SYNC.COMM.Knot1	-1.583e+07	-6.059e+06	-1.444e+06	3.038e+05	1.303e+07	2.104e+06
FP.Grp1.Knot1	1.076e+09	-6.863e+08	-1.284e+08	-2.937e+07	3.976e+05	-3.207e+05
FP.Grp1.Knot2	5.540e+05	-6.341e+05	-4.787e+04	1.551e+05	-3.732e+05	3.974e+05
FP.THREAD.SYNC.Knot1	-3.401e+06	8.260e+03	-7.692e+04	3.769e+05		
FP.THREAD.SYNC.Knot2		-9.226e+05	2.173e+04			
FP.COMM.Knot1		1.424e+06	2.412e+05	1.119e+05	-1.561e+05	
FP.COMM.Knot2		1.098e+06	2.125e+05	1.545e+05	-3.912e+04	
FP.THREAD.SYNC.COMM.Knot1		6.892e+08	1.284e+08	2.992e+07	2.954e+05	
INST.XU.ALL.Knot1	-2.842e+07	-3.092e+07	-2.010e+07	-1.794e+07	-8.337e+06	-7.663e+06
INST.XU.ALL.Knot2	-5.499e+06	-1.020e+07	-5.722e+06	-6.596e+06	-3.713e+06	-1.453e+06
XU.THREAD.SYNC.Knot1	4.533e+06	1.106e+07	1.317e+07	1.487e+07	5.186e+06	1.063e+07
XU.THREAD.SYNC.Knot2	-2.718e+05	4.159e+06	3.464e+06	3.177e+06	1.561e+06	1.749e+06
XU.COMM.Knot1	1.669e+07	1.569e+07	4.091e+07	1.590e+06	1.461e+06	1.029e+06
XU.COMM.Knot2	7.110e+06	6.791e+06	7.807e+06	2.556e+06	1.381e+06	2.895e+05
XU.THREAD.SYNC.COMM.Knot1	2.541e+06	1.029e+07	2.962e+06	3.785e+06	1.260e+06	5.658e+05
L1D.Misses.Knot1	5.177e+06	-6.338e+05	4.655e+06	1.249e+06	-1.441e+05	4.873e+03
L1D.Misses.Knot2	6.158e+05	1.118e+06	9.111e+05	1.597e+06	6.306e+05	2.695e+05
L1D.THREAD.SYNC.Knot1	-1.595e+06	1.360e+05	-2.207e+05	-2.460e+05	-2.032e+05	3.176e+04
L1D.THREAD.SYNC.Knot2	5.757e+05	-9.706e+04	5.159e+04	-1.456e+05	-5.118e+04	2.145e+04
L1D.COMM.Knot1	-5.169e+06	5.484e+05	-5.352e+05	-2.768e+05	-2.910e+05	4.502e+04
L1D.COMM.Knot2	-2.346e+06	1.840e+04	-2.872e+05	-2.807e+05	-1.122e+05	4.635e+04
L1D.THREAD.SYNC.COMM.Knot1	-4.546e+06	1.709e+05	-1.914e+05	-1.326e+06	-3.201e+05	8.460e+04
L1P.Misses.Knot1	6.237e+06	3.011e+04	1.390e+06	2.569e+05	1.555e+06	1.563e+05
L1P.Misses.Knot2	3.338e+06	1.200e+06	1.338e+06	7.449e+04	1.221e+06	7.791e+04
L1P.THREAD.SYNC.Knot1	2.484e+05	3.196e+05	-1.706e+05	-1.234e+05	-3.503e+05	-1.385e+04
L1P.THREAD.SYNC.Knot2	5.989e+05	-9.041e+05	-1.051e+05	-1.160e+05	-2.703e+05	-3.582e+04
L1P.COMM.Knot1	-2.189e+06	1.092e+06	2.682e+05	-2.774e+05	-6.043e+05	-1.800e+05
L1P.COMM.Knot2	-4.523e+04	1.258e+06	-5.249e+05	-6.066e+04	-4.107e+05	-8.129e+04
L1P.THREAD.SYNC.COMM.Knot1	-7.269e+06	1.476e+06	-1.403e+06	-6.641e+04	-7.491e+05	-1.931e+05
R^2	0.98	0.94	0.94	0.97	0.91	0.98
Adj. R^2	0.95	0.90	0.93	0.97	0.90	0.98
MSE	3.23E+12	9.11346E+11	3.33689E+11	56889909990	1.40E+11	1.00E+10

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Table 6: Cubic Spline Models for layouts $4 \times 4, 4 \times 8, 8 \times 8, 8 \times 16, 16 \times 8, 16 \times 16$. The variables are of the form <variable>.knot1 and <variable>.knot2 which correspond to the breakpoints (mean) that define the spline. Note that not all the variables appear in the cubic spline models. In general, if the coefficient of an attribute is not displayed, it does not occur in the model.

4 Appendix

4.0.4 Effect of size of train-test data set on linear model performance:

In order to test how the R^2 and Mean Square Error of the linear regression models are affected by the size of the train-test data set, we performed a set of experiments by developing linear regression models for the following configurations $4X4, 4X8, 8X8, 8X16, 16X8, 16X16$ using a reduced set of attributes. Table 7 lists the attributes used in these models. The examples in the dataset are split into train-test pairs by randomly sampling $x\%$ percentage for training. Three different percentages of x were used – $50, 70,$ and 90. The generation of the train-test pairs was repeated five times to study sensitivity of the regression coefficients to sampling and effects on generalization performance. The results from these experiments are reported in Table 8. In general, it was noted that there was not much variation if 50%, 70% or 90% of the data was used for training models. Following this, the experiments with a larger set of attributes were performed by randomly assigning 70% of the data as training and the rest as a test set.

1	Committed Load Misses
2	Committed Cacheable Loads
3	L1p Miss
4	All XU Instruction Completions
5	All AXU Instruction Completions
6	FP Operations
Response	Cycles

Table 7: Predictor and Response Variables used in preliminary linear and non-linear machine learning models.

		50—50		70—30		90—10	
Confgn.	Trial No.	R^2	MSE	R^2	MSE	R^2	MSE
$4X4$	1	0.43	7.01E-06	0.51	2.04E-06	0.49	3.53E-06
	2	0.49	9.48E-06	0.47	8.55E-06	0.50	1.77E-05
	3	0.51	9.58E-06	0.51	2.04E-06	0.49	2.38E-05
	4	0.49	9.76E-06	0.52	5.29E-06	0.50	5.34E-06
	5	0.51	1.21E-05	0.51	6.91E-07	0.50	6.68E-06
$4X8$	1	0.49	4.43E-06	0.49	5.21-05	0.50	8.77E-05
	2	0.50	5.78E-06	0.5	4.45E-05	0.50	9.75E-05
	3	0.49	3.74E-06	0.50	4.16E-05	0.50	1.03E-04
	4	0.49	2.14E-06	0.50	5.24E-05	0.50	1.12E-04
	5	0.49	1.08E-05	0.50	5.75E-05	0.50	1.08E-04
$8X8$	1	0.50	6.88E-06	0.50	2.48-06	0.49	6.47E-05
	2	0.49	7.67E-06	0.50	2.37E-06	0.5	6.585E-05
	3	0.49	6.65E-06	0.50	3.71E-06	0.50	5.83E-05
	4	0.50	7.35E-06	0.50	3.37E-06	0.5	7.00E-05
	5	0.49	6.28E-06	0.49	1.71E-06	0.50	7.23E-05
$8X16$	1	0.5	2.28E-05	0.49	5.34-05	0.49	6.30E-05
	2	0.49	2.24E-05	0.49	5.31E-05	0.50	6.29E-05
	3	0.49	2.18E-05	0.50	5.24E-05	0.50	6.28E-05
	4	0.50	2.01E-05	0.50	5.21E-05	0.49	6.08E-05
	5	0.50	2.20E-05	0.50	5.11E-05	0.5	6.35E-05
$16X8$	1	0.5	7.23E-06	0.5	6.65E-05	0.50	9.41E-05
	2	0.50	7.13E-06	0.5	6.60E-05	0.49	9.09E-05
	3	0.50	7.00E-06	0.49	6.78E-05	0.49	9.26E-05
	4	0.50	7.23E-06	0.50	6.97E-05	0.50	3.60E-05
	5	0.49	6.65E-06	0.50	6.92E-05	0.5	9.19E-05
$16X16$	1	0.49	9.54E-07	0.5	1.32-04	0.5	1.67E-04
	2	0.50	9.03E-07	0.5	1.29E-04	0.5	1.66E-04
	3	0.50	8.87E-07	0.5	1.32E-04	0.5	1.66E-04
	4	0.49	1.10E-06	0.5	1.31E-04	0.5	1.64E-04
	5	0.5	9.98E-07	0.5	1.35E-04	0.5	1.64E-04

Table 8: Experiments to study the sensitivity of R^2 and MSE values from linear regression models for different sizes of train-test data.