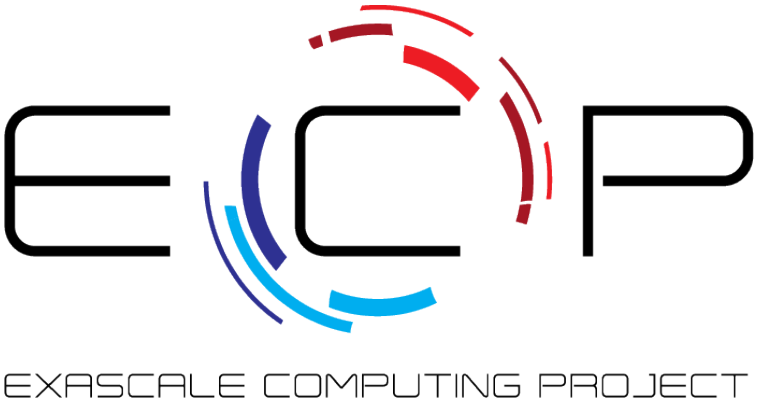
Quantitative Performance Assessment of Proxy Apps and Parents

Report for ECP Proxy App Project Milestone AD-CD-PA-1040

David Richards[1], Omar Aaziz[2], Jeanine Cook[2], Hal Finkel[3], Brian Homerding[3], Tanner Juedeman[2], Peter McCorquodale[4], Tiffany Mintz[5], and Shirley Moore[5]

[1]Lawrence Livermore National Laboratory, Livermore, CA
[2]Sandia National Laboratories, Albuquerque, NM
[3]Argonne National Laboratory, Chicago, IL
[4]Lawrence Berkeley National Laboratory, Berkeley, CA
[5]Oak Ridge National Laboratory, Oak Ridge, TN

April 24, 2018



LLNL-TR-750182

Executive Summary

This report completes the AD-CD-PA-1040 Milestone:

> We will develop a quantitative methodology to compare the fidelity of the ECP proxy applications with respect to the parent ECP application they represent. Fidelity includes comparison of appropriate dynamic execution characteristics (e.g., memory characteristics for a memory-matching proxy), computational requirements, and hardware bottlenecks. This methodology will include metrics and platform specifications (e.g., tools, specific systems), and will be applied to evaluate 4 ECP proxy applications.

We have satisfied this milestone by developing an unsupervised machine learning methodology that uses hardware performance counter-derived metrics as input to a clustering model that outputs similarity of the proxy/parent pairs; principal component analysis (PCA) reduces the dimensionality of the data prior to input to the clustering model. We use the following hardware performance counter metrics as input to the PCA:

- Instructions per cycle (IPC).
- Micro-ops per cycle (UIPC).
- Cache miss rates and ratios at various levels of the hierarchy.
- Fraction of cache bandwidth used at various levels.
- Instruction mix — Floating point, load, store, branch, and other (mostly integer) instructions. We compute each instruction category as a percentage of the total instructions committed.

In addition to these metrics, we use FLOPS/instruction and arithmetic intensity to characterize hardware bottlenecks of each proxy/parent pair. As described in Section 3, data was collected using gprof, HPCToolkit, and LDMS on various Intel Haswell-based systems. We have applied this methodology to four ECP proxy applications and their respective parents:

- SW4lite and SW4 (seismic modeling)
- Nekbone and Nek5000 (thermal transport)
- SWFFT and HACC (cosmology/FFT)
- ExaMiniMD and LAMMPS (molecular dynamics)

From this work, we conclude the following: *The four target proxy applications are indeed good representations of the computation and memory behavior of their respective parent applications.* Our methodology did not adequately assess the representativeness of communication across these proxy/parent application pairs, so we made no conclusions with respect to similarity. Comparison of communication patterns will be addressed in future milestones.

From the characterization data presented in Sections 3.4 and 4.3, we conclude that all of the proxies examined are acceptably similar to their parents with respect to cache behavior, relatively low cache bandwidth usage, and relatively low arithmetic intensity that indicates a relatively high data movement per floating-point operation. However, because of known issues with the Haswell Performance Monitoring Unit, the conclusions pertaining to arithmetic intensity may be inaccurate. We will update these conclusions in a future report after further exploration on an alternate platform.

1 Introduction

The purpose of the ECP Proxy Applications Project is to improve the quality of proxies created by ECP and maximize the benefit received from their use. To accomplish this goal, we have assembled and curated an initial ECP proxy app suite consisting of proxies developed by other DOE/ECP projects that are intended to represent the most important features (particularly performance) of exascale applications. To both improve the quality of these proxies and maximize benefit from their use, we must understand if the selected proxies accurately represent the intended characteristics of their parent applications (e.g., memory, computation, communication, other).

To date, we have completed an initial performance characterization, including dynamic profiling and hardware bottleneck analysis where appropriate, and have initial results on the machine learning-based methodology that we have developed to compare proxy to parent applications. The primary proxy/parent applications that we use in this work are:

- SW4lite and SW4 (seismic modeling)
- Nekbone and Nek5000 (thermal transport)
- SWFFT and HACC (cosmology/FFT)
- ExaMiniMD and LAMMPS (molecular dynamics)

In addition to the four target proxy/parent pairs noted above, we also performed some evaluation of the following proxies:

- CoMD (molecular dynamics)
- miniFE (finite element)
- XSBench (Monte Carlo neutronics)

A description of each of the proxies that we use in this study, including problems, problem sizes, scaling configurations, and mapping to ECP applications is presented in Section 2; Section 3 presents a performance characterization of the all of the (seven) proxy applications, including the parent applications that map to the four target proxies. In Section 4 we present our methodology for quantitatively comparing each of the four target proxies to their respective parent applications, then continue to present the results of the application of this methodology to each of the proxy/parent pairs; Section 5 presents our conclusions from this work and our planned future work.

2 ECP Proxy Applications and Problem Space Mapping

Version 1.0 of the ECP proxy application suite[1]contains 13 proxy applications, most of which were developed prior to the ECP project. At the time of the curation of the initial suite, few proxies were available from ECP applications. Therefore, many of the current proxies do not have a direct ECP parent application that they are intended to represent. For this work, we chose existing proxies that map to ECP applications in development or that are being used as components in workflow integration. The versions of each of the four proxy/parent pairs analyzed in this report are shown in Table 1 This section containes detailed information on each proxy/parent pair including the intended scope of the proxy as well as representative problem sizes (Table 2) and scaling configurations for both proxies and parents.

[1]The current version of the suite can be found at
http://proxyapps.exascaleproject.org/ecp-proxy-apps-suite

Proxy	Version	Parent	Version
SW4lite	2.0	SW4	2.0
Nekbone	3.1	Nek5000	17
SWFFT	1.0	HACC	1.0
ExaMiniMD	1.0	LAMMPS	17 Aug 2017

Table 1: Proxy/Parent version information

Proxy/Parent	Problem/Input size
SW4lite/SW4	LOH.1-h50.in, LOH.2-h50, time=5 (single-node) LOH.1-h50.in, LOH.1-h50, time=9 (multinode)
Nekbone	Dim=3; polynomial order=8; spectral multigrid=off max local elements per MPI rank=300
Nek5000	eddy_uv, with Dim=3; polynomial order=8 max local elements per MPI rank=300
SWFFT	n_repetitions=100; ngx=1024
HACC	steps=100; ngx=1024
ExaMiniMD/LAMMPS	units=lj; nx, ny, nz=100; Timestep=0.005; Run=18000 (single- and multinode)
ExaMiniMD	units=SNAP; nx, ny, nz=100; Timestep=0.005; Run=18000 (single-node)
miniFE	nx=420, ny=420, nz=420
XSBench	-s large -l 600000000 -G unionized

Table 2: Proxy/Parent Problems/Input Sizes

2.1 ExaMiniMD

ExaMiniMD is a proxy application and research vehicle for Molecular Dynamics (MD) applications such as LAMMPS. ExaMiniMD is being used in the ECP Co-design Center for Particle Applications (CoPA) and in the ECP Ristra project, which is an ATDM code project at LANL. LAMMPS is being used in the ECP Molecular Dynamics at the Exascale with EXAALT (EXascale Atomistics for Accuracy, Length and Time) project.

Compared to previous MD proxy apps (MiniMD, CoMD), the design of ExaMiniMD is significantly more modular. The main components such as force calculation, communication, neighbor list construction and binning are derived classes whose main functionality is accessed via virtual functions. This allows a developer to write a new derived class and drop it into the code without touching much of the rest of the application.

ExMiniMD's parent application is LAMMPS. Like LAMMPS, ExaMiniMD uses spatial domain decomposition. That is, each individual processor in a cluster owns a subset of the simulation box. Both LAMMPS and ExaMiniMD allow users to specify a problem size, atom density, temperature, timestep size, number of timesteps to perform, and particle interaction cutoff distance. But compared to LAMMPS, ExaMiniMD's feature set is extremely limited, and only two types of interactions (Lennard-Jones/ EAM) are available. No long-range electrostatics or molecular force field features are available.

ExaMiniMD uses neighbor lists for the force calculation, as opposed to cell lists, which are employed by, for example, CoMD. The neighbor list approach (or variants of it) is used by most commonly used MD applications, such as LAMMPS, Amber, and NAMD. Cell lists are employed by some specialized codes, in particular for very large scale simulations which might be memory capacity limited.

For the studies presented here, we use the Lennard-Jones interaction, with dimensions set as

shown in Table 2. Sizes were chosen based on conversations with both ExaMiniMD and LAMMPS developers. We also chose sizes that could be appropriately used in scaling studies.

Note that ExaMiniMD and LAMMPS have implemented a new interaction that is a much more complicated and computationally expensive potential that attempts to approach quantum chemistry accuracy when modeling metals and other materials. Because we learned about this interaction potential too late to include results in this report, the next proxy/parent assessment milestone will include data using this potential.

2.2 Nekbone

Nekbone is a proxy app for Nek5000, which is a spectral element code designed for large eddy simulation (LES) and direct numerical simulation (DNS) of turbulence in complex domains. Nek5000 and Nekbone are being used in several ECP projects including Multiscale Coupled Urban Systems, ExaSMR, and the Center for Efficient Exascale Descretizations (CEED).

Nek5000 is a thermal hydraulic code that simulates thermal transport on a full range of scales set by the geometry encountered within a reactor. The spectral element method provides an efficient means of reducing numerical dispersion and dissipation errors while retaining the geometric flexibility needed to represent the complex coolant passageways. Nek5000 has a broad range of applications including vascular flow, ocean modeling, combustion, heat transfer enhancement, stability analysis and MHD (magnetohydrodynamic) flows.

Nekbone reportedly implements the computationally intensive linear solvers that account for a large percentage of the Nek5000 run time, as well as the communication costs required for nearest-neighbor data exchanges and vector reductions. Therefore, our assumption according to the documentation (and the cited milestone report) is that Nekbone in its entirety can be used as a faithful representation of the computation, memory behavior, and communication of that in Nek5000. The Nekbone kernel is embedded in a conjugate gradient iteration to solve the 3D Poisson equation. Preconditioning is either a simple diagonal scaling (simpler than Nek5000) or a spectral element multigrid on a block or linear geometry which is more similar to the multigrid structure found in Nek5000. The Nekbone kernel implements the matrix-vector product at the heart of the spectral element method.

The problem size information in this report is derived from work performed on Nekbone in the ExaSMR project [12]. These trials were for single node performance, primarily on the Intel Xeon Phi and GPU using an OpenACC port of Nekbone. Domains were brick-like 3D arrangements of cubic elements and the problem sizes were:

- Polynomial order parameter (nx1): 8 or 16
- Number of elements (nelt): up to 16384 (when nx1=8) and up to 2048 (when nx1=16).

or strong scaling studies, the report suggests a maximum local problem size of approximately 4000 local elements for GPU experiments, 500–1000 local elements for KNL, and fewer than 500 for CPU only experiments.

We had a very difficult time determining how to map an equivalent problem across Nekbone and Nek5000. Despite much interaction with developers, it is still unclear that we have done this correctly. We were able to map most of the simpler parameters, but mapping geometry and computational algorithm was very unclear. From our communication with developers for Nek5000, we chose the eddy_uv problem. However, this is a 2D solution to the Navier-Stokes equations, where Nekbone implements Poisson. Further, we mistakenly set the geometry in Nekbone to 3D rather than 2D, and we ran Nekbone with spectral element multigrid off. The difference this makes

Case	Ngp	Nts	Nodes	MPI-tasks	OMP-threads/task
LOH.1-h50	1.23e8	1073	8	256	N/A
LOH.2-h50	—	—	8	256	N/A
LOH.1-h50	—	—	8	256	1
—	—	—	—	128	2
—	—	—	—	64	4
LOH.2-h50	1.23e8	1073	8	256	1
—	—	—	—	128	2
—	—	—	—	64	4

Table 3: Scaling configurations for SW4 and SW4lite

in the underlying hardware behavior is addressed in Section 4.2. We will address this issue in future milestone reports.

2.3 SW4lite

SW4lite is a bare bones version of the SW4 seismic modeling code that is intended for testing performance optimization of key numerical kernels, particularly with respect to memory layout and threading. SW4 and SW4lite are being used exclusively by the High Performance, Multidisciplinary Simulations for Regional Scale Earthquake Hazard and Risk Assessments (EQSIM) ECP Project.

McCallen and co-authors [11] compare baseline performance of SW4 and SW4lite using two problem cases to show that SW4lite can be used to make performance enhancements in SW4. Based on their work, we assume that SW4lite is representative of the computation, communication, and memory behavior of SW4.

SW4lite supports MPI only and hybrid MPI+OpenMP programming models. There is also a CUDA version for GPUs. The inputs that are relevant for performance testing are the LOH1 and LOH2 test cases, which simulate ground motion in a material model consisting of a layer of soft material on top of a bedrock halfspace (referred to as a Layer-Over-Halfspace, or LOH model). The LOH1 and LOH2 cases both use an isotropic elastic model. The LOH1 case represents a small earthquake with a point moment tensor source term. The LOH2 case models a larger earthquake that results from slip over a finite fault plane, which is represented by 3200 discretized point moment sources. The domain is discretized by one Cartesian grid.

SW4 can be run with the same LOH1 and LOH2 inputs—they are actually identical files to those found in the SW4lite distribution. SW4 also has some larger, more realistic problems (Hayward and Berkeley inputs), but SW4lite can not accommodate these. SW4 was originally written as an MPI-only code. However, an MPI+OpenMP implementation is being developed. At this point, we only compare the MPI-only implementation, although the scaling table below (Table 3) for the Haswell (shepard) architecture shows values for both programming models.

2.4 SWFFT

SWFFT is the 3D, distributed memory, discrete fast Fourier transform from the Hardware Accelerated Cosmology Code (HACC). SWFFT and HACC are used in the ECP ExaSky project.

The main SWFFT build is an MPI implementation. There is also an MPI+OpenMP build that uses the openMP version of the fftw3 library. Currently HACC has three total copies of the

<Run Command for 1 nodes with 128 threads> ./comd -i 4 -j 4 -k 8 -x 50 -y 50 -z 200 #atoms: 1000000
<Run Command for 2 nodes with 256 threads> ./comd -i 4 -j 4 -k 16 -x 50 -y 50 -z 400 #atoms: 2000000
<Run Command for 4 nodes with 512 threads> ./comd -i 4 -j 4 -k 32 -x 100 -y 100 -z 200 #atoms: 4000000
<Run Command for 8 nodes with 1024 threads> ./comd -i 8 -j 8 -k 16 -x 100 -y 100 -z 400 #atoms: 8000000
<Run Command for 16 nodes with 2048 threads> ./comd -i 8 -j 8 -k 32 -x 100 -y 100 -z 800 #atoms: 16000000
<Run Command for 32 nodes with 4096 threads> ./comd -i 8 -j 8 -k 64 -x 200 -y 200 -z 400 #atoms: 32000000
<Run Command for 64 nodes with 8192 threads> ./comd -i 16 -j 16 -k 32 -x 200 -y 200 -z 800 #atoms: 64000000
<Run Command for 128 nodes with 16384 threads> ./comd -i 16 -j 16 -k 64 -x 200 -y 200 -z 1600 #atoms: 128000000
<Run Command for 256 nodes with 32768 threads> ./comd -i 16 -j 16 -k 128 -x 400 -y 400 -z 800 #atoms: 256000000
<Run Command for 512 nodes with 65536 threads> ./comd -i 32 -j 32 -k 64 -x 400 -y 400 -z 1600 #atoms: 512000000
<Run Command for 1024 nodes with 131072 threads> ./comd -i 32 -j 32 -k 128 -x 400 -y 400 -z 3200 #atoms: 1024000000
<Run Command for 2048 nodes with 262144 threads> ./comd -i 32 -j 32 -k 256 -x 800 -y 800 -z 1600 #atoms: 2048000000
<Run Command for 4096 nodes with 524288 threads> ./comd -i 64 -j 64 -k 128 -x 800 -y 800 -z 3200 #atoms: 4096000000
<Run Command for 8192 nodes with 1048576 threads> ./comd -i 64 -j 64 -k 256 -x 800 -y 800 -z 6400 #atoms: 8192000000
<Run Command for 16384 nodes with 2097152 threads> ./comd -i 64 -j 64 -k 512 -x 1600 -y 1600 -z 3200 #atoms: 16384000000
<Run Command for 32768 nodes with 4194304 threads> ./comd -i 128 -j 128 -k 256 -x 1600 -y 1600 -z 6400 #atoms: 32768000000
<Run Command for 65536 nodes with 8388608 threads> ./comd -i 128 -j 128 -k 512 -x 1600 -y 1600 -z 12800 #atoms: 65536000000

Table 4: Trinity scaling study for CoMD.

double complex grid, two of which do the out-of-place backward transform. SWFFT replicates the transform and is representative of the computation and communication involved.

The main parameters to SWFFT are:

- n_repetitions: number of repetitions of FFT.
- ngx: Number of grid vertexes along one side. Should be a number that is near the cube root of (∼3.5% of total RAM/16) with small prime factors.
- ngy ngz: optional to run non-cubic DFFT (HACC does not use this feature but useful in creating representative problem space).
- Python code available (also under development) to suggest grid sizes based off total RAM.

The application will scale with the size of the double complex grid.

We communicated directly with a developer on the HACC team and were directed in terms of the problem sizes for both HACC and SWFFT shown in Table 2.

2.5 CoMD

CoMD (`https://github.com/ECP-copa/CoMD`) is a long standing and much analyzed and modified reference implementation of typical classical molecular dynamics algorithms and workloads. It implements two types of interactions (Lennard-Jones and EAM) and uses cell lists for the force calculation. Quite recently, CoMD was used to study scaling and optimization opportunities on large scale KNL clusters. The following runs were provided by the development team and are documented in Table 4. From these runs we are able to determine that a good weak scaling study would be to have approximately 1,000,000 atoms for a given Trinity node and to rely on on-node multithreading.

2.6 miniFE

MiniFE is a proxy application that represents operations in implicit finite element codes. It uses a un-preconditioned conjugate gradient solver and sparse linear algebra motifs that are typical of several ECP application projects including Candle, ExaFEL, GAMESS, Urban, ExaStar, ExaBiome, MFIX, and ExaSGD.

For miniFE, the recommended size is 120 cubed per core. This is slightly less than 1 GB per core. For example:

mpirun -np 64 ./miniFE.x -nx 640 -ny 640 -nz 640

This size problem uses slightly less than 1 GB per core, which is large enough that the solver is not running in cache. It is also small enough to allow for the added complexity of the codes.

2.7 XSBench

XSBench is a mini-app representing a key computational kernel of Monte Carlo neutronics applications such as OpenMC. The proxy supports an openMP version and an MPI+openMP version. There are additional build options for verification and profiling. XSBench models the most time intensive part of a typical MC reactor core transport algorithm, the calculation of macroscopic neutron cross sections [18]. The kernel accounts for around 85% of the total runtime of OpenMC [14].

Default run parameters are representative of the parent application, the user can adjust -l to add more time without affecting the memory access patterns or footprint.

- -s (size) defaults to large. The XL and XXL options do not directly correspond to a physical model.
- -g (gridpoints) defaults to 11,303. Corresponds to the average number per nuclide in OpenMC H-M Large Model.
- -G (grid type) defaults to unionized (instead of nuclide). Unionized is typically used in Monte Carlo codes (faster speed, significant increase in memory usage).
- -l (lookups) defaults to 15,000,000. Can increase to wash out time spent on initialization or increase runtime for performance counter purposes.

Table size will not change as the problem scales.

3 Quantitative Performance Characterization

As a first step toward developing a quantitative comparison methodology for proxy/parent application pairs, we did an initial characterization of the behavior of the proxy apps (but not the parents) on a Haswell system. We started with dynamic function profiling, then continued with more in-depth characterization using hardware performance counters. Communication was not included in this initial characterization, but we do include it in Section 4. This initial characterization was a precursor step in the development of the comparison methodology that we ultimately developed and executed.

We began the characterization with dynamic function profiling using *gprof*, then continued with collecting hardware performance counter data on the entire execution of each proxy and on the functions that were accountable for the largest percentage of execution. The dynamic execution profiles presented are those we collected during larger, distributed runs. Further, we produced these dynamic profiles for both the proxies and their respective parents. Although we do not present it here, we did collect dynamic execution profiles for single-node runs as well. Dynamic execution profiles only varied slightly between single-node and larger distributed executions. This is important to note because all of the hardware counter-based characterization data presented in this section was extracted from single-node runs. Hardware performance counter data for larger distributed runs is presented in Section 4.

3.1 Methodology

Although this Milestone requires comparison of only four proxy/parent pairs, we present characterization data for seven proxies and four parent applications. For performance characterization, we look at two primary aspects:

1. Dynamic execution time: Here we use the dynamic profiling tool *gprof [8]* to understand in which functions an application is expending some percentage of total execution time. We use these profiles for two primary purposes: (1) to drive our per-function, single-node hardware counter characterization, and (2) for comparing these profiles between proxy/parent pairs to gain some understanding of the basic performance and functional similarity between the two. This also helps with identifying where to look in the actual code to understand the implementation.
2. Node and memory behavior: Hardware performance counter sampling and instrumentation is used to provide insight on many aspects pertaining to node, cache, and memory behavior. For the comparison methodology presented in Section 4, we use a much broader event/metric set, but we constrain characterization to the following metrics:
    - Cache miss rates at various levels of the hierarchy
    - Fraction of memory bandwidth used
    - Fraction of cache bandwidth used at different levels
    - FLOPS/instruction
    - Arithmetic intensity (FLOPS/DRAM bytes). Here we compute arithmetic intensity using data produced by HPCToolkit.

Note that to compute the fraction of memory and cache bandwidth used, we used the Roofline Model [3] that is implemented in the CS Roofline Toolkit [4] to obtain the maximum bandwidths that our experimental machine can achieve. Using this in conjunction with hardware performance counter data, we can compute the fraction of bandwidth actually used.

3.2 Measurement Platform

The choice of a measurement platform was primarily motivated by the availability of systems across the four national labs involved in this assessment work (LBNL, ORNL, ANL, SNL). We chose the Intel Haswell architecture for the hardware platform. This was readily available either at NERSC (where we have a project system allocation) or at the individual labs. Although Haswell is not a new platform, it is currently installed in the Trinity machine, which is the latest acquisition at LANL. Table 5 shows the architectural details of this platform.

We attempt to keep the compiler constant across all of the platforms so that results are comparable. The compiler we chose is Intel 18.0.1.163. All compiler flags for a particular proxy/parent are kept the same across all experimentation, again, so that results are comparable.

To gather dynamic function profiling information and hardware performance counters, we used various tools. We chose not to constrain the entire assessment team to using a fixed suite of tools, but rather let partners choose their tool of choice for the various tasks. We used this opportunity to cross-validate results across different tools.

For dynamic function profiling, we use *gprof*. Gprof is a gnu, open source profiling tool that can generate profiles for parallel and serial code executions; for parallel execution, gprof can generate a profile for every process. Gprof generates three different types of profiles:

1. The flat profile shows how much time a program spent in each function, and how many times

Component	Details
μop cache	1536 μops, 8 way, 6 μop line size, per core
L1 data cache	32 KB, 8 way, 64 sets, 64 B line size per core
L1 instruction cache	32 KB, 8 way, 64 sets, 64 B line size, per core
L2 cache	256 kB, 8 way, 512 sets, 64 B line size, per core
L3 cache	2–45 MB, 12–16 way, 64 B line size, shared
Memory (per node)	128 GB DDR4-2133 MHz (64GB per socket)
Cores/threads	16/32
Sockets/node	2
Total nodes	32
Interconnect	Mellanox FDR Infiniband
Max Memory BW	68 GB/sec

Table 5: Hardware Characteristics of Haswell Platform

that function was called. This provides a concise catalog of which functions burn most of the cycles.

2. The call graph shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function.
3. The annotated source listing is a copy of the program's source code, labeled with the number of times each line of the program was executed.

For this work, we report only the flat profile.

In addition to dynamic function profiling, we also use the Haswell Performance Monitoring Unit (PMU) and its hardware performance counters to measure: (1) cache behavior and bandwidth utilization at the various levels of the cache hierarchy, and (2) instruction mix characteristics such as floating-point, integer and memory operations, and branch instructions (Section 4 contains additional discussion of the Haswell PMU). We primarily use two tools to collect these measurements—HPCToolkit [9] and LDMS [1].

HPCToolkit is an integrated suite of tools for measurement and analysis of program performance on computer systems ranging from multicore desktop systems to the nation's largest supercomputers. By using statistical sampling of timers and hardware performance counters, HPCToolkit collects accurate measurements of a program's work, resource consumption, and inefficiency and attributes them to the full calling context in which they occur. HPCToolkit works with multilingual, fully optimized applications that are statically or dynamically linked and supports measurement and analysis of serial codes, threaded codes (e.g. pthreads, OpenMP), MPI, and hybrid (MPI+threads) parallel codes. Although the tool supports standard metric measurements, we use the facility that enables direct input of specific perf [13] events. Perf (or perf tool) is a standard Linux facility that provides an interface to the system's hardware performance counters through its PMU.

LDMS (Light-weight Distributed Metric System) is a low-overhead, total system tool that enables scalable monitoring of large-scale computer systems and applications. It comprises a monitoring core and a collection of plug in samplers, each of which is designed to measure a specific component or behavior of the system or application. LDMS takes advantage of RMA (remote memory access), a capability on many network interfaces for directly accessing a designated portion of memory and delivering its contents across the network, without the sending node being

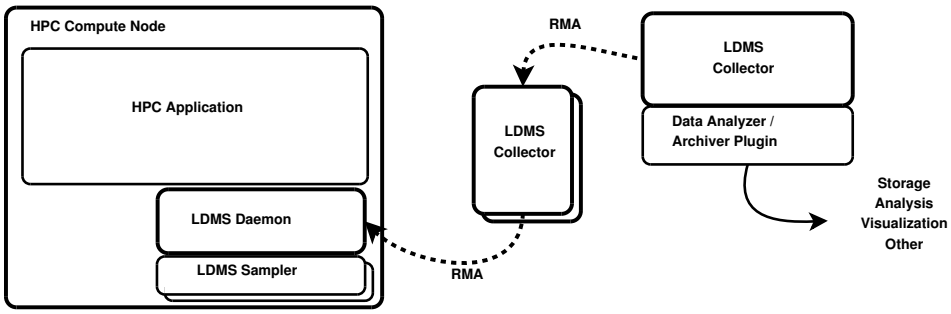


Figure 1: LDMS Architecture

interrupted at the processor or O/S level. This is an ideal capability for HPC monitoring purposes since the application can keep on running while the locally created monitoring data is delivered off-node using RMA.

LDMS consists of several components, shown in Figure 1. Local daemon processes on each compute node manage local monitoring data collection and interact with specific sampler plugins.

A sampler is responsible for collecting a particular metric. The local LDMS daemon is configured with the sampling rate, and at that rate it notifies each registered sampler to update its metric with the most recent measurement sample. LDMS implements samplers to measure:

1. Network-related information: Congestion, delivered bandwidth (total), operating system traffic bandwidth, average packet size, and link status
2. Shared file system information (e.g., Lustre): open, closes, reads, writes
3. Memory related information: current free, active
4. CPU information: utilization (user, sys, idle, wait)
5. MPI information: all mpiP metrics (Section 3.1)
6. PAPI events: hardware event counters that the PAPI (Performance Application Programming Interface) [10] interface can access, arranged in the form of menus based loosely on hardware components (e.g., branch predictor, cache hierarchy, execution units). This sampler automatically recognizes when an HPC job is started and a aches to the correct processes and collects the metrics, without needing to run the application using any of the command-line or other HW-compatible tools.

In this work, we use the PAPI sampler. Although a sampler to collect MPI information exists, it has yet to be fully tested. Therefore, to collect MPI communication information, we use the mpiP library directly.

The Empirical Roofline Tool (ERT) within the CS Roofline Toolkit is used to measure theoretical peak bandwidths in the cache and memory hierarchy in order to determine the percentage of peak that an application actually utilizes. The ERT automatically generates roofline data for a given computer. This includes the maximum bandwidth for the various levels of the memory hierarchy and the maximum GFLOP rate. This data is obtained using a variety of "micro-kernels". The Roofline Toolkit and ERT are based on the Roofline Model, which is a visually intuitive performance model used to bound the performance of various numerical methods and operations running on a variety of computer architectures.

3.3 Dynamic Profiling

We first present dynamic profiling results for the four proxy/parent pairs that we target for comparison in this milestone. Again note that we profiled both proxies and parents and these profiles are extracted from multinode, distributed parallel runs (although serial and single-node profiles remain essentially equivalent). Problem and/or input sizes used for both proxies and parents are shown in Table 2. Table 1 shows the precise version of each of the proxies and parent applications used in this work.

Figure 2 presents dynamic function profiles of the four target proxy/parent pairs collected using gprof. Each of the plots shows the functions that account for approximately 100% of the program execution time. The SWlite and SW4 profiles match fairly well as expected, since SW4lite closely mirrors SW4 and the problem inputs are identical. They both spend the majority of execution time in a function called *rhs4th3fortsgstr* that computes a one-sided approximation of the spatial operator in the elastic wave equation, which must be the computational kernel of the application.

The Nekbone and Nek5000 profiles differ significantly. The *mxmf2* function, which appears in both profiles, is a computational kernel of some sort, however, neither of these codes is presently well documented, which makes this difficult to determine. The CG portion of Nekbone is prevalent in its profile; *glsc3* (global scalar product) is a Fortran function called within the vector-matrix product in the CG iteration. Given the description of Nekbone in Section 2, its profile seems sensible. The Nek5000 profile is odd, with 51% of its execution time spent in *other*. We examined this profile in more detail and found that *other* comprises numerous short-running functions, with many of them being called around million times. Many of these functions also appear in Nekbone, but they are called on the order of a hundred thousand times rather than a million, so in aggregate, they do not account for a large percentage of the execution time and do not appear in the profile. We need to investigate these profiles further in future work to gain a better understanding of their mapping. This will be reported in subsequent assessment milestone reports.

The SWFFT and HACC dynamic profiles differ significantly. The *distribution_3_to_2* and *redistribution_2_to_3* are the main functions in SWFFT that are called during each step. These functions essentially redistribute the 3D global grid and the three 2D pencil distributions to create a DFFT object that coordinates the operations to actually execute the 3D distributed memory DFFT. In HACC, the *Step10_int* function is called millions of times by the node force calculation function. This calculation is the short-range force kernel in HACC. The *other* portion of HACC consists mainly of calls to two functions *distribution_2_to_3 distribution_3_to_2*, which are similar to those in SWFFT. Given the description of SWFFT in Section 2, its profile seems sensible.

The ExaMiniMD and LAMMPS profiles are somewhat similar. ExaMiniMD is implemented in Kokkos [5], and, therefore, the function names follow the Kokkos schema. The function accounting for the largest percentage of execution time in ExaMiniMD is *ParallelFor<ForceLJNeigh>* where similarly LAMMPS uses the *PairLJCut* function. These functions both seem to be doing computation on the the LJ force interaction. Due to the documentation limitation for ExaMiniMD, we could not relate the *ParallelFor<Neighbor2D>* function behavior to LAMMPS.

Figure 3 presents the dynamic function profiles for CoMD, miniFE, and XSBench. All of these profiles were collected from a serial execution. These profiles are as expected: CoMD spends most of its time doing the LJ force calculation; miniFE spends most of its time in the CG solve, and the XSBench function *set_grid_ptrs* consumes the majority of time doing binary searches through the nuclide grid. The case of XSBench shows the potential hazard of collecting performace data only at the whole application level. The time spent in *set_grid_ptrs* is an initialization expense and is not related to the cross section look ups the proxy is intended to represent.

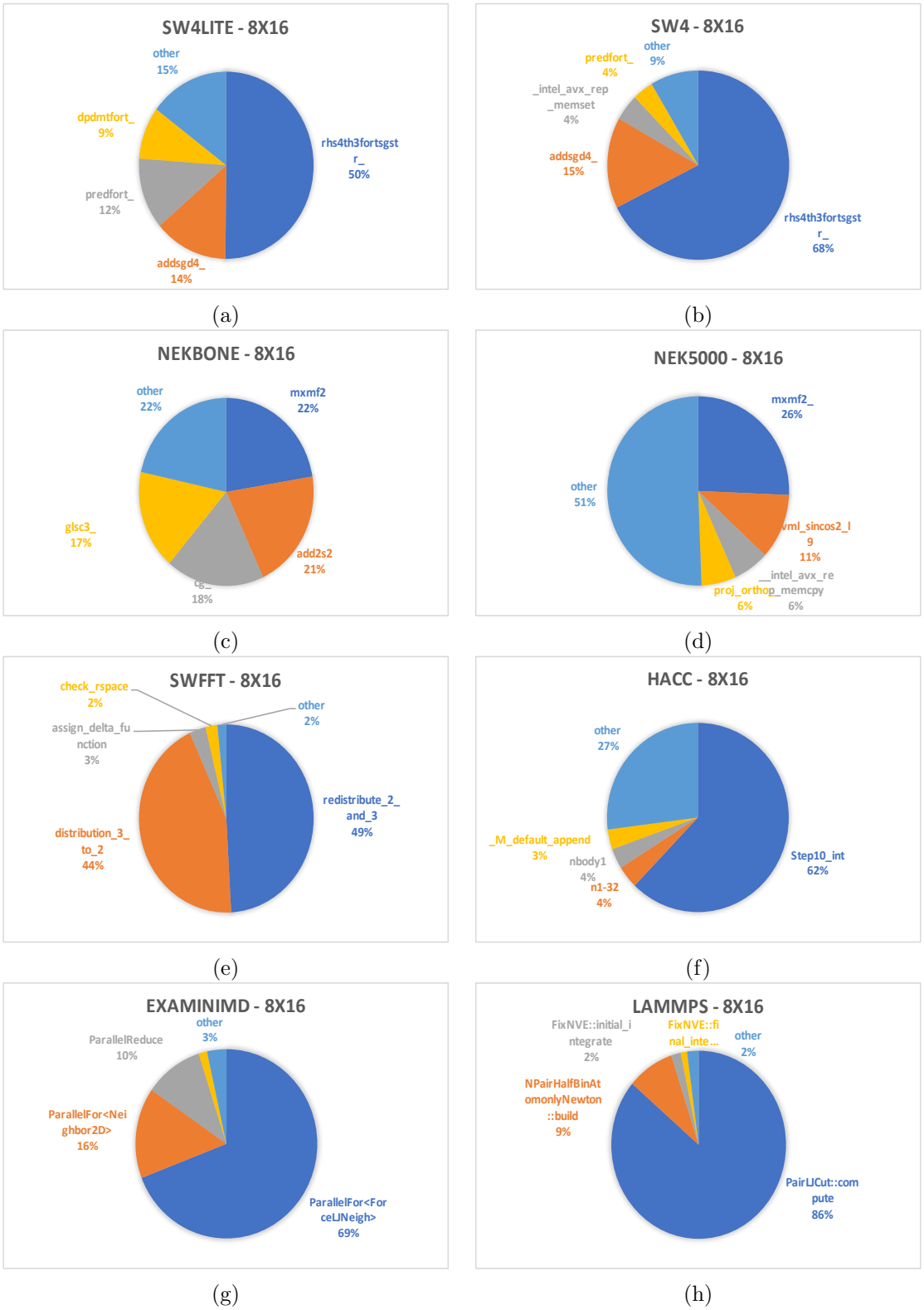


Figure 2: Dynamic Function Profiles Target Proxy/Parent Pairs

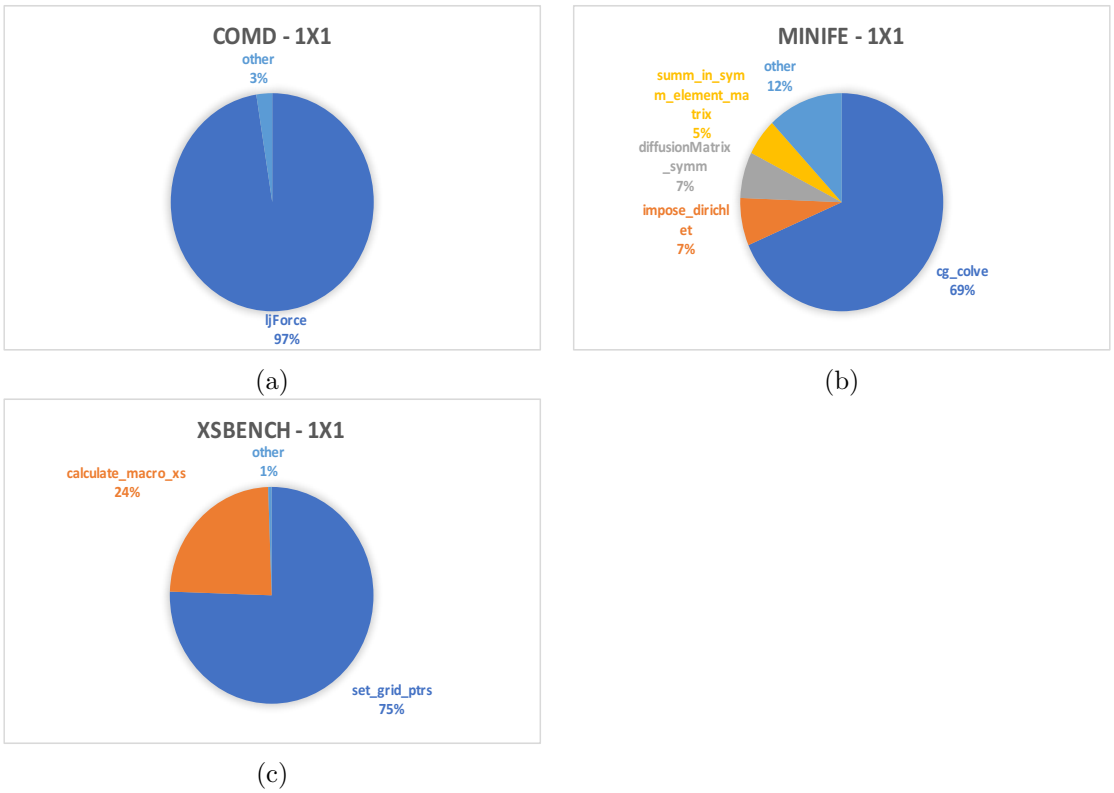


Figure 3: Dynamic Function Profiles Additional Proxies

3.4 Hardware Performance Counter Characterization

Here we present our initial efforts to gather hardware performance counter data on the four target proxy apps. The data presented was collected during serial or single-node execution, using HPCToolkit. A few issues should be pointed out about these studies:

- The Intel Haswell PMU has many known issues, several of which pertain to L2 cache and affect the computed miss rate, and are primarily due to how prefetches into the L2 are accounted for by the performance counters. Errata for the PMU events has been thoroughly studied in attempt to understand these issues. This issue manifests itself as potential inaccuracies in the total number of accesses to the L2. Therefore, we typically compute two metrics pertaining to miss percentages in the cache hierarchy: (1) miss rate, which is a global measure: miss rate=number of misses to cache level/total instructions retired, and (2) miss ratio, which is a local measure: miss ratio=number of misses to cache level/total number of accesses to that cache level. The second measure, miss ratio, is that which is affected by the PMU issue. We report both for completeness.
- The FP instruction counters, those which account for the various FP ops, are known to be faulty and/or missing altogether on the Haswell PMU, again as pointed out in Errata and discussed in several online working groups. There are only two FP counters, one that counts all AVX FP instructions and another that counts X87 FP instructions, but there are no counters for other FP operations. Hence, non-AVX and non-X87 FP operations are not counted at all. Because many codes do not take full advantage of AVX, these FP counts are typically relatively low compared to the actual number of FP operations executed by the code

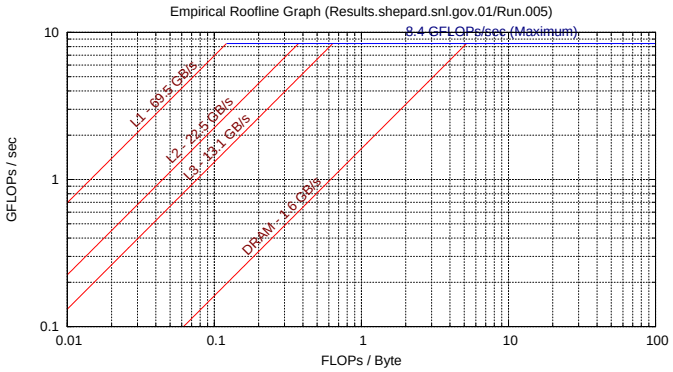


Figure 4: Roofline Model, Shepard (Haswell) Platform

- We compute arithmetic intensity (FLOPS/DRAM byte) and FLOPS/instruction here, but because of the issue counting FP instructions noted above, these may not be accurate. In our next assessment milestone, we are moving platforms to either Broadwell or Skylake, which will permit us to accurately obtain these counts.
- To compute the bandwidth utilized, we use the Roofline Toolkit and model, as noted in Section 3.2. An example of the output of this tool will be presented and discussed below.

The Roofline Toolkit runs several kernels on a given system and takes measurements of peak bandwidths at various cache/memory levels and FLOPS rates that are attainable on the machine. Figure 4 shows the peak bandwidths and FLOP rate for the Shepard (Haswell) testbed system at SNL. We use these peak performance projections from the roofline model to scale bandwidth utilization in the data subsequently presented. At this time, our only use of the Roofline Toolkit is to obtain a maximunm bandwith to use as the denominator when calculating fraction of bandwidth used.

Figures 5–8 show performance characterization (bandwidth, aritmetic intensity, cache miss rate, and cache miss ratio) for the seven proxy apps that we examine in detail for this work. This data was collected using the hardware performance counters through the HPCToolkit perf interface for the entire duration of the proxy run; the metrics shown are averages over the entire execution. On the y-axes of Figure 5, we show the miss rate (which is an accurate global measure) on the left and the IPC on the right. Note that the maximum IPC (instructions per cycle) for the Haswell architecture is 4, which is the retirement width. ExaMiniMD and Nekbone show relatively good performance, with high IPC and small miss rates. CoMD has small miss rates and a lower IPC, which probably indicates a hardware bottleneck at the execution units or retirement stage, which could be inherent to the algorithm. SW4lite performs very poorly. This will be investigated more deeply in future milestones to see if we can determine the underlying issue. The most noteworthy point to make from Figure 6 is the difference between miss ratio and the miss rate shown in Figure 5. Overall, miss ratios seem abnormally high, which is indicative of the Haswell L2 cache PMU issue. This will be revisited when we move to a platform with a more reliable PMU.

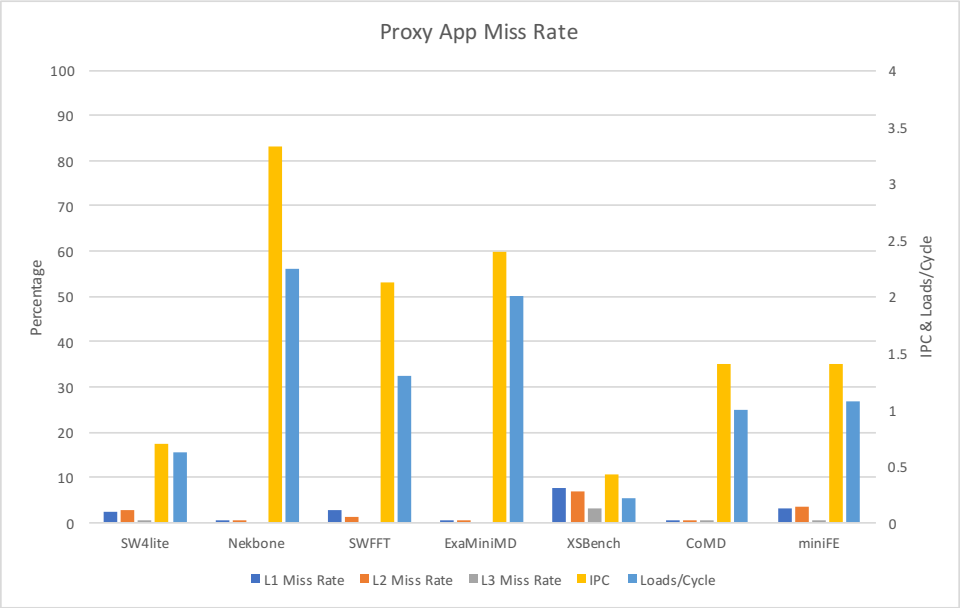


Figure 5: Proxy Cache Miss Rates

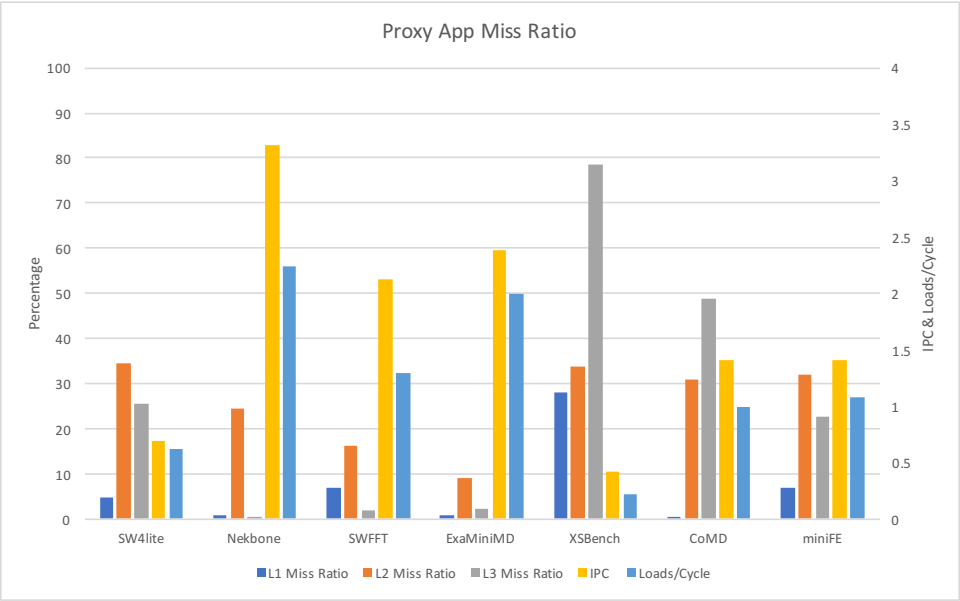


Figure 6: Proxy Cache Miss Ratio

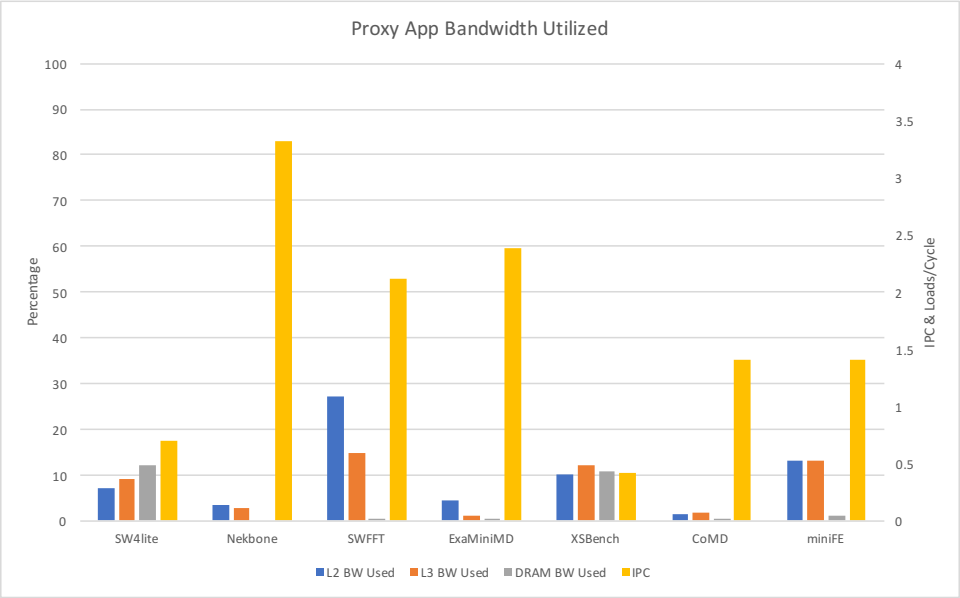


Figure 7: Proxy Bandwidth Utilization

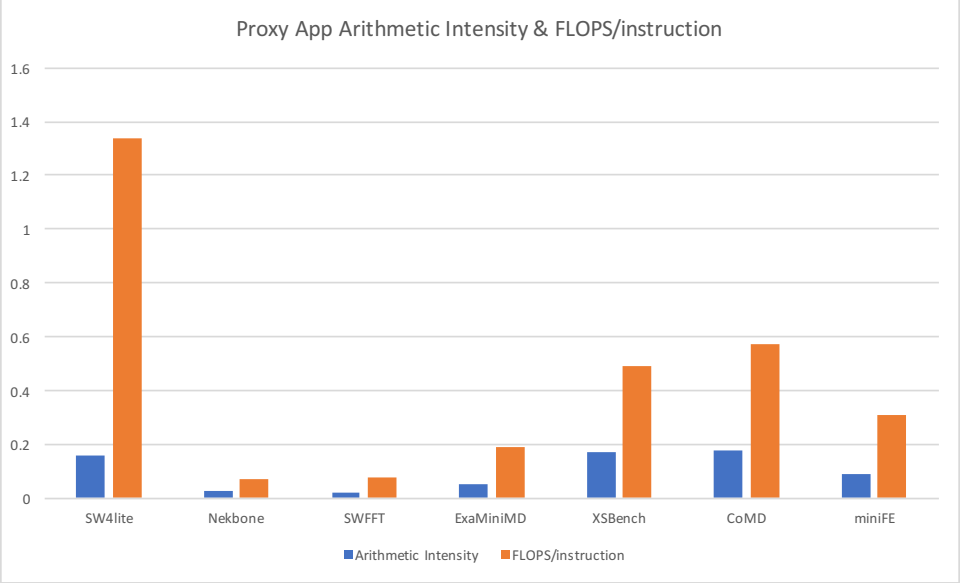


Figure 8: Proxy Arithmetic Intensity

Figure 7 shows the proxy app bandwidth utilization at various levels of the cache hierarchy. The y-axis on the left shows the bandwidth utilization as a percentage of the peak bandwidths that can be realized. Recall that we obtained peak bandwidths using the Roofline Toolkit, see Figure 4). As shown in the figure, these proxies use very little of the available bandwidth, with the maximum being around 28% L2 bandwidth utilized by SWFFT. All of the other bandwidth utilizations for all of the proxies are under 20%. Keep in mind that all of these metrics are computed as an average over the entire execution of the proxy. In the future, we will also present min/max for completeness, although even with that, it is not likely that bandwidth utilization is a hardware bottleneck at least for these proxies.

The arithmetic intensity data shown in Figure 8 may help explain the apparently poor performance of SW4lite. It clearly executes a large percentage of FP operations, and may experience a bottleneck in the the issue stage due to resource contention for FP units (depending on the distribution of these FP ops). We can look at this using hardware performance counters and will do so in subsequent milestone reports. All of the other proxies have very low FLOPS/instruction. Arithmetic intensity for all of these proxies is very low, which reflects their cache performance and bandwidth utilization shown in the figures.

Figures 17–31 in Appendix A show the cache and bandwidth performance metrics for some of the functions that account for the largest percentage of the execution time (according to the dynamic profiles in Figure 2). Our next assessment will probe function level metics more deeply.

4 Quantitative Performance Comparison of Proxies to Parent Applications

A proxy application is a smaller, less complex application that is designed to represent some key characteristic(s) of a larger, more complex parent application. Within the DOE, proxies are used by application groups to aid development and understand performance. They are also used as co-design vehicles in hardware development by vendor partners. The role of proxies in co-design of future computational systems drives the need to demonstrate that proxies are indeed representative of their larger parent applications in the way they were intended. In this work, we develop a methodology by which we can quantitatively compare hardware behavior and bottlenecks of proxy/parent pairs in order to understand the representatives of the proxies.

In this section, we present a methodology that we are developing for quantitative performance comparison of proxy/parent pairs based on an unsupervised machine learning technique, using performance counter data (communication, computation, memory) as input. We fisrt discuss the methodology, then present results and analysis.

4.1 Proxy to Parent Application Quantitative Comparison Methodology

One of the primary tasks of this milestone is to develop a methodology to quantitatively understand whether a proxy is truly representative of a parent application. There is some prior work that aims to quantitatively compare proxy to parent apps [2, 17, 15]. Some of this work is based on collecting dynamic data, primarily from hardware performance counters, but the comparison between proxy and parent apps is done qualitatively—i.e., the data is quantitative, but the comparisons between data sets for proxies and parents is only qualitative. The work in [15] is most closely related to our methodology, providing precedent to what we do here, but they focus more on communication.

Only a few types of quantitative data can be collected dynamically from an application: (1) timing information, such as the dynamic function profiles presented in Section 3.3, (2) hardware

performance counter data such as that collected from a CPU PMU, (3) communication (MPI) data collected from a tool such as mpiP, and (4) software performance counter data such as that collected from a tool like Byfl.

Using only dynamic function profiling data in a comparison methodology is not sufficient in that it does not contain any information about hardware performance bottlenecks. Function profiling data is very useful in understanding if the proxy and parent are executing the same functions (not necessarily named the same) and are spending similar percentages of the total time executing these functions. Because our goal is to understand if hardware bottlenecks in proxies and parents are the same or similar, we choose to use hardware performance counter data to understand hardware bottlenecks at the node level and mpiP data to understand communication behavior. The caviat here is that the hardware performance counter data is hardware (architecture) dependent and using this in the comparison methodology means that a proxy/parent may map closely on a certain architecture but map as completely distinct on another. For this milestone, we provide a proxy/parent mapping on only a single architecture. We are currently collecting data on additional architectures, and this data will be reported in future milestones. The metrics derived from software performance counter data are not architecture dependent. However, the tools that enable collection of this data can have very large overheads (up to about 90x) and often work only for serial executions, making them prohibitive to use practically in this type of performance comparison. We have collected some software counter related data and will look at how we can pull this into a comparison in future milestones.

Our technique is based on using hardware performance counter data derived metrics in conjunction with mpiP data metrics as input to a clustering algorithm that uses Manhattan distance to find similarities in the proxy/parent data is in terms of distance between clusters. Because clustering algorithms typically cannot handle data with large dimensionality, we use principal components analysis (PCA) as a pre-filter on the performance counter data to reduce the dimensionality of the data. We use the following hardware performance counter metrics as input to the PCA:

- Instructions per cycle (IPC)
- Micro-ops per cycle (UIPC)
- L1, L2, L3 miss ratio: uses the number of accesses to the particular cache level as the denominator; number of misses to a particular cache level as the numerator.
- L1, L2, L3 miss rate: uses the total number of load instructions as the denominator; number of misses to a particular cache level as the numerator.
- L1 to/from L2 bandwidth
- L2 to/from L3 bandwidth
- Instruction mix: Floating point, load, store, branch, and other (mostly integer) instructions. We compute each instruction category as a percentage of the total instructions committed. Note that due to Haswell PMU issues, the instruction mix data may be skewed because of problems with floating-point related event counters.

Since all of the proxies that we use are reported to be representative of computation, communication, and memory behavior of their respective parent application, we choose a set of hardware performance counter events that are indicative of computation (IPC, UIPC, instruction mix) and memory behavior (cache miss rates, ratios, and bandwidth utilizations). Communication behavior is reflected in mpiP data, which is described below. Note that we did not use arithmetic intensity, FLOPS/instruction, or DRAM bandwidth in this analysis. Arithmetic intensity requires the number of DRAM bytes transferred per FLOP instruction and can be collected either using dynamic binary instrumentation tools such as PIN [16] or Intel SDE [6], or HPCToolkit. PIN and Intel SDE

send_apptime_percent	isend_count / Total app time	sendrecv_AV_Byte / Total app time
send_AV_Byte / Total app time	irecv_apptime_percent	sendrecv_count / Total app time
send_count / Total app time	irecv_AV_Byte / Total app time	bcast_apptime_percent
recv_apptime_percent	irecv_count / Total app time	bcast_AV_Byte / Total app time
recv_AV_Byte / Total app time	allreduce_apptime_percent	bcast_count / Total app time
recv_count / Total app time	allreduce_AV_Byte / Total app time	wait_apptime_percent
isend_apptime_percent	allreduce_count / Total app time	waitall_apptime_percent
isend_AV_Byte / Total app time	sendrecv_apptime_percent	barrier_apptime_percent

Table 6: mpiP Collected Metrics

Send: combines send, isend	recv_mpi_percent
Recv: combines recv, irecv	recv_AV_Byte
Bcast: combines bcast , sendrecv, allreduce	recv_count
Wait: combines wait, waitall, barrier	bcast_apptime_percent
send_apptime_percent	bcast_mpi_percent
send_mpi_percent	bcast_AV_Byte_avg
send_AV_Byte_rate	bcast_count_avg
send_count_rate	wait_apptime_percent
recv_apptime_percent	wait_mpi_percent

Table 7: mpiP Combined Metrics

(which is based on a PIN tool) have prohibitively large overheads with respect to measurement on parent applications and we experienced issues with HPCToolkit successfully producing results on any long-running executions or codes more complex than a proxy. We will interface with the HPC-Toolkit developers in the future to work on this issue. We chose not to use the FLOPS/instruction metric because of the lack of FP-related event counters implemented in the Haswell PMU. On the experimental testbed used at SNL (Shepard), the paranoid bit setting is such that we cannot measure any uncore events, which prohibits accurate measurement of DRAM bandwidth. We are in the process of changing this issue on several of the SNL testbeds so this will not be a problem in the future.

We collect MPI communication data for all applications using the mpiP library. mpiP is a lightweight profiling library for MPI applications that collects statistical information about MPI functions and produces a flat file at the end of the application's execution. Using the mpiP file, we extract the Aggregate Time and Aggregate Sent Message Size from the instrumented application. The file also contains data for the 20 most used call sites. We aggregate all of the data for the same MPI routine across all call sites and then compute the metrics (rates) as shown in Table 6.

Since different applications use different MPI functions, many of the collected rates above output zero values. We also found that some parent and proxy applications do not use the same MPI routines. To correct for this disparity, we reduce the mpiP data by combining analogous behavior calls, for example, *send* and *isend* information will be combined into *send* only information. Table 7 shows the calls that we aggregated in attempt to impose some consistency in the mpiP data across all proxies and parents.

While aggregating the MPI data remedied the problem with consistency of mpiP data between proxy and parents, we found that this technique did not aid at all in understanding communication behavior of the applications. We attempted to use communication data in our machine learning model to understand similarity, but found that it actually skewed the clustering results. The mpiP data produced only one prinicipal component and this component made distinctions between the

App	Proc	Node	AVG Runtime (sec)	Number of runs
ExaMiniMD	128	8	971	5
LAMMPS	128	8	686	5
Nekbone	128	8	530	5
Nek5000	128	8	690	5
HACC	128	8	620	5
SWFFT	128	8	604	5
SW4lite	128	8	809	5
SW4_LOH1	128	8	780	5
SW4_LOH2	128	8	802	5

Table 8: Ranks, nodes, and runtimes for proxies and parents.

applications that were not present when using hardware performance counter data alone. Efforts to understand the best way to include communication metrics in our comparisons are on going. For exampls, we plan to extract communication patterns for each of the proxy/parent pairs in the future, then develop a method to quantitatively determine similarity within these patterns. Results of these efforts will be presented in future milestone reports.

4.2 Comparing Proxies to Parents with Unsupervised Clustering

In this section, we discuss results for our four target proxy and parent applications, and show how clustering successfully groups applications with similar behavior using hardware performance counter data defined by our methodology. For each application we collect performance and MPI data from five identical runs. As explained above, at this time we are not including MPI data in our analysis. Through LDMS, we collect data for each of the hardware counter events every second, for each process (rank) associated with the application. To aggregate this data for each of the five runs, we always select Rank 0, because it often does some extra work and varies more in terms of performance than the other ranks/processes, then we randomly select seven other ranks. We examined the data across each rank for each metric to verify that the variance was low. For each of these eight ranks, we average each of the performance counter events. This leaves us with a single set of data that is averaged over eight ranks for each of the five runs of each of the proxy/parent pairs. Therefore, five sets of data for each proxy/parent application is used as input to a principal components analysis (PCA). The PCA data is then subsequently input into our clustering model to produce final results.

Initially, we ran each proxy/parent application in several configurations with different numbers of nodes and MPI ranks. We attempted to treat each run as a unique data set for input to PCA and the clustering model. However, this created too much confusion in the analysis and we ultimately chose to focus on a single configuration. Therefore, all of the data input into the analysis was collected using 128 MPI ranks, distributed across 8 nodes, using 16 cores per node, with a single rank per core.

Table 8 shows the general configuration data describing the application runs that were evaluated. Note that for each proxy/parent application, we use the input configuration files that come with the distribution. We simply changed parameter values as necessary to match the problems and sizes as reflected in Table 2.

We use the R Statistical Computing Tool [7] to implement our unsupervised machine-learning-based clustering algorithm. Specifically, we use the hierarchical clustering method in R to group the application runs into K clusters, and use the *Elbow* method to select the best K value. The

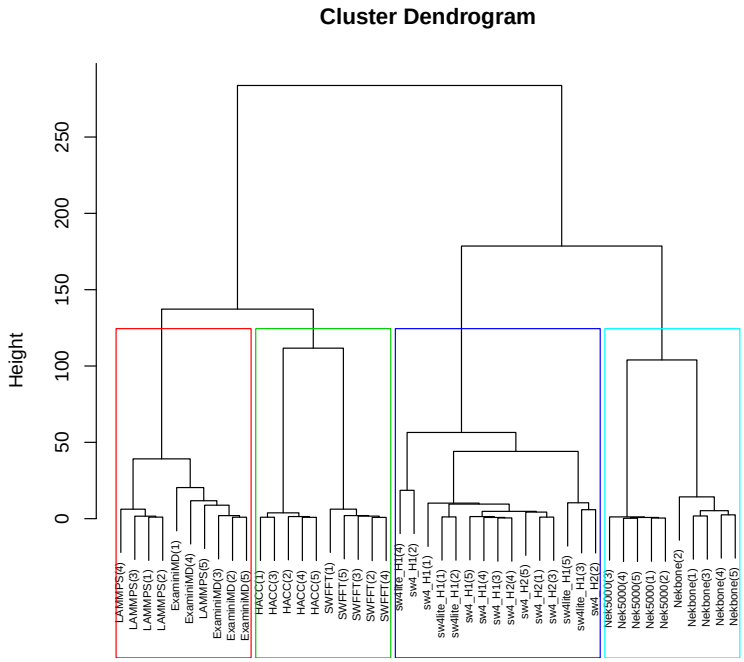


Figure 9: Cluster Dendrogram

resulting clusters contain applications with similar behavior based on the input metrics (computation, memory behavior). Figure 9 shows the partitions output by the clustering model. In the dendrogram, the y-axis indicates the height, which is a measure of similarity–the lower the height, the higher the similarity. The resulting dendrogram shows two primary, large clusters, one containing LAMMPS, ExaMiniMD, HACC, and SWFFT, the other comprising SW4, SW4lite, Nek5000, and Nekbone, meaning that each of these clusters exhibit performance that is similar. LAMMPS, ExaMiniMD, HACC, and SWFFT demonstrate more similarity amongst them than SW4, SW4lite, Nek5000, and Nekbone. Of all of the proxy/parent pairs, LAMMPS and ExaMiniMD are the most similar, followed by SW4 and SW4lite, then Nek5000 and Nekbone. Out of all the proxy/parent pairs, HACC and SWFFT, although similar, are the most different.

Figure 10, shows the contribution of each principal component with respect to explaining the hardware counter data from which they are derived. In our methodology we select the principle components that explain 90% of the data variance as input to the cluster; according to Figure 10 we select the first 6 PC's. PC1 explained 49%, PC2-4 explained 35%, and PC5-6 explained 9% of the variance. We also present in Figure 11 a heatmap view showing the importance of each of the input variables to the principal component analysis data reduction step, for each application. This view gives an indication of which HW counter metrics are important for which applications, though it does not give an indication of why they are important, since the PCA translation essentially precludes such an interpretation. We can see that most of the rates are important in PC1, and since PC1 explains the largest percentage of variance in our data set, this indicates that these rates are important factors in partitioning the application in the clustering algorithm.

The remaining principle components have different importance levels, where L2 miss rate and

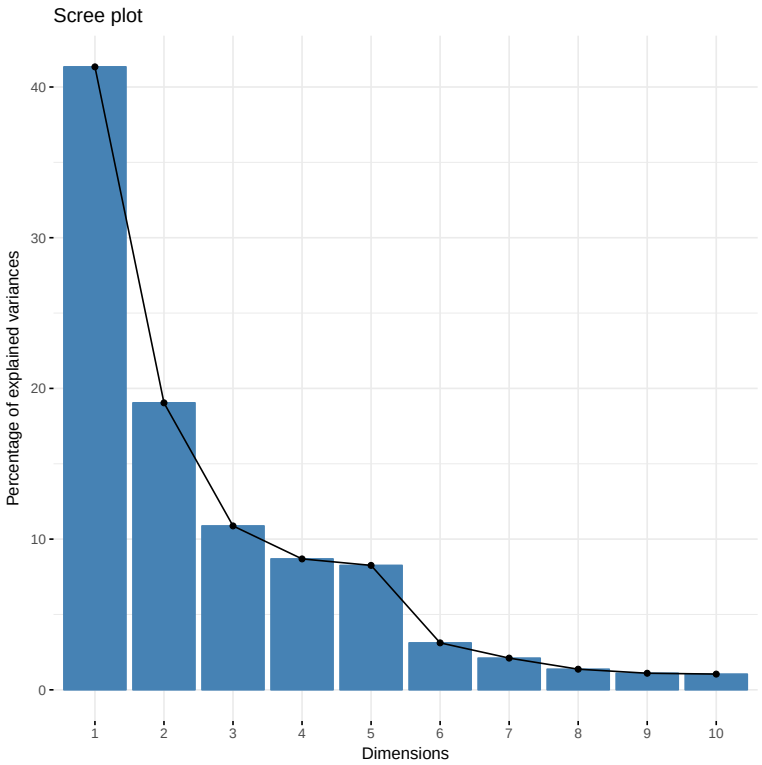


Figure 10: Principle components contribution

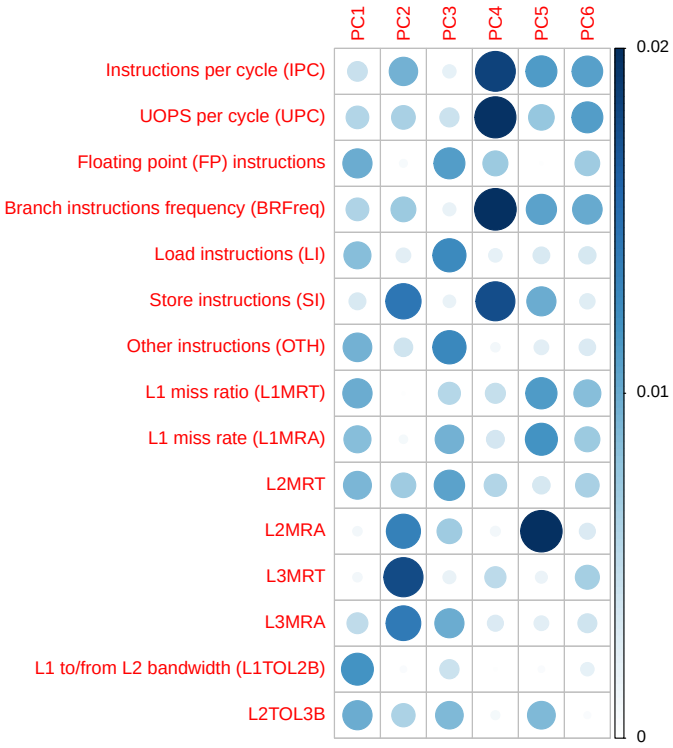


Figure 11: Rates importance per PC

L3 miss rate and ratio, in PC2, are the greatest contributors along with the store instruction percentage. PC3 has three main hardware counters that contribute to its importance. These rates are other instruction, floating point instruction, and load instruction rates. IPC, UPC, and branch frequency rates contributed the most to PC4, where L2 miss rate contributed the most to PC5. IPC and UPC also contributed the most to PC6. In summary, all of the metrics that we chose for the analysis affect the results of the clustering model.

4.3 Raw Data by Proxy/Parent Pair

To further validate the conclusions of the clustering model just described, figures 12–16 show the raw hardware counter data (BW, Miss Rate, Miss Ratio, and Instruction Mix) for each proxy and parent in ouy study. Evaluating this data one pair at a time, it becomes clear that a qualitative evaluation of the counter data produces consistent results with our clustering model. In other words, each proxy is better matched to its parent than any other in the study. We also observe that the high dergree of similarity within each pair is a good indication that the chosen proxies are good representations of their parents.

4.3.1 SWFFT/HACC

From Figures 12–16, we can see that the hardware behavior is very similar for both SWFFT and HACC, yet distinctly different than the other proxy/parent pairs. None of the proxy/parent pairs exhibit "good" IPC as shown in Figure 12—they roughly achieve 30–40% of max on average. It is somewhat interesting to note that HACC (and SW4) has a higher IPC than its proxy. This does make some sense in that if the proxy is primarily the kernel and does not include much of the set-up and overhead code, the IPC average should reflect that of the kernel, which we expect would be lower.

Compared to the other proxy/parent pairs in Figure 13, SWFFT and HACC use a small percentage of the available L1 to/from L2 and L2 to/from L3 bandwidth, with only ExaMiniMD and LAMMPS using less. This is relatively consistent with the global cache miss rates (i.e., cache misses/instruction) shown in Figure 15. Smaller miss rates should show less bandwidth utilization.

Notice in Figure 14 that there is essentially zero FP activity for both SWFFT and HACC. Other proxy/parent pairs exhibit this as well. In Haswell, this measures AVX instructions only, meaning that HACC and SWFFT do not vectorize well at all. Also note SWFFT and HACC are characterized by a fairly typical, but relatively large compared to the other applications, branch percentage. Scientific applications usually abide by a rule-of-thumb of about 10% branch instructions.

Compared to the other proxy/parent pairs in Figure 15, the behavior of SWFFT and HACC together is similar, yet distinct from the other application pairs with respect to L1 cache miss rate. SWFFT and HACC are characterized by L1 miss rates that are in the medium range relative to the other apps. Their L2 miss rates are more similar to other app pairs, but their L3 rates are larger than most of the other applications. L1 miss rate is very important in PC1, which is the PC that describes the largest percentage of variance in the data set, which is probably what largely contributed to SWFTT and HACC clustering together.

4.3.2 Sw4Lite/SW4

From Figures 12–16, we can see that the hardware behavior is again very similar for both SW4lite and SW4, yet distinctly different than the other proxy/parent pairs. For SW4, we collect execution data using both the LOH1 and LOH2 problems (see Section 2.3). However, for SW4lite, we only collect data using the LOH1 problem input.

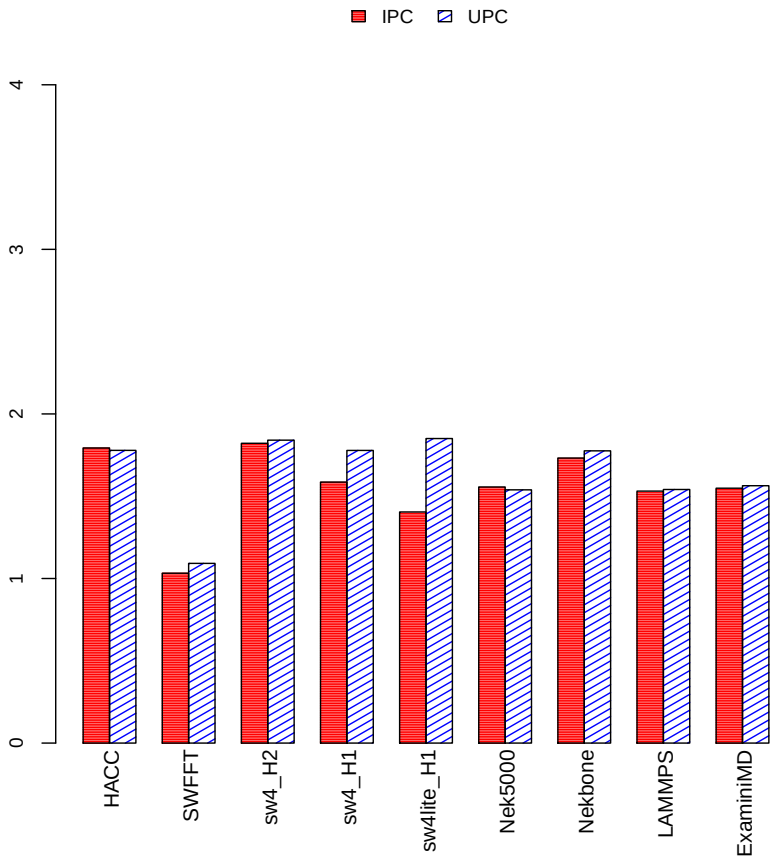


Figure 12: HW Counter Rates: Instructions per cycle (IPC)

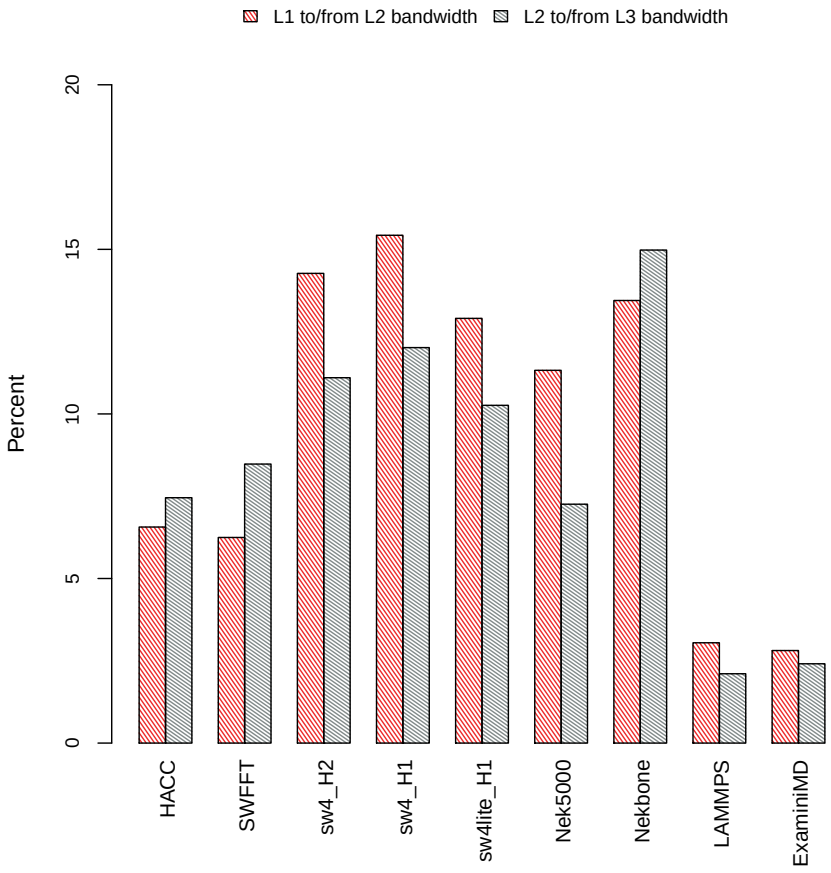


Figure 13: HW Counter Rates: Memory bandwidth

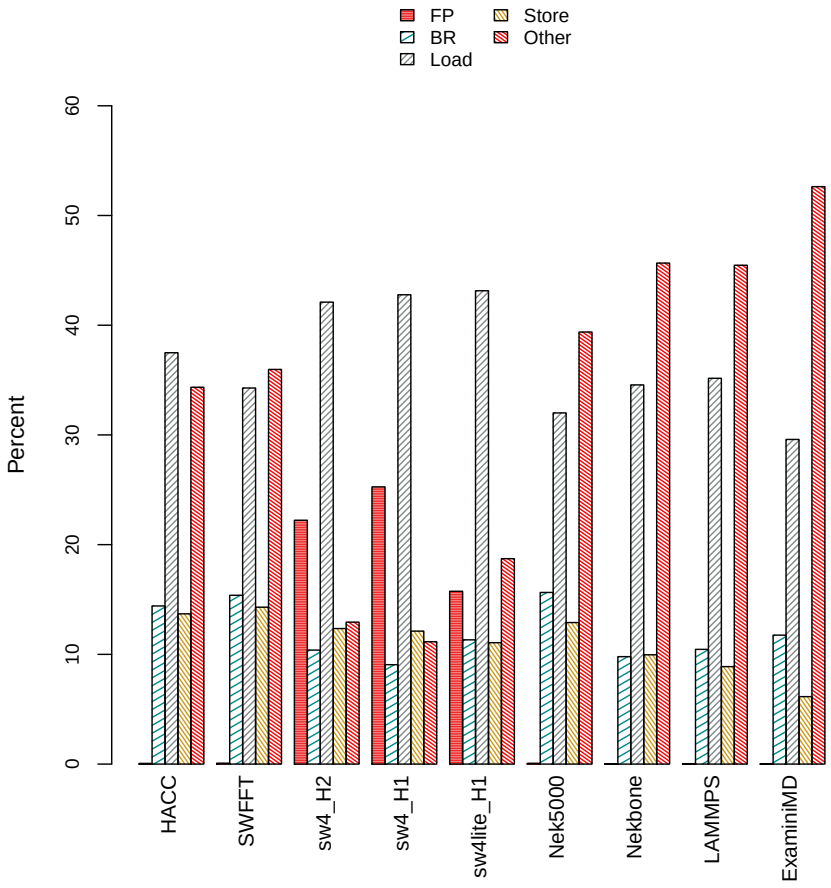


Figure 14: HW Counter Rates: Instruction Mix

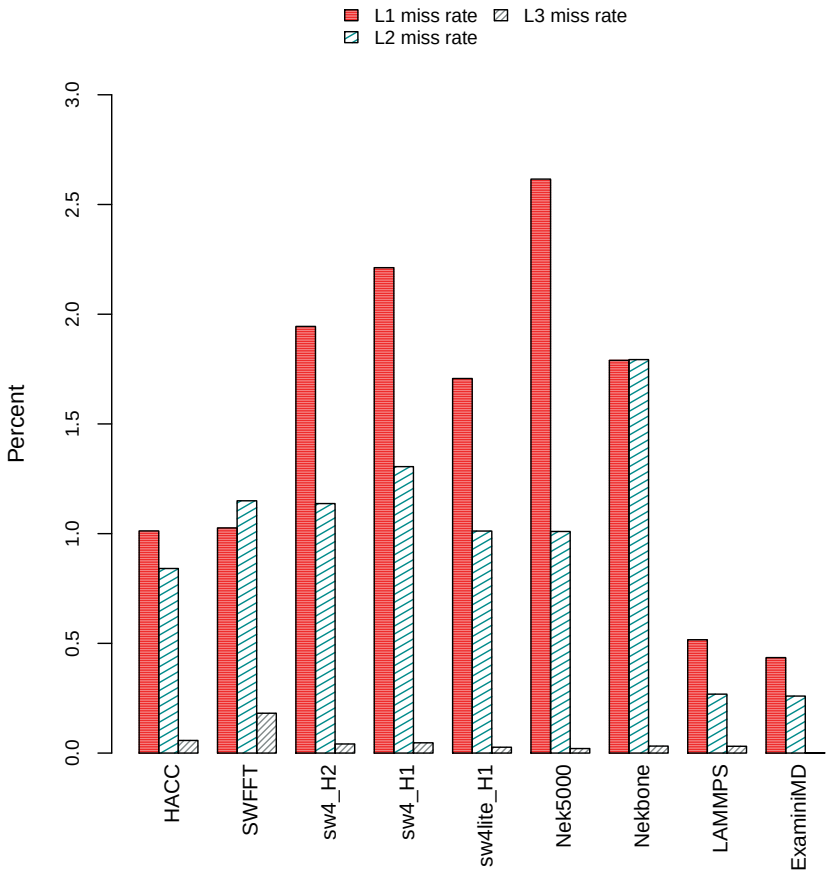


Figure 15: HW Counter Rates: Cache Miss Rates

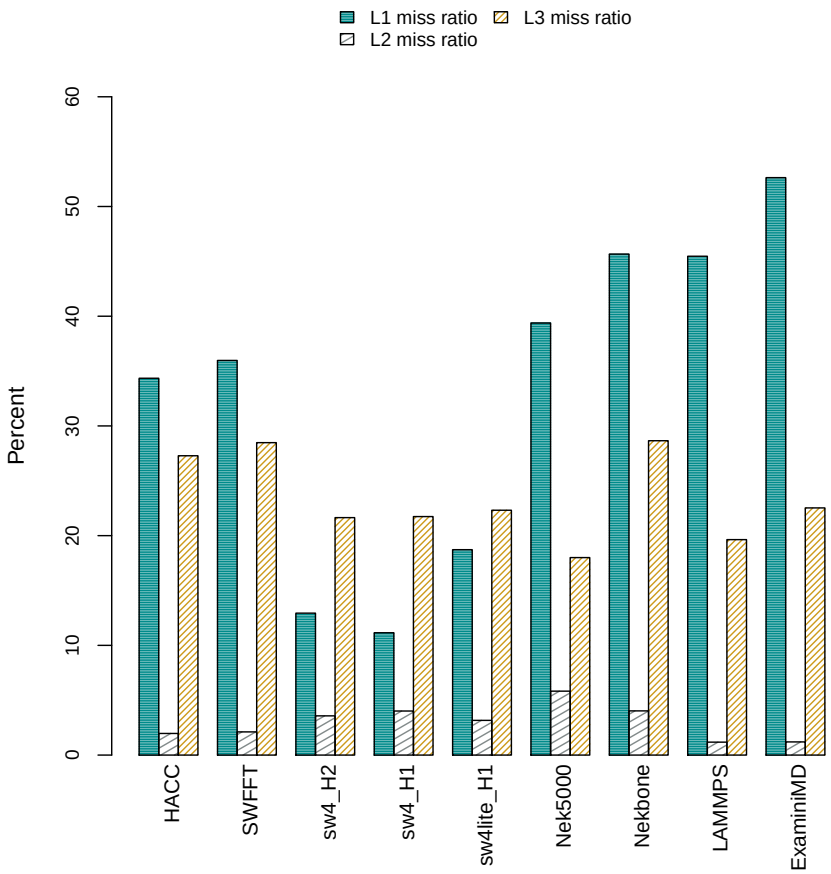


Figure 16: HW Counter Rates: Cache Miss Ratios

In Figure 12, we see that SW4, like HACC, has a higher IPC than its proxy, SW4lite. Similar to the HACC/SWFFT explanation, this could be due to SW4lite not containing all of the set-up and overhead code that SW4 comprises, which likely executes efficiently. If SW4lite is mostly kernel, its IPC should be lower. Figure 13 prevalently shows that SW4lite and SW4 utilize more of the L1 to/from bandwidth than any other proxy/parent pair; this is consistent with their high L1 cache and L2 cache miss rates shown in Figure 15.

Figure 14 indicates unique behavior in SW4lite and SW4 with respect to FP and load instruction percentages, which also affects the *other* category of instructions. This pair of apps is characterized by a relatively large percentage of AVX instructions, indicating fairly good vectorization behavior. Therefore, it seems odd that SW4lite and SW4 execute such a large percentage of load instructions. This warrants further investigation. It could be an anomaly in the way vectorized memory load events are counted by the PMU.

Finally, SW4lite and SW4 have distinctly large L1 miss rates (small L1 miss ratios) compared to the other proxy/parent pairs (Figures 15 and 16). And again, the high weight of L1 miss rate in PC1 probably contributes significantly to their clustering result.

4.3.3 Nekbone/Nek5000

In Figure 12, we see that Nek5000 has a lower IPC than Nekbone. This could be due to the dynamic profile that shows numerous functions that are called millions of times that are not called as frequently in Nekbone. Or this may be due to the poorer cache behavior of Nek5000. More investigation is required to completely understand this.

Nek5000's L1/L2 bandwidth utilization, as shown in Figure 13, is smaller than that of Nekbone. This does not make sense when looking at the global cache miss rate of Nek5000 in Figure 15, which for L1 is much higher than that of Nekbone. Again, all of the data for Nek5000 and Nekbone should more closely scrutinized to make more concrete conclusions.

Nek5000 and Nekbone have very similar instruction mixes (Figure 14), characterized by essentially no FP/AVX instructions. Nek5000 does more branching than any of the other applications, which makes sense given its problem coverage space and complexity. It probably has the most extensive code base of all the apps (LAMMPS may be close). Instruction mix lend significant weight to PC1, which may be what contributes the most to Nek5000 and Nekbone clustering together.

4.3.4 ExaMiniMD/LAMMPS

ExaMiniMD and LAMMPS have the most similar behavior of all of the proxy/parent application pairs. Their IPC (Figure 12) and bandwidth utilization (Figure 13) are distinct and practically identical. Their bandwidth utilization is extremely low, but correlates well with their cache behavior as seen in Figure 15—LAMMPS and ExaMiniMD have the best cache performance of all the applications. From this data, we would expect these applications to have a better IPC than they do—their IPC is only about 1.5 (with a max of 4). Much behavior can be masked by averages, which could be the case here. We do plan to examine all of the hardware performance counter data for the entire execution of each of the proxy/parent pairs in order to better understand the average behavior reported here.

LAMMPS and Nekbone appear to have very similar instruction mix as shown in Figure 14. LAMMPS and ExaMiniMD (and Nekbone) have a distinct larger percentage of *other* category instructions, even though they and other apps have very low percentages of FP/AVX instructions. A dynamic binary instrumentation tool could help explain this data. We plan to use Intel SDE in

the future to generate detailed instruction mixes from which we can gain a better understanding of the hardware performance counter data.

5 Summary and Conclusion

In this milestone, we perform an initial performance characterization and develop a methodology to quantitatively compare and reveal similarities in the performance of proxy/parent application pairs. We target four proxy/parent pairs, namely SWFFT/HACC, SW4lite/SW4, Nekbone/Nek5000, and ExaminiMD/LAMMPS. From this work, we conclude the following: *The four target proxy applications are indeed good representations of the computation and memory behavior of their respective parent applications. Because of problems with our methodology, we make no conclusions at this time in terms of representativeness of communication across these proxy/parent application pairs.*

Although we conclude from the data we collected that the proxies are representative with respect to their computation and memory behavior, we did not study hardware bottlenecks such as memory bandwidth or latency issues. These issues are difficult to detect in average metrics and we did not extract values specific to particular functions or application phase. This will be done in future work. We will also refine our methodology in the future to include more in-core metrics to help identify potential performance issues.

5.1 Lessons Learned

This was a very large effort and much was learned from the experience. We made some obvious mistakes that will be corrected in the future, but we also made some decisions that we now know were poor and will be revisited in the future. The following list encapsulates our major lessons learned:

- Make sure that we understand the application and proxy problem and how they map to each other. Also motivate the problem (e.g., why is this problem important?) to be used through interaction with the ECP application code teams. Be sure that we scale the problems the way they are intended to be scaled. Thoroughly document this and release it to the community.
- Turn spectral multigrid on for Nekbone and re-run experiments.
- Run both ExaminiMD and LAMMPS with the SNAP potential, collect data, and re-execute the model.
- Run SW4lite with the LOH2 input and report results.
- For all parent applications, use Intel SDE (PIN) to collect DRAM bytes so we can compute arithmetic intensity.
- For all proxy/parent pairs, use Intel SDE (PIN) to collect more detailed instruction mix information.
- Collect Byfl data for as many proxy/parent pairs as we can given the tool constraints. Pay particular attention to dependence distance data.
- Use a misses/load metric in the characterization and the quantitative comparison model.
- Generate Intel vectorization reports and cross-check these results with the hardware performance counter results.
- Migrate our experimental infrastructure to a Broadwell or Skylake architecture and repeat all experimentation. These PMUs are more reliable than that on Haswell.
- Extend our experimental infrastructure to include GPU performance measurement. A GPU-integrated platform should be included in the next assessment milestone.

- Instrument the proxy and parent functions to gain a better understanding of behavior and mapping and where any bottleneck behavior may be occurring. We have the capability to do this with existing samplers in LDMS.
- To ensure that we are consistent across the team with respect to proxy/parent problem/size/configuration, place the actual run configuration files in the repository for team access.
- Consider adding more in-core metrics to the characterization and quantitative comparison methodology to attempt to identify hardware bottlenecks.
- Generate communication patterns for each proxy/parent application pair from the mpiP data that we have collected (we may need to use an additional tool). Develop a method to compute quantitative similarity.
- Look into the issue observed in the SW4lite/SW4 comparison data where there is simultaneously a large percentage of AVX and load instructions. This may or may not be an issue, but should be investigated further.
- Add a cluster quality measure to the comparison methodology.

Our plan is to address each one of these items in our next round of assessment. For our next milestone, we will include the four proxy/parent pairs assessed here in addition to the eight new target pairs that will be assessed in the next milestone. Our goal is to have a very solid methodology and analysis of all of the ECP proxy/parent applications at the conclusion of the ECP Proxy Application Project.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-TR-750182

A Additional Performance Data

The graphs in this appendix show cache and bandwidth performance metrics for some of the functions that account for the largest percentage of the execution time in SW4lite, Nekbone, SWFFT, and ExaMiniMD (according to the dynamic profiles in Figure 2). Note that data for ExaMiniMD includes runs using both the Lennard-Jones and SNAP interactions.

A.1 SW4lite

Looking at SW4lite data in Figures 17–19, we see hardware performance counter metrics for the three functions that account for the largest percentage of execution time, and together account for 76% of the total execution time. In Figure 5 the overall IPC is slightly less than one, which is consistent with the function data, with the *addsgd4fort* function contributing largely to the overall IPC. Overall cache miss rates are also consistent with those for each of the three functions, with the *predfort* function having poor cache performance that probably largely contributes to the overall cache miss rates. Cache bandwidths in Figure 19 do not seem to correlate well with miss rates and ratios. For example, *addsgd4fort* uses the largest percentage of L2 and L3 BW, but does not have the highest L2/L3 miss rates/ratios. It would be useful to examine the data types used in these functions to better understand bandwidth utilization.

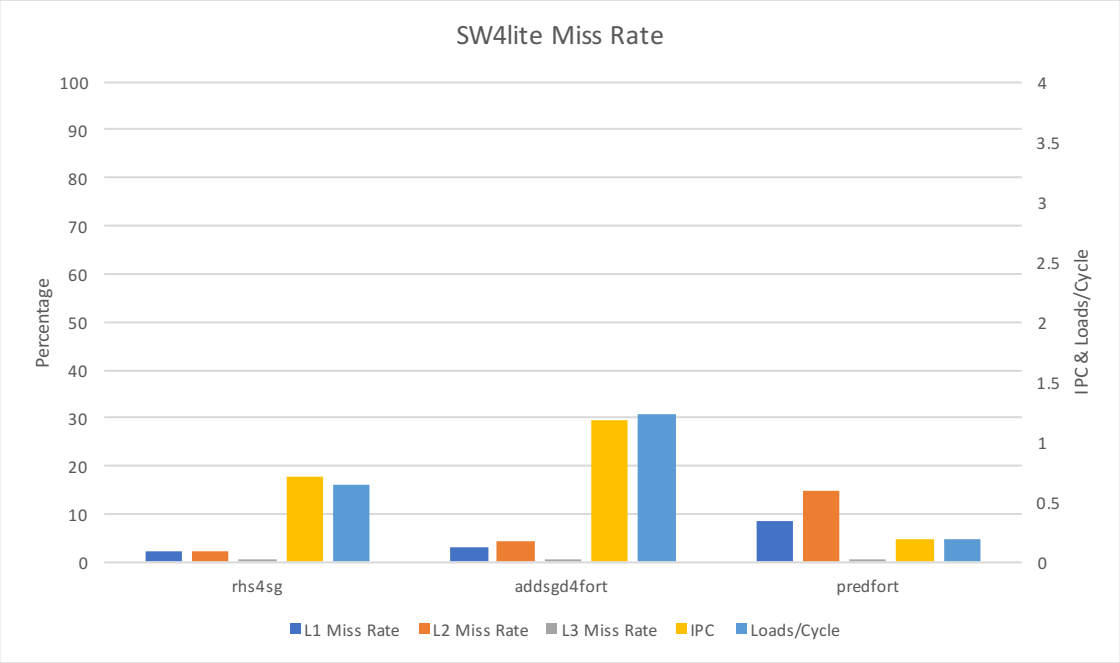


Figure 17: SW4lite Cache Miss Rate

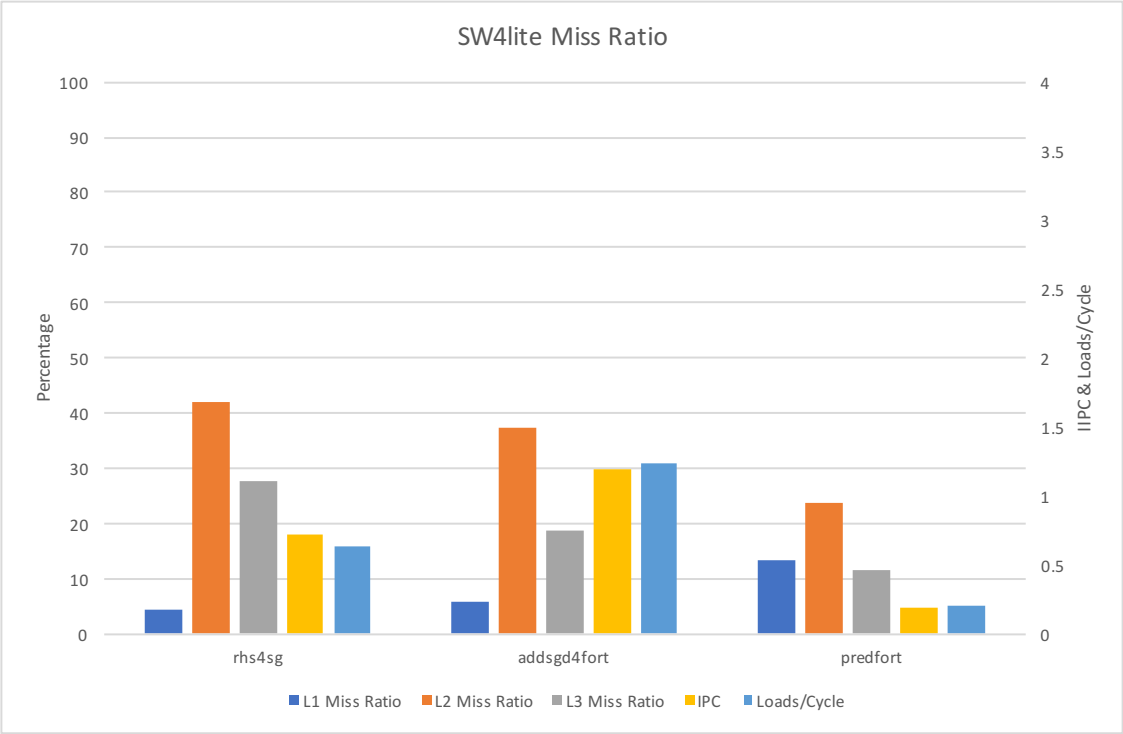


Figure 18: SW4lite Cache Miss Ratio

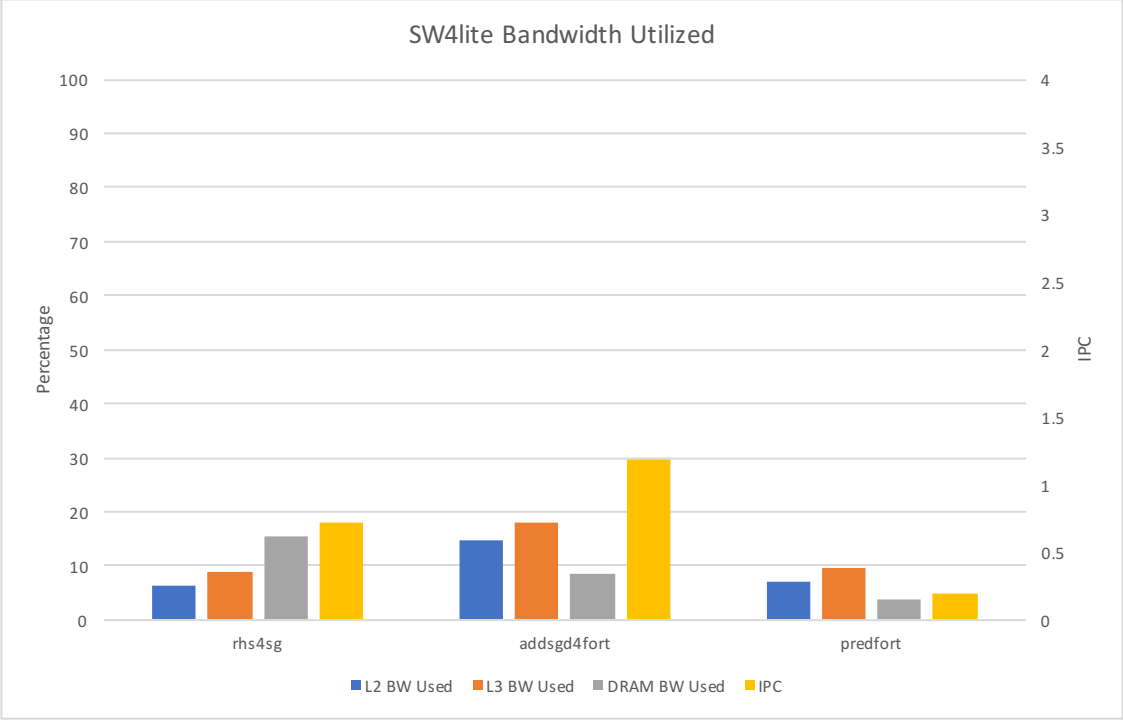


Figure 19: SW4lite Bandwidth Utilization

A.2 Nekbone

Nekbone has relatively large overall IPC, which is reflected in its function data, with all functions having fairly high IPC. *glsc3* largely contributes to Nekbone's overall L2 miss ratio and its L2 and L3 bandwidth utilization.

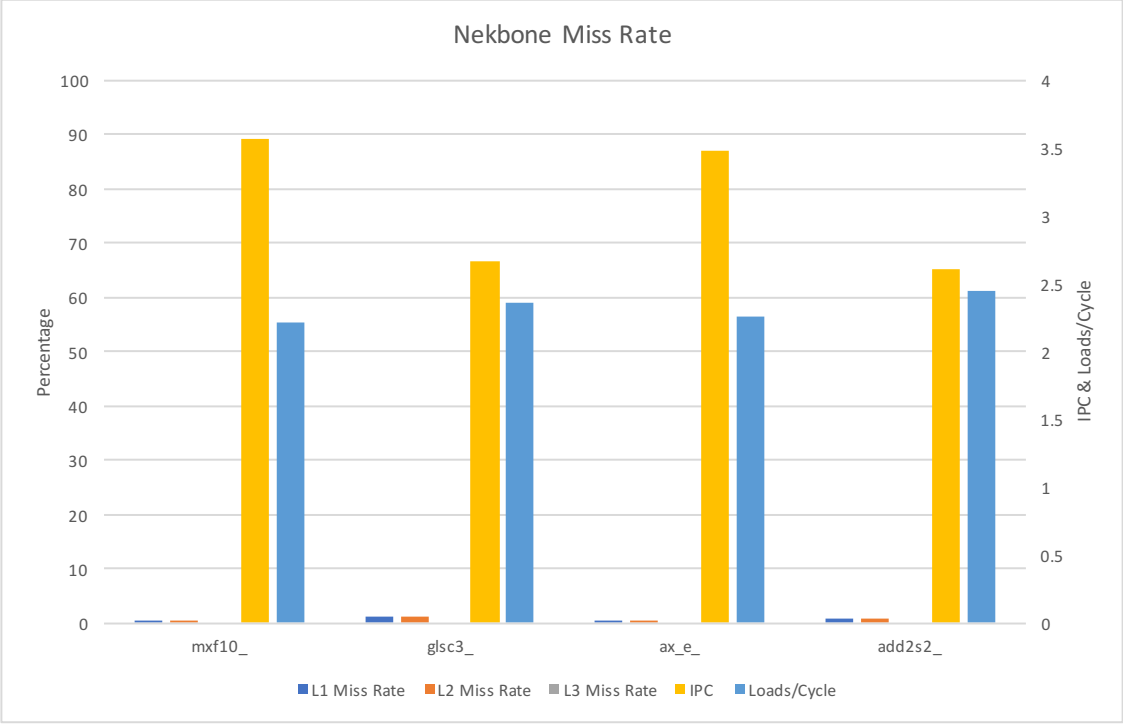


Figure 20: Nekbone Cache Miss Rate

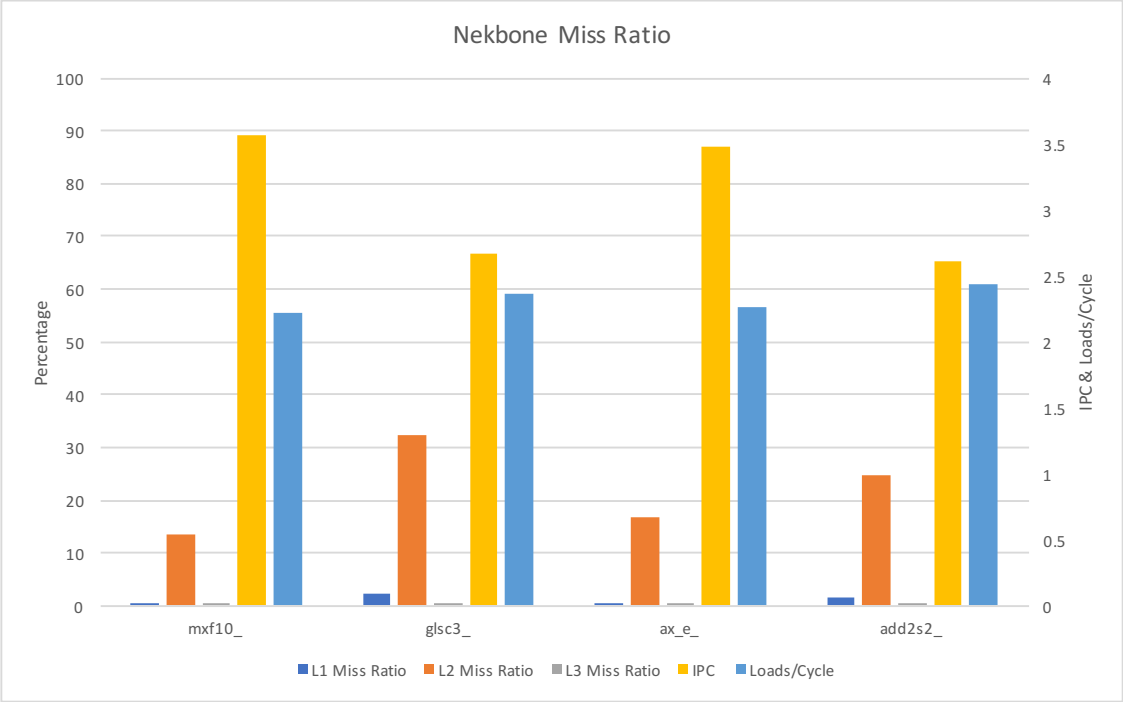


Figure 21: Nekbone Cache Miss Ratio

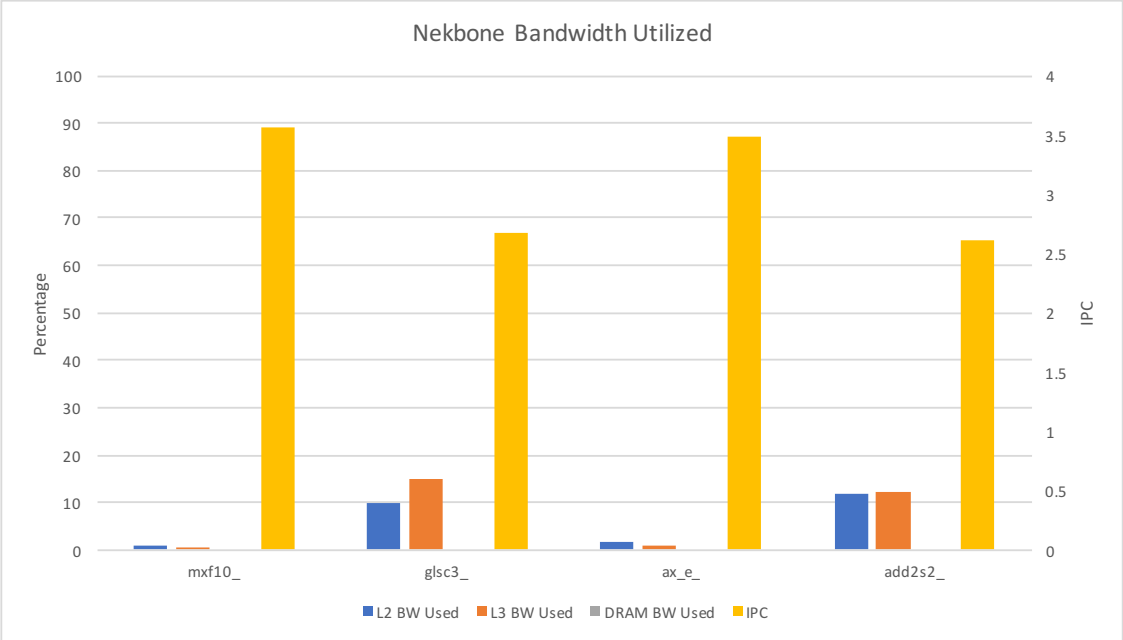


Figure 22: Nekbone Bandwidth Utilization

A.3 SWFFT

The *redistribute_2_to_3* function in SWFFT accounts for about 50% of the total execution time. It largely contributes to the overall SWFFT IPC and L2 and L3 bandwidth utilization. SWFFT has the highest bandwidth utilization of all the proxies, which can be attributed to the *redistribute_2_to_3* function.

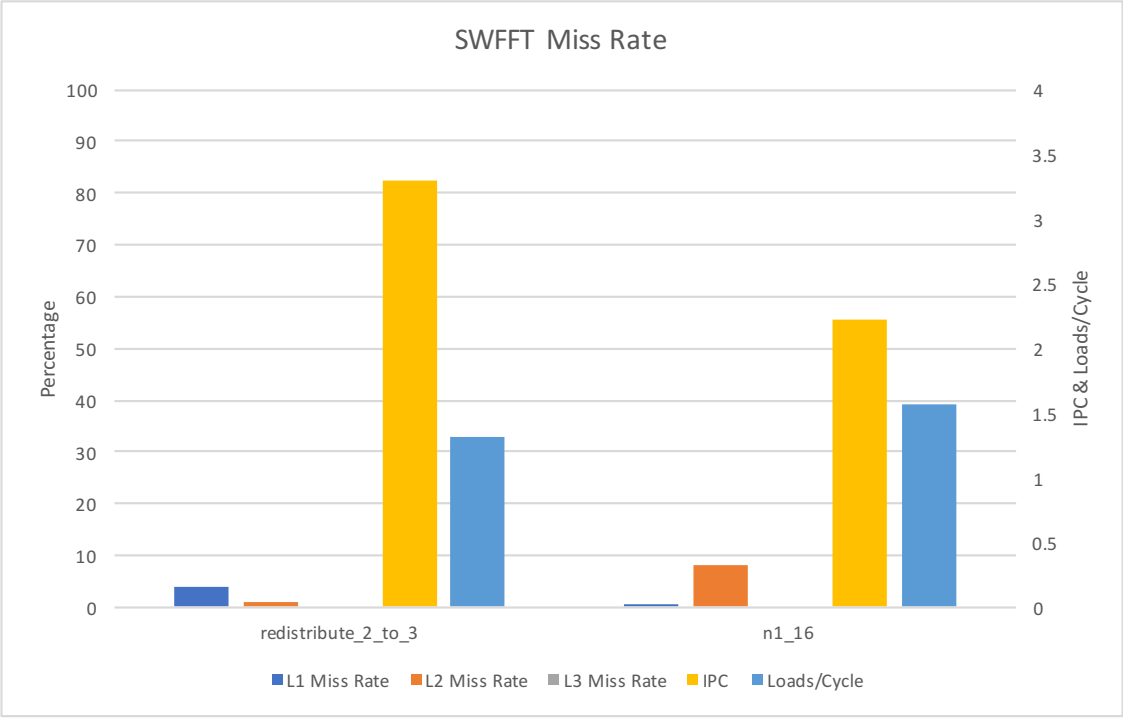


Figure 23: SWFFT Cache Miss Rate

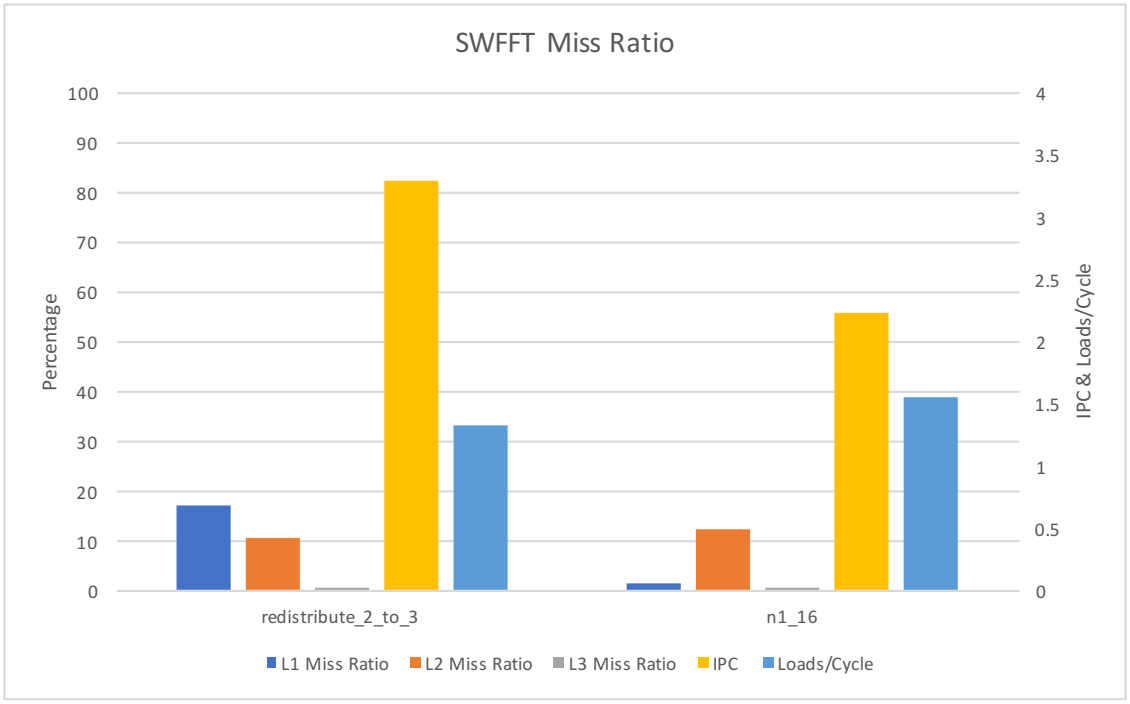


Figure 24: SWFFT Cache Miss Ratio

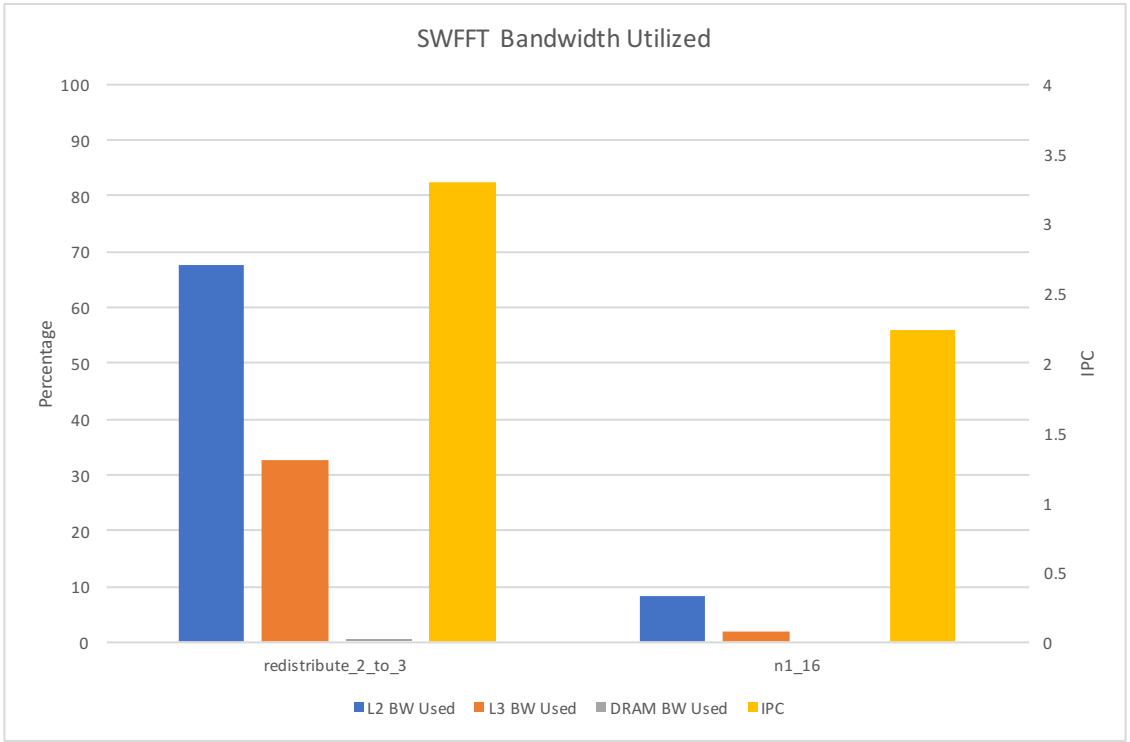


Figure 25: SWFFT Bandwidth Utilization

A.4 ExaMiniMD

The ExaminiMD function data in Figures 26–31 includes both the Lennard-Jones and SNAP interactions. What is interesting here is that the LJ potential has a much lower IPC than that for the SNAP interaction. This is counter-intuitive given the documented complexity of the SNAP interaction. The overall data in Section 3.4 for ExaminiMD shows the SNAP potential. Cache miss rates/ratios, IPC, and bandwidth utilization at the function level are all consistent with the overall data.

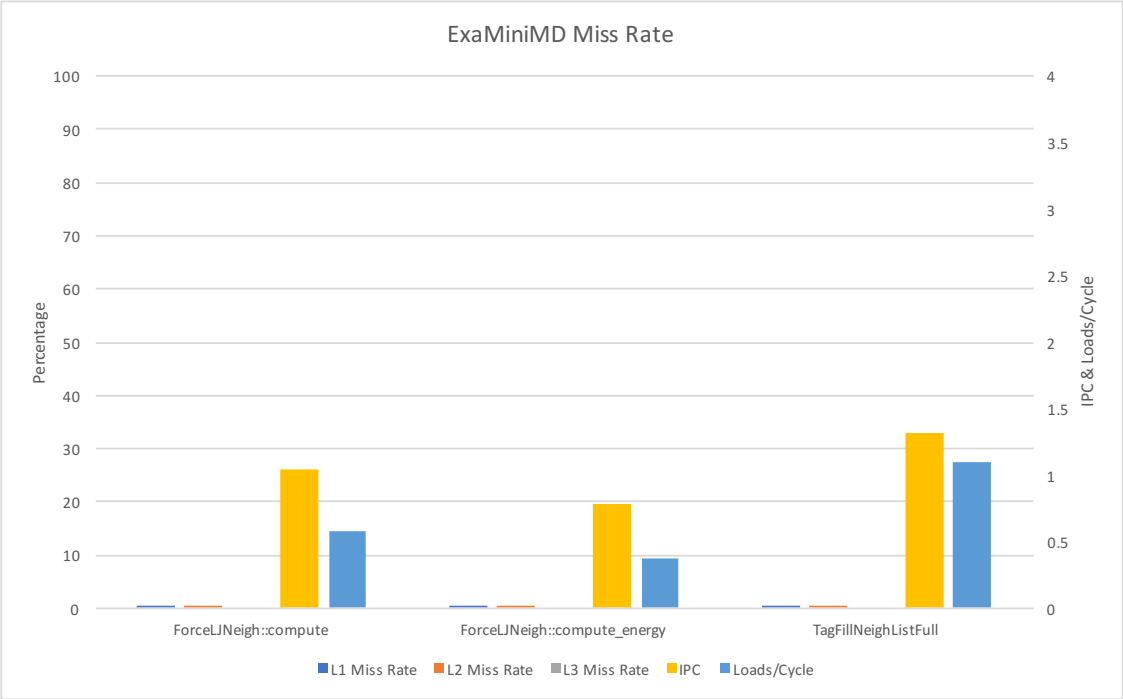


Figure 26: ExaminiMD (lj) Cache Miss Rate

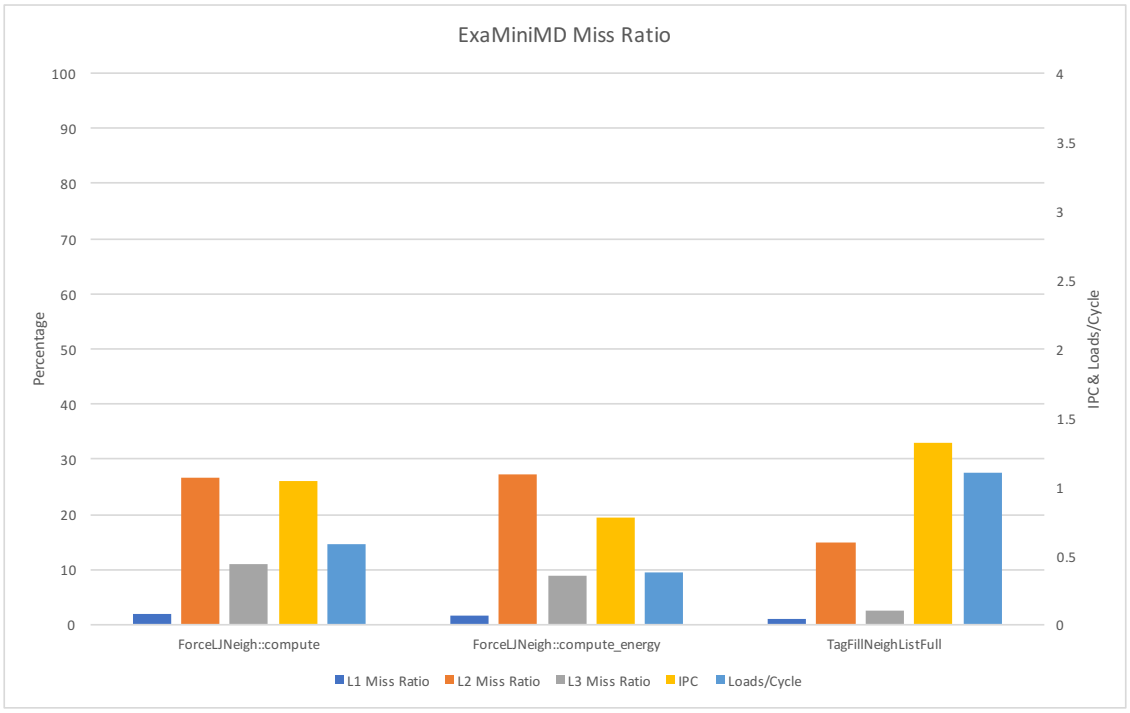


Figure 27: ExaminiMD (lj) Cache Miss Ratio

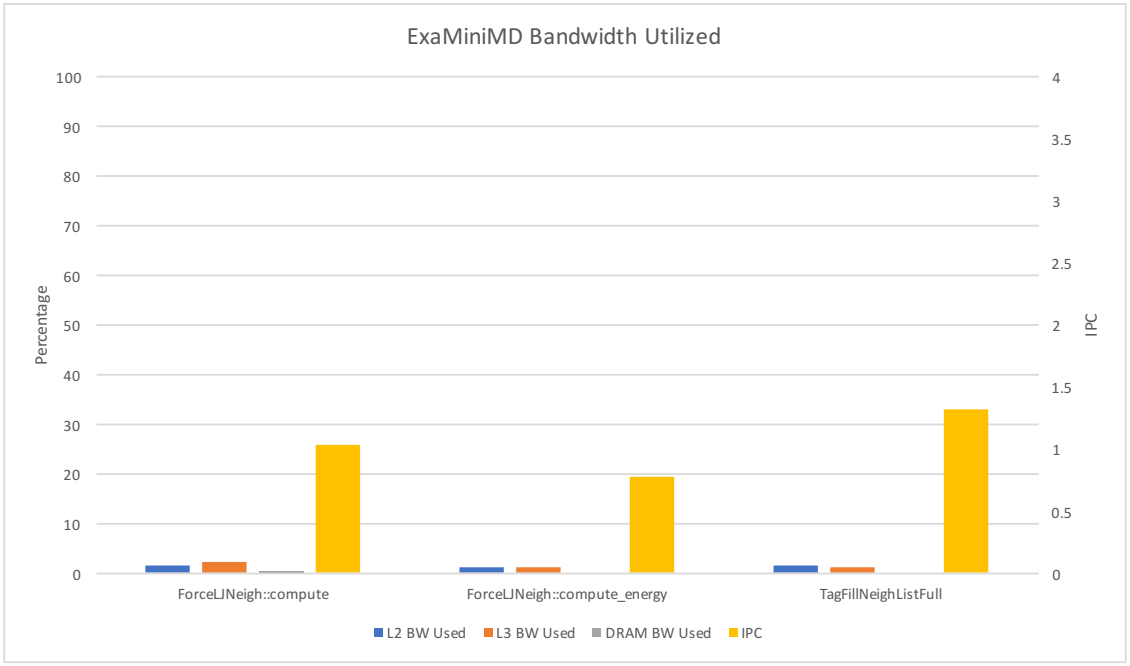


Figure 28: ExaminiMD (lj) Bandwidth Utilization

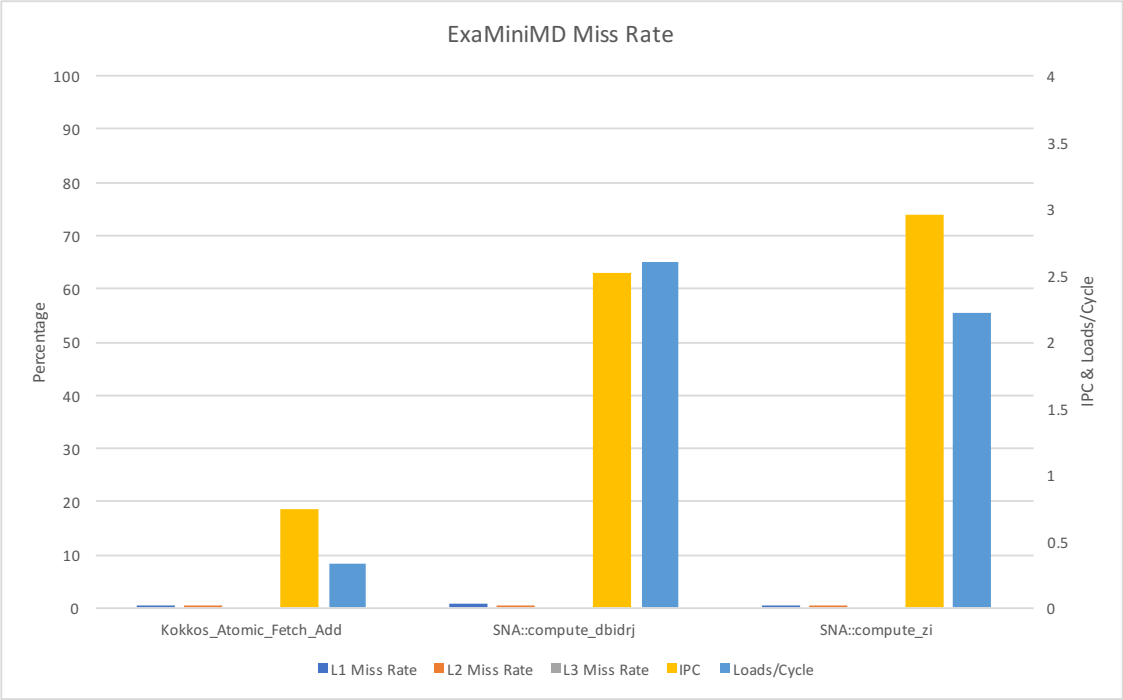


Figure 29: ExaminiMD (SNAP) Cache Miss Rate

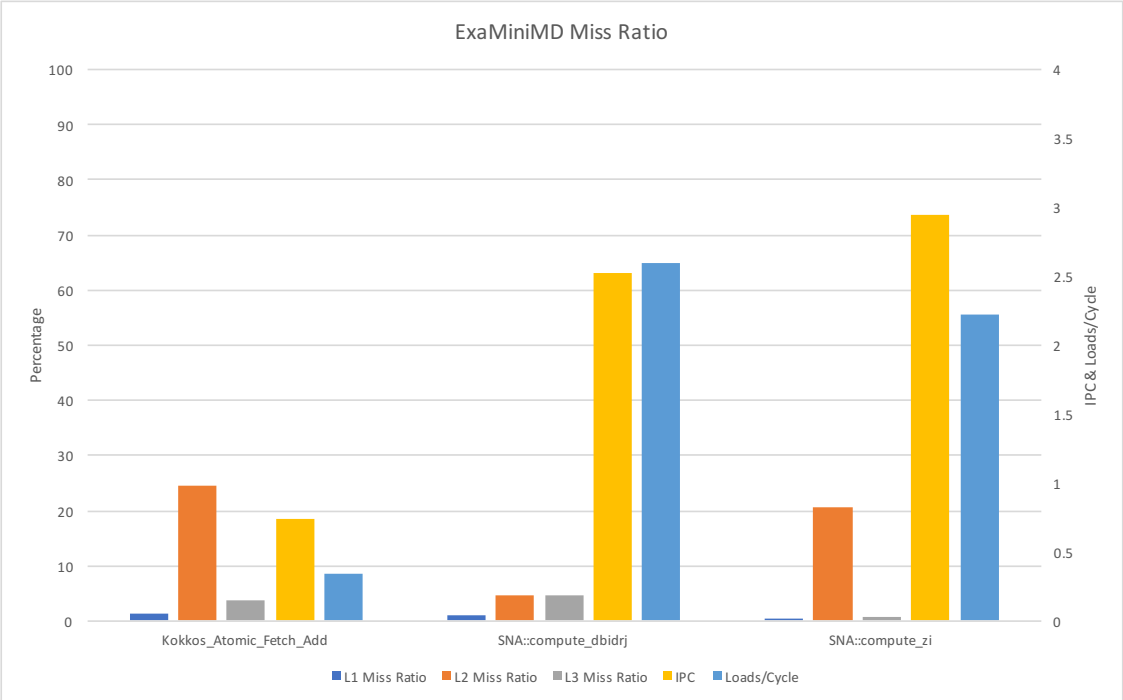


Figure 30: ExaminiMD (SNAP) Cache Miss Ratio

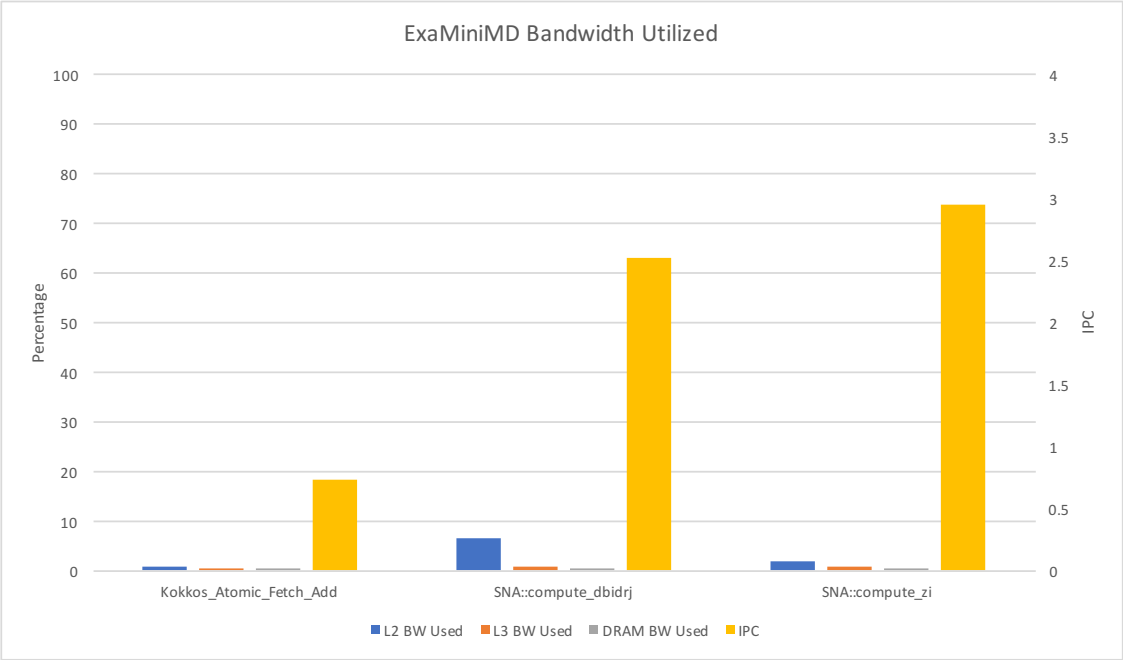


Figure 31: ExaminiMD (SNAP) Bandwidth Utilization

References

[1] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker. The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165, Nov 2014. doi:10.1109/SC.2014.18.

[2] G. Bauer, S. Gottlieb, and T. Hoefler. Performance modeling and comparative analysis of the milc lattice qcd application su3_rmd. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 652–659, May 2012. doi:10.1109/CCGrid.2012.123.

[3] Lawrence Berkeley Lab Computational Research Division. Roofline Performance Model. URL: https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/.

[4] Lawrence Berkeley Lab Computationl Research Division. CS Roofline Toolkit. URL: https://bitbucket.org/berkeleylab/cs-roofline-toolkit.

[5] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[6] Intel Software Development Emulator. https://software.intel.com/en-us/articles/intel-software-development-emulator. URL: https://software.intel.com/en-us/articles/intel-software-development-emulator.

[7] The R Project for Statistical Computing. https://www.r-project.org. URL: https://www.r-project.org.

[8] GNU Gprof. https://sourceware.org/binutils/docs/gprof/. URL: https://sourceware.org/binutils/docs/gprof/.

[9] HPCToolkit. http://hpctoolkit.org. URL: http://hpctoolkit.org.

[10] PAPI: Performance Application Programming Interface. http://icl.cs.utk.edu/papi/. URL: http://icl.cs.utk.edu/papi/.

[11] D. McCallen, A. Petersson, A. Rodgers, and M. Miah. High performance, multidisciplinary simulations for regional scale earthquake hazard and risk assessments. Technical report, Exascale Computing Project, Milestone ECP-ADSE19-EQSIM, 2019.

[12] E. Merzari, R. Rahaman, S. Patel, M.S. Min, D. Shaver, P. Fischer, and A. Siegel. Cfd smr assembly performance baselines with nek5000. Technical report, Exascale Computing Project, Milestone ECP-SE-08-47, 2017.

[13] Aleksandar Milenkovic. Perf Tool: Performance Analysis Tool for Linux, 2012. URL: http://lacasa.uah.edu/portal/Upload/tutorials/perf.tool/PerfTool.pdf.

[14] A.R. Siegel, K. Smith, P.K. Romano, B. Forget, and K.G. Felker. Multi-core performance studies of a monte carlo neutron transport code. *International Journal of High Performance Computing Applications*, 28(1), 2014.

[15] S. Sreepathi, M. Grodowitz, R. Lim, P. Taffet, P. Roth, J. Meredith, S. Lee, D. Li, and J. Vetter. Application characterization using oxbow toolkit and pads infrastructure. In *1st International Workshop on Hardware-Software Co-Design for High Performance Computing (Co-HPC)*, November 2014.

[16] PIN A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. URL: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[17] J.R. Tramm, A.R. Siegel, T. Islam, and M. Schulz. Xsbench: The development and verification of a performance abstraction for monte carlo reactor analysis. In *International Conference on Physics of Reactors (PHYSOR2014)*, volume 47, 2014.

[18] J.R. Tramm, A.R. Siegel, T. Islam, and M. Schulz. Xsbench–the development and verification of a performance abstraction for monte carlo reactor analysis. *Journal of Nuclear Science and Technology*, 52(7–8), 2015.