Proxy or Imposter? A Method and Case Study to Determine the Answer

Omar Aaziz*, Jeanine Cook*, Courtenay Vaughan*, David Richards†,
*Sandia National Laboratories, Albuquerque, NM USA
Email: (oaaziz|jeacook|ctvaugh)@sandia.gov
†Lawrence Livermore National Laboratory, Livermore, CA USA
Email: (richards12)@llnl.gov

***Abstract*—As the HPC community moves toward exascale, understanding application behavior is more important due to the increase in size and complexity of systems. While applications also grow larger in size and complexity, the need for proxy applications is crucial because of their ease of use and fast execution. They have become an essential aid for system vendors to evaluate new advanced architectures and for application developers to more quickly resolve algorithm and optimization issues. Therefore, proxies must be representative in behavior and function of the applications they mimic. In this work, we present a methodology to understand if a proxy represents a parent application based on a comparison of computational kernels, appropriate dynamic execution characteristics, and hardware bottlenecks. Based on this method, we conclude that miniQMC is a fairly good proxy for QMCPACK, but could be improved based on our analysis.**

***Index Terms*—Workload characterization; Proxy applications; Performance evaluation; Big data**

I. Introduction

Proxy applications provide a means to understand or refactor a larger parent application while using a smaller, more tractable and flexible code. They are typically developed either to be used for programming model and algorithmic exploration or as a co-design tool for architecture and system development as they are designed to represent some behavioral characteristic of the parent application such as computation, memory behavior, or communication. In both use-cases, the proxy must be a "good" representation of the parent. If it is not, algorithmic and other development done in the proxy will not result in the same behavior when implemented in the parent. Even worse in the co-design case, a system could be designed to resolve performance constraints that are non-existent in the parent, resulting in unexpected, sub-optimal system performance. Therefore, it is absolutely necessary to determine if a proxy faithfully represents its parent application in the context of what it was designed to represent.

Determining the representativeness of a proxy compared to its parent can be done in several ways. It is essential to qualitatively understand if the applications implement the same algorithmic solvers, conditioners, data structures, etc., while dynamic profiling can reveal which functions are primary kernels and if kernels and functions remain the same across the two applications. Tools such as roofline models [1] can reveal whether the proxy and parent are computationally and/or memory bound in a similar way. Comparison may go deeper and proceed into collecting hardware performance counter events and deriving metrics that can be compared between proxy and parent. These metrics may be further analyzed through statistical and other techniques (e.g., clustering, similarity measures) to determine how well the proxy matches its parent.

In prior work, we explore using statistical comparison techniques on hardware performance counter data to determine if a proxy is representative of its parent with respect to computation, memory, and communication behavior [2], [3]. Here we aim to use a combination of qualitative understanding of the proxy and parent in conjunction with dynamic profiling, roofline modeling, and qualitative comparison of quantitative metrics derived from various hardware performance counter data and analyses to understand representativeness. We extend our work beyond what is traditionally collected from hardware performance counters to include metrics that are implemented in Intel's Top-Down Microarchitecture Analysis (TMA) [4] that identifies specific hardware performance bottlenecks in the underlying processor microarchitecture. We use miniQMC and QMCPACK as our proxy and parent application, respectively, and present a characterization method to identify similarities and differences in these applications at both high (kernel) and low levels (microarchitecture) of abstraction.

II. Methodology

Here we present the method we use for this particular case study to determine if miniQMC is representative of its parent, QMCPACK. This method includes quantitative measurement and comparison at both the hardware and application/kernel level. All of the characterization uses CPU-only, MPI+OpenMP implementations of both proxy and parent. The outcome of this work is to determine if miniQMC represents QMCPACK with respect to important performance characteristics and understand the limitations to this representativeness, and to identify a general method that can be applied to other proxy/parent pairs. Our assessment methodology includes:

1) a clear qualitative comparison and description of the proxy versus the parent with respect to primary functionality,

978-1-7281-4734-5/19/$31.00 2019 IEEE

2) dynamic profiling to understand if key kernels and functions implementing these kernels are as expected and consistent across the two applications,
3) roofline modeling to understand at a high-level if the two applications are characterized by the same bottlenecks (e.g., compute-bound versus memory bound, cache bound, floating-point operations bound),
4) measurement and comparison of important metrics (e.g.,cache bandwidths, FLOPs, instruction mix) for the entire application execution and per-kernel,
5) identification and comparison of microarchitecture bottlenecks for the whole application execution.

The bottleneck analysis is based on the Intel Top-Down Microarchitecture Analysis. This is a hierarchical methodology that uses a pipeline slot abstraction to identify which components in the architecture (if any) are causing performance bottlenecks based on cycle counts. This hierarchy is shown in [4]. Each category in the hierarchy is associated with an area/component in the microarchitecture that could potentially demonstrate a hardware bottleneck. The methodology identifies bottlenecks using a predefined threshold based on an expected number of execution cycles for each area/component and suggested by Intel for HPC applications.

The method is hierarchical in that starting at the top, if a bottleneck is identified, the user can choose to drill-down to the next level of metrics to determine its cause. The top level of the hierarchy comprises: (1) frontend bound (instruction fetch and decode), (2) bad speculation (branching), (3) backend bound (memory hierarchy), (4) core bound (instruction issue/execution), and (5) retiring (instruction retire). In our analysis of miniQMC and QMCPACK, we primarily see issues in the backend and core.

III. Experimental Platform

We use QMCPACK v3.4 and miniQMC v. 0.4. We compile both the proxy and the parent with the Intel 18.1.163 compiler, using all default flags contained in the distribution makefile in addition to an architecture specific flag (-xCORE-AVX512) and a flag to force loop unrolling.

A. Computational Platforms

The Blake testbed at Sandia National Labs, NM, is an Intel Skylake Platinum 8160 with characteristics as shown in Table I. This architecture has 6 memory channels per socket, with a total of eight execution units including three vector units (two support 512-bit ops) that all do general vector ops and FMA. Skylake supports the AVX512 ISA, and has a new core-to-core, two-dimensional mesh memory fabric that potentially reduces latency and increases bandwidth between cores and memory. Key characteristics of the Intel Skylake Platinum 8160 used are shown in Table I. The characteristics with an asterisk are noted in Intel documentation as estimates.

Component	8160
L1 data cache	32 KB, 8 way, 64 B line size per core, private 403 GB/sec max BW $*$
L1 instruction cache	32 KB, 8 way, 64 B line size, per core, private
L2 cache	1 MB, 16 way, 64 B line size, per core, private 134 GB/sec sustained BW $*$
L3 cache	33 MB, 12–16 way, 64 B line size, shared, non-inclusive
Memory (per node)	192GB DDR4-2666 MHz
Cores/threads	24/48
Sockets/node	2
Total nodes	40
Interconnect	Intel Omnipath
Max Memory BW	119.21 GB/sec (19.87 GB/sec single channel)

TABLE I
HARDWARE CHARACTERISTICS OF SKYLAKE PLATFORMS

B. Profiling and Measurement Tools

For profiling, we use primarily HPCToolkit [5]. It is not capable of automatically generating an execution profile, but we manually extract this from the function execution information that the tool does generate. We use Intel VTune Amplifier [6] 2018.2.0 to generate cache-aware roofline models, and HPCToolkit and the hardware counter sampler within LDMS [7] (Light-Weight Distributed Monitoring System) to collect hardware performance counter data. Within the hardware counter sampler, we implement Intels Top-Down Microarchitecture Analysis (TMA) to identify hardware bottlenecks for whole application execution. We are working on implementing LDMS samplers that enable us to collect hardware counter and TMA data per function. We currently use HPCToolkit to obtain per-function hardware performance counter data.

IV. miniQMC and QMCPACK

MiniQMC is a quantum Monte Carlo code that comprises the important computational kernels of its parent, QMCPACK, and is intended to represent computational and memory, but not communication behavior of QMCPACK. Quantum Monte Carlo methods are often applied in material science to understand the electronic structure of molecular and solid state systems. The computational motifs of miniQMC and QMCPACK are particle methods, dense and sparse linear algebra, and Monte Carlo.

MiniQMC is meant to be a computational proxy, and therefore, implements no inter-rank communication. QMCPACK is fully MPI parallelized and since it is Monte Carlo, we expect a random communication pattern, which is what we observe. Because miniQMC and QMCPACK have no similarity in communication behavior by design, we do not include analysis of communication similarity in this work.

A. Algorithms and Key Kernels

MiniQMC, like QMCPACK, implements a direct solve of the Schrodinger wave equation, which provides accuracy at the expense of computational intensity. From this wave equation, the probability of particle position and particle energy are computed. MiniQMC and QMCPACK both implement Variational and Diffusion Monte Carlo (DMC). DMC samples the exact wave function, while Variational Monte Carlo (VMC) uses an approximate wave function. In this work, we focus on DMC.

A walker represents a 3D particle position, R. An ensemble of walkers is generationally and stochastically propagated through a defined electronic structure. Each propagation step moves the particle through the structure using a drift-diffusion process. The particle's local energy is computed at each step to determine if the particle dies, continues propagation, or reproduces. This changing particle population potentially creates imbalance that is addressed by periodic load balancing.

The four key kernels in both miniQMC and QMCPACK are the following:

1) Determinant update (inverse update): Uses the Sherman-Morrison algorithm to compute the Slater determinant. The Slater determinant provides an accurate approximation of the wave function being solved. This kernel relies on BLAS2 functions and is the source of the N^3 scaling in the application. This is clearly seen in the dynamic profile, presented in Section IV-C1.
2) Splines: Is invoked for every potential electron move. It computes the 3D spline value, the gradient (4x4x4xN stencil), and the Laplacian of electron orbitals. This kernel is memory bandwidth limited. Its large memory footprint makes data layout and memory hierarchy considerations critical to performance.
3) Jastrow factors (1, 2, and 3-body): The Jastrow factor represents the electronic correlation beyond the mean-field level in QMC simulations. Correlations are decomposed into 1, 2, and 3-body terms (electron-nucleus, electron-electron, and electron-electron-ion, respectively). This is a computationally intensive kernel.
4) Distance tables: These tables hold distances between electrons and electrons and atoms as matrices of all pairs of particle distances. Two tables are maintained—one for electron-electron pairs and one for electron-ion pairs. Minimum image and periodic boundary conditions are applied. Tables are updated after every successful MC electron move. Algorithms implementing this kernel have a strong sensitivity to data layout.

B. Problem Selection and Validation

MiniQMC comprises all of the key kernels that are in QMCPACK, although their relative importance in terms of percentage of total execution time is slightly different [8]. MiniQMC implements one walker per MPI rank and is meant for single-node explorations only. QMCPACK is fully MPI parallelized and is designed to take advantage of large-scale systems. Both miniQMC and QMCPACK support OpenMP threads, where the number of threads for a DMC calculation should be chosen to be only slightly larger than the number of walkers [9]. Therefore, to compare the proxy to the parent, we chose a single-node configuration, with one OpenMP thread per MPI rank; we chose the number of MPI ranks based on the available socket memory.

The exascale challenge problem for QMCPACK is to simulate transition metal oxide systems of approximately 1000 atoms to 10 meV statistical accuracy with performance portability. The transition oxide of choice is nickel oxide (NiO), and the target number of atoms is 1024. The 1024 atom problem is extremely memory intensive and cannot practically be executed on available contemporary systems without running out of memory. We chose a fairly contemporary system for our experimental platform and the largest problem that we can execute on this system (192GB/node memory) is 256 atoms (3072 electrons), which uses about 12GB per core. This system has 24 cores per socket, two sockets per node, but we use only 4 cores per socket (48GB) in order to force a reasonable run time and to ensure we execute within memory limits.

Kernel	miniQMC (meas/base)	QMCPACK (meas/base)
Determinant	73.9/83	71.6/82
Single-Particle Orbital (SPO)	11.5/5	11.0/6
Distance	12.8/5	12.2/6
Two Body Jastrow	1.4/2	4.5/2
Total %	99.6/95	99.3/96

TABLE II
COMPARISON OF KERNEL FUNCTION TIMING AGAINST BASELINE, MINIQMC AND QMCPACK

	miniQMC	QMCPACK
Execution time (secs)	1551.45	1612.32
Executed instructions	4.73E12	2.19E12

TABLE III
EXECUTION TIME AND INSTRUCTIONS EXECUTED.

To match the 256 atom NiO problem used for QMCPACK, for miniQMC we use the "-g 2 2 2" flag, which is 256 atoms and 3072 electrons [10]. We also use the "-r 0.999" to more accurately reproduce a DMC run. For comparison of miniQMC and QMCPACK, we use 8 ranks on a single node.

We do a high-level, order of magnitude validation to confirm that our QMCPACK and miniQMC runs perform as expected by reproducing a portion of the data from [8], which we call *baseline* data. The data was generated on a Skylake 8160, which is the same platform we use for our runs (although L3 cache and memory size may vary across the two systems). The baseline and our data is the single-thread case using 3072 atoms. Note, however, that we are using this baseline as a very rough guide. If we are within an order of magnitude of the baseline, we conclude that our application is running correctly. We know that the baseline data and our measured data likely use different versions of these codes and may largely differ in terms of kernel percentage execution times. This was the best data available at the time for obtaining some understanding that we were executing these applications correctly.

Table II shows results of our validation, which are the percentages of total execution time for each of the kernels in miniQMC and QMCPACK. Our results show about 10% less *Determinant* kernel execution time, and about 5-6% more time in *SPO* and *Distance*, and roughly the same amount of execution time in *Two Body Jastrow*. The baseline data shows 4-5% of the execution time outside of the four main kernels, while our data shows effectively all of the execution time is accounted for by those four kernels. Overall, the execution time percentages and the relative amount of execution time

spent in each function in our runs is within the same order of magnitude to the baseline timings. Note that the measured kernel times for both applications differ by a maximum of about 3%, with the *Determinant* and *Two Body Jastrow* kernels showing the largest percentage of execution time difference between miniQMC and QMCPACK.

Table III shows the execution time and the total number of instructions executed for each application. The execution time differs by about 4%. The number of instructions executed differs by about 74%, with miniQMC executing a significantly larger number of instructions. These are actual instructions retired, so does not account for instructions that are executed speculatively. Data presented in Section IV-C3 indicates a higher percentage of load instructions and a much lower percentage of vectorization for miniQMC. This could be why we see such a difference in the number of instructions executed. QMCPACK is doing fewer loads and fewer FP instructions, but they operate on larger data since they are vector instructions.

C. Results and Analysis

Here we present our data and analysis for comparing miniQMC and QMCPACK. We examine the results from each analysis, then make final observations with respect to similarity and representativeness in Section IV-E.

1) Dynamic Profiling: Table IV shows the HPCToolkit-generated dynamic profiles of miniQMC and QMCPACK, respectively, and also shows which functions implement each of the key kernels. The profiles appear to be fairly similar. In terms of percentages of execution time, the developers of miniQMC do state that miniQMC implements the key kernels of QMCPACK, but the relative percentages of execution time may vary, and looking at the table, this is absolutely true. Other observations include:

- The *Determinant* kernel comprises the same functions, except that QMCPACK has an additional function, *evaluateLog*.
- *SPO* has no functions in common between miniQMC and QMCPACK.
- The *Distance* kernel has two of three functions in common, with the third function only appearing in the profile of miniQMC (namely, *ParticleSet::setActive*).
- In the *Two Body Jastrow* kernel, there is some similarity, but there are more functions in QMCPACK.

Adding up the contributions of each of the kernels for both miniQMC and QMCPACK as shown in Table II, we see that execution time contributions of the key kernels are fairly similar with the exception of *Two Body Jastrow*, which shows the largest difference in total execution time. To summarize, the kernel execution times are similar, but the functions comprising the kernels are different as seen in Table IV. This indicates that we should examine per-kernel data rather than per-function data for lower-level similarity.

2) Roofline Model: We use a cache-aware roofline model generated by the Intel Advisor tool to get a general understanding of the behavior of both the proxy and parent and to determine at a high level if that behavior is similar. Figure 1 shows the roofline models of miniQMC and QMCPACK for a single core and single thread execution. The tool generates the values on the roofline curves by running a suite of benchmarks and measuring sustained cache/memory bandwidths and FLOPs rates on the particular system. From the data that we do have on sustained bandwidths (Table I), these measurements look reasonable on these plots. From the executing application, the tool measures cache/memory bandwidths and FLOPs rates for each function or loop and places these with respect to the roofline as a single dot. The larger the dot means that that function or loop accounts for a larger percentage of the total execution time, indicating greater opportunity for impacting performance if optimized. Dot size can be custom color-coded; we chose red-yellow-green for largest down to smallest percentage of execution time loops or functions. To determine how each function or function loop (dot) is bound, draw a straight line up from the particular function and the first line it hits determines what it's bound by.

In Figure 1 we only show the functions or loops that benefit most from optimization as determined by the roofline tool (i.e., only red (largest) and yellow dots, no green dots (smallest) are shown). These correlate with kernel functions that account for larger percentages of the execution time as shown in Table IV. In both roofline plots, the MKL BLAS loops appear in the same respective place. The leftmost red dots are bound by scalar add GFLOPs; the rightmost red dots are bound by L3 bandwidth and scalar add GFLOPs. Assuming that these BLAS operations are matrix-matrix, they should be compute bound. MKL is a top function in terms of total execution time and it is implemented in the *Determinant* kernel, which is known to be computationally bound. Looking at the functions represented by the yellow dots, we see that QMCPACK's roofline has two additional functions on the plot compared to miniQMC's roofline. One of these is associated with the *Distance* kernel and the other with the *Two Body Jastrow* kernel (i.e., *evaluateLogandStore*). These may appear in QMCPACK's roofline but not in miniQMC's because of the difference in percent execution time – these account for a smaller amount of time in miniQMC. Of the two yellow-dot functions that the plots have in common, the spline function that is part of the SPO kernel is computationally bound as expected by DP (double-precision) Vector FMA (floating-point multiply-add) GFLOPs in both cases. The spline function in the solve loop is bound differently in the two applications. In QMCPACK, it is bound by L3 bandwidth and in miniQMC, this function is bound by DP vector add GFLOPs. This is a discrepancy that warrants further investigation.

In summary, the rooflines have similar functions that are mostly bound by the same hardware components and QMCPACK's extra functions on the roofline are understandable. The function boundedness seems to mostly match the expectations of how the four key kernels are bound.

3) Using Hardware Performance Counters to Understand Behavior Similarity: We begin this analysis by examining several key metrics in our LDMS hardware performance counter sampler that help in understanding differences and similarities

Kernel	miniQMC	% Time	QMCPACK	%Time
Determinant	DiracDeterminant::acceptMove	57.8	DiracDeterminantBase::acceptMove	49.3
	DiracDeterminant::ratioGrad	6.2	DiracDeterminantBase::ratioGrad	7.7
	MKL	5.2	DiracDeterminantBase::ratio	2.3
	DiracDeterminant::ratio	4.7	MKL	10.0
			DiracDeterminantBase::evaluateLog	2.3
Single-Particle Orbital (SPO)	einspline_spo::MultiBspline::evaluate_vgh	9.3	SPOSetBuilderFactory::createSPOSet	11.0
	einspline_spo::MultiBspline::evaluate_v	1.2		
	einspline_spo::MultiBspline::set	1.0		
Distance	ParticleSet::makeMoveAndCheck	4.8	ParticleSet::makeMoveOnSphere	11.2
	ParticleSet::setActive	4.8	ParticleSet::makeMoveAndCheck	1.0
	DistanceTableAA::makeMoveOnSphere	3.2		
Two Body Jastrow	TwoBodyJastrowOrbital::BsplineFunctor::acceptMove	1.4	TwoBodyJastrowOrbital::BsplineFunctor::ratio	4.0
			OneBodyJastrowOrbital::BsplineFunctor::ratioGrad	0.5

TABLE IV
KERNEL FUNCTION PROFILES.

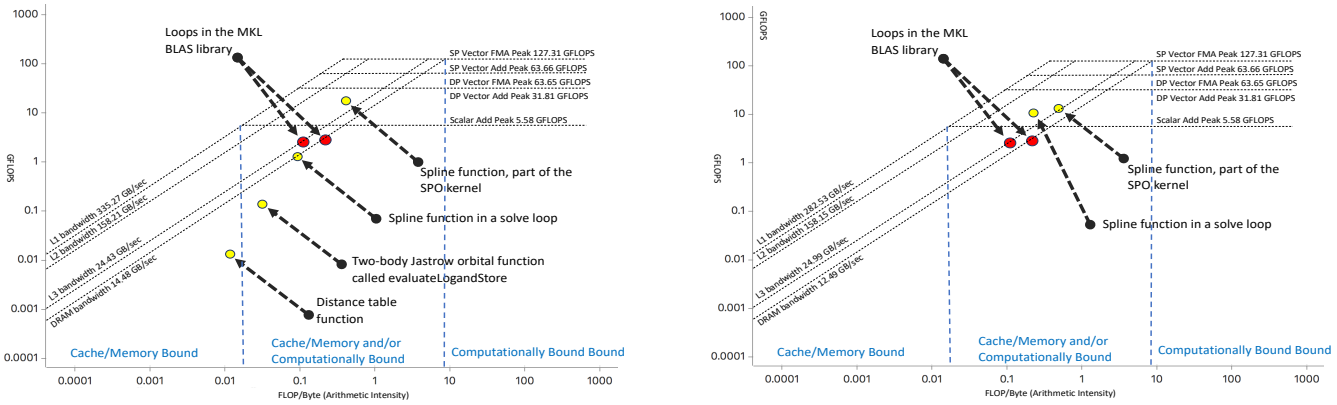


Fig. 1. QMCPACK roofline (left) and miniQMC roofline (right)

in proxy/parent performance. These metric definitions are taken largely from the Likwid [11] performance tool, with some metrics based on our own analysis. We include here only the metrics that show distinct behavior between miniQMC and QMCPACK. All metrics are an average per-core and are derived from events measured during the whole execution of each application. Each process executes on a single core and the deviation in results between processes is within 5% or less.

We start by looking at whole-execution throughput shown on the left in Figure 2. MiniQMC has smaller CPI (cycles per instruction) and CPU (cycles per micro-op), meaning a higher throughput than QMCPACK. Because their dynamic profiles are not very similar on a per-function basis, this is not surprising. The per-kernel throughput is shown in the figure on the right. The largest difference between miniQMC and QMCPACK is in the *Jastrow* kernel, which exhibits no function similarity (Table IV). The next largest difference is seen in the *SPO* kernel, again, where no function similarity is seen in the profile. Ideal CPI for this architecture is 0.17 (i.e., issues up to 6 instructions per cycle), so both applications are significantly higher than ideal (lower is better), indicating some stall behavior somewhere in the pipeline.

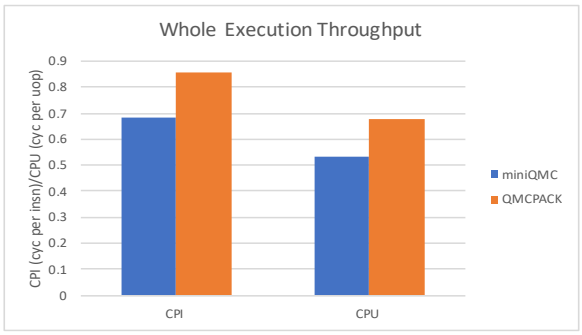
Figure 3 shows the branch behavior of miniQMC and QMCPACK. Looking at the whole execution on the left, QMCPACK does more frequent branching and has a much lower branch misprediction rate than miniQMC. MiniQMC

Cache MPKI	L1D	L2	L3
miniQMC - Determinant	3.44	2.32	1.3
QMCPACK - Determinant	87.42	0.04	11.75
miniQMC - Jastrow	1.69	1.59	0.69
QMCPACK - Jastrow	18.81	0.97	2.14

TABLE V
CACHE MISSES PER THOUSAND INSTRUCTIONS (MPKI)

implements only one walker, so does not have an inner loop that iterates over the walkers as QMCPACK does. This may account for some of the differing branch behavior since this walker loop is implemented within the main computational loop. To fully understand the difference in behavior, we need to look more deeply into each key kernel. Figure 3 on the right indicates that the largest differences in branches per instruction between miniQMC and QMCPACK come from the *Jastrow* and *SPO* kernels. The misses per branch metric is showing the largest differences in the *Distance* and *Jastrow* kernels.

Cache behavior in terms of misses per thousand instructions (MPKI) and cache bandwidths also show large differences between miniQMC and QMCPACK as seen in Table V. We only include kernels with large differences and where the magnitude of MPKI is greater than one. Looking at this data, we see that all of these MPKIs are of relatively small magnitude, with perhaps the exception of QMCPACK *Determinant* L1 data cache. The largest differences between



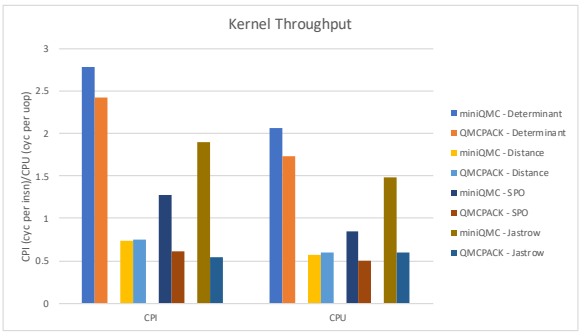
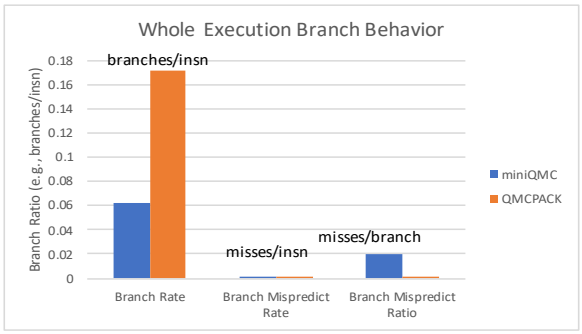


Fig. 2. Throughput in Cycles per Instruction (CPI) and Cycles per Micro-op



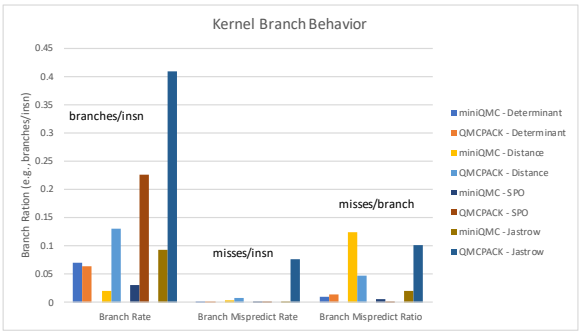


Fig. 3. Whole Execution and Kernel Branch Behavior

Cache Bandwidth (GB/sec)	L1L2 Total	L1L2 Load	L1L2 Evict
miniQMC	2.6	1.98	0.49
QMCPACK	30.0	16.45	13.48
Cache Bandwidth (GB/sec)	L2L3 Total	L2L3 Load	L2L3 Evict
miniQMC	2.57	1.92	0.65
QMCPACK	14.08	5.38	8.7

TABLE VI
Determinant KERNEL CACHE BANDWIDTHS

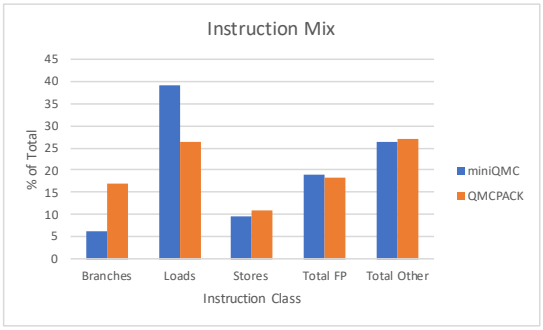


Fig. 4. Instruction Mix

miniQMC and QMCPACK are in the *Determinant* and *Jastrow* kernels. Perhaps these two functions in the two applications implement/operate on different data structures with different data layouts. The working set size, although the input data is supposed to be equivalent, may be much larger in QMCPACK. Detailed code inspection is required and remains for future work. This difference in MPKI may help explain the low throughput of QMCPACK relative to miniQMC.

Cache bandwidth data again shows the *Determinant* kernel being an outlier, which corresponds to cache MPKI. All of the other kernels show such low cache bandwidth utilization that we do not include their data here. The *Determinant* kernel of QMCPACK has relatively high bandwidth usage primarily between the L1 and L2 cache, but does not exceed bandwidth limits. Note that this is per-process (per-core) bandwidth data. According to [12], sustainable per-core read/write cache bandwidth between L1 and L2 is around 60–80/20–40 GB/sec per-core for our platform. The range accounts for the data size, ranging from small to large. [12] also reports sustained per-core read/write bandwidth between L2 and L3 for our platform as approximately 30–70/18-22 GB/sec.

Note that we did use performance counter events to measure stall activity due to cache and memory behavior. The *Determinant* kernel in QMCPACK exhibits the largest percentage of stall cycles due to the cache/memory subsystem, with about 40% of total cycles due to L1D pending misses and about 30% from L2 pending misses and memory accesses. This supports the observed data in Tables V and VI.

Looking at the instruction mix in Figure 4 we see that the two applications are similar except in the percentage of loads. MiniQMC executes significantly more load instructions than does QMCPACK. This could be because QMCPACK does much more AVX512 than miniQMC as shown in Table VII. AVX512 operates on wide data words, so the actual load instructions issued to memory fetch wider words, meaning fewer overall memory instructions issued. Because miniQMC

FP Breakdown (% of total FP)	miniQMC	QMCPACK
AVX512	3.9	11.5
AVX2	0.02	0.04
SSE	0.0	0.38
Scalar	15.0	6.26
Vectorization Rates (% of total by precision)	**miniQMC**	**QMCPACK**
SP	0.0	0.0
DP	20.98	65.49

TABLE VII
FP BREAKDOWN AND VECTORIZATION RATES

does relatively smaller number of AVX512 instructions, its scalar FP instruction percentage is large.

Table VII shows the vectorization rates for single-precision and double-precision FP instructions. All of the vectorization for both applications is done on double-precision FP instructions. This is probably because the data in the main kernels of both apps is double- rather than single-precision FP. Note the large difference between miniQMC and QMCPACK in AVX512 vectorization. Comparatively, miniQMC does minimal vectorization.

The takeaway from this kernel data is that we clearly see the function profile differences in the *Jastrow* and *SPO* kernels for various non-cache-related metrics. However, we also see large differences in cache and memory-related metrics in the *Determinant* kernel, which may indicate a mis-match in data layout, data structures, and/or working set size between the two applications, in spite of executing the same problem. Together, these noted differences could generate the overall execution differences.

D. Identifying Hardware-Level Bottlenecks using TMA

Figure 5 shows TMA for miniQMC and QMCPACK, starting at the top level. At Level 1, we see that both applications are backend bound, meaning there is a performance issue in the microarchitecture backend, which indicates that micro-ops are not being delivered to the issue pipe due to lack of resources for accepting them in the backend. This could be either because of execution stalls due to the memory subsystem (memory bound) or stalls due to sub-optimal execution port utilization (core bound). Note that miniQMC is also bound on bad speculation.

Drilling further down the backend hierarchy, we see that at level 2 both applications again exhibit the same behavior in that they are both memory bound and slightly core bound. At the third level of the backend bound hierarchy, execution stalls are occurring due to DRAM issues for both applications. The fourth level shows that the DRAM causes execution stalls due to both its bandwidth (for miniQMC and QMCPACK) and its latency (QMCPACK). QMCPACK has more complexity in function than does miniQMC. So it being also latency bound where miniQMC is not is not necessarily surprising. Although not shown in the figure, drilling down on the memory latency bottleneck shows that the latency of local DRAM accesses is the root cause of the execution stalls in the backend pipeline of this architecture. The takeaway is that both the proxy and the parent appear to be identifying performance degradation due to the same hardware resources.

If we drill down on the core bound bottleneck in Figure 6, we see that both applications show an execution port utilization bottleneck. This indicates that port over-utilization could be causing excessive execution stalls in the backend. However, drilling down further (not shown in the figure), we find that multiple execution ports are not simultaneously over-utilized. Examining the performance counter metrics we collect in addition to TMA, we did see port utilization of port 0 (executes integer, branch, and vector instructions) reach between 60 - 70% for some kernels in both applications. Ports are associated with queues and from queueing theory, a rule of thumb is that utilization around 70% could cause stalling. We wonder if the core boundedness identified by TMA could be indicating a problem, but the HPC threshold is actually too low to flag it in the methodology. We are currently exploring these sorts of issues in the methodology.

Pertaining to miniQMC's bad speculation bottleneck (Figure 5), drilling down to level 2 metrics (not shown), branch mispredicts exceed the threshold, identifying a bottleneck. Cycles are lost fetching the mispredicted execution path and from the subsequent pipeline flush. Why this happens in miniQMC and not in QMCPACK is not completely clear. miniQMC has much lower number of branches per instruction compared to QMCPACK and also has comparatively much larger basic blocks. This seems like these characteristics would lead to better branch prediction rates in miniQMC rather than worse. It could be that since miniQMC only encapsulates the key kernels in QMCPACK that we see only that behavior and there is enough additional code in QMCPACK with very different behavior from the kernels that dominates the overall branching behavior in QMCPACK. Further investigation is needed to more accurately identify this observed difference in branching behavior. We are presently analyzing per kernel TMA data to help determine this.

E. Is miniQMC a Good Proxy for QMCPACK?

Determining whether or not a proxy is a good representation of a parent application is somewhat ambiguous and really depends on how the proxy will be used. Therefore, we conclude that the best answer to whether or not a proxy is an imposter is to not answer the question directly, but rather provide information on key characteristics and behaviors and a guide to aid users in their decision as to whether or not to use a proxy in lieu of its parent.

Our conclusion is that miniQMC is a good proxy for QMCPACK for certain cases, but does not faithfully model QMCPACK in every aspect. At the whole-application level, hardware bottlenecks and kernel-only execution profiles are essentially the same. When looking at the entire execution, these applications differ in key metrics pertaining to throughput, branch and cache behavior, instruction mix, and vectorization. The per-kernel behavior shows these differences coming primarily from the *Determinant* and *Jastrow* kernels.

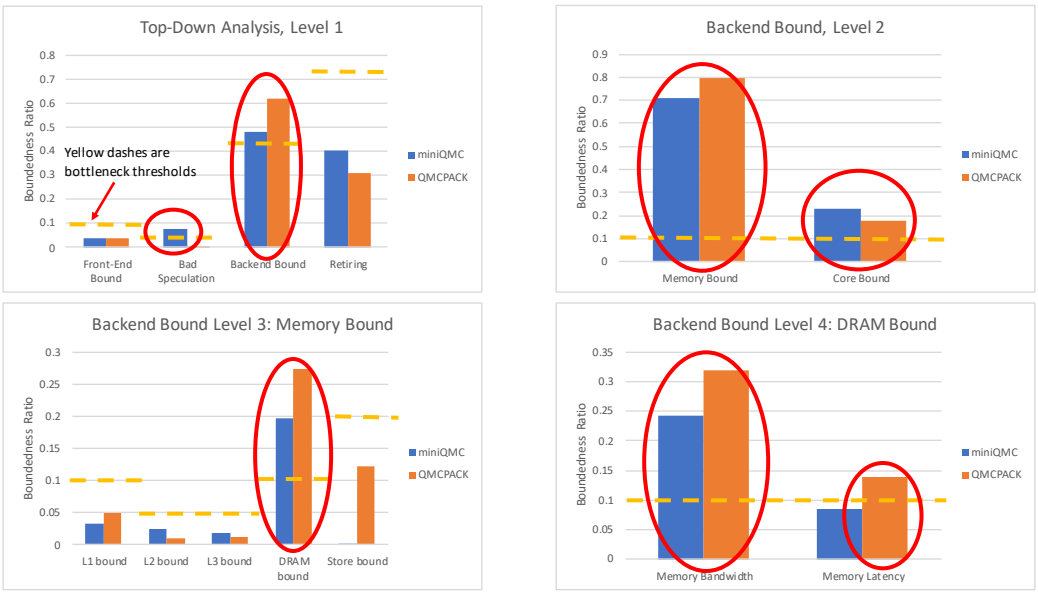


Fig. 5. TMA backend bound

Characteristic	Good Proxy?	Characteristic	Good Proxy?	Characteristic	Good Proxy?
MPI Comm	Red	Kernel Execution Profile	Yellow	Vectorization	Yellow
Front-end Bound	Green	Branch/insn	Red	Insn Mix	Red
Bad Speculation	Green	Branch miss/insn	Yellow	L1D/L2/L3 Cache MPKI	Red
Back-end Bound	Green	Branch miss/branch	Red	L1L2All BW	Red
Retiring	Green	L2L3Total& Evict BW	Red	L2L3Load BW	Yellow
CPI	Yellow	CPU	Yellow	Roofline	Green

TABLE VIII

SUMMARY OF MINIQMC/QMCPACK REPRESENTATIVENESS. GREEN INDICATES REPRESENTATIVE, YELLOW IS PARTIALLY REPRESENTATIVE, RED DENOTES NOT REPRESENTATIVE. LEVEL THRESHOLDS ARE AN ORDER OF MAGNITUDE DIFFERENCE IN THE PARTICULAR CHARACTERISTIC FOR RED, LESS THAN ORDER OF MAGNITUDE BUT STILL A DIFFERENCE WITHIN REASONABLE MEASUREMENT TOLERANCE FOR YELLOW, GREEN IS THE SAME.

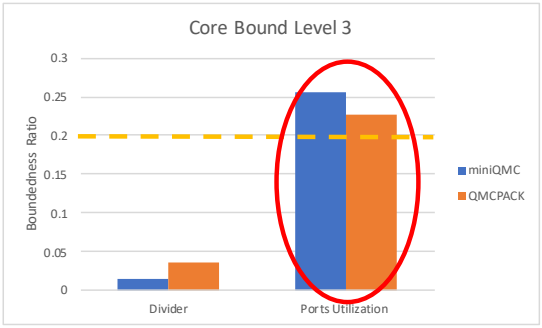


Fig. 6. TMA core bound

We summarize similarities and differences between miniQMC and QMCPACK in Table VIII. All of the Top-Down Analysis characteristics are green because the hardware bottlenecks identified are the same in both applications. The kernel execution profile is labeled yellow because there is an approximate 3% difference in execution percentage in the the Two-Body Jastrow kernel. Cache MPKI is red because there is an order of magnitude difference between miniQMC and QMCPACK for all cache levels. We realize the level thresholds are somewhat subjectively defined and it probably depends on the individual characteristic. However, the chart is a guide and we encourage users to assess specific quantitative differences in individual characteristics in making their use decision.

The methodology we follow is useful for identifying proxy representativeness of its parent. We examine the applications from both the software (kernels and kernel functions) and the hardware perspective, which did provide useful information. This methodology can and will be applied in the future to study additional proxy/parent pairs.

V. ACKNOWLEDGEMENT

This research was supported by the Exascale Computing Project (ECP), Project Number 17-SC-20-SC, a collaborative effort of two DOE organizations, the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem including software, applications, hardware, advanced system engineering, and early testbed platforms, to support the nation's exascale computing imperative. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE- NA0003525.

REFERENCES

[1] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[2] O. Aaziz, J. Cook, J. Cook, T. Juedeman, D. Richards, and C. Vaughan, "A methodology for characterizing the correspondence between real and proxy applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 190–200.

[3] O. Aaziz, J. Cook, J. Cook, and C. Vaughan, "Exploring and quantifying how communication behaviors in proxies relate to real applications," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, Nov 2018, pp. 12–22.

[4] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 35–44.

[5] https://hpctoolkit.org, "Hpctoolkit." [Online]. Available: https://hpctoolkit.org

[6] https://software.intel.com/en-us/vtune-amplifier-help, "Intel vtune amplifier 2019 user guide." [Online]. Available: https://software.intel.com/en-us/vtune-amplifier-help

[7] A. Agelastos *et al.*, "The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications," in *SC'14: Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 154–165. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.18

[8] https://github.com/QMCPACK/miniqmc/wiki/Comparing-performance-with-QMCPACK, "Comparing performance with qmcpack." [Online]. Available: https://github.com/QMCPACK/miniqmc/wiki/Comparing-performance-with-QMCPACK

[9] https://docs.qmcpack.org/qmcpack_manual.pdf, "Qmcpack: User's guide and developer's manual." [Online]. Available: https://docs.qmcpack.org/qmcpack_manual.pdf

[10] https://github.com/QMCPACK/miniqmc/wiki/How-to-build-and-run-miniQMC, "How to build and run miniqmc." [Online]. Available: https://github.com/QMCPACK/miniqmc/wiki/How-to-build-and-run-miniQMC

[11] https://github.com/RRZE-HPC/likwid, "Likwid performance monitoring and benchmarking suite." [Online]. Available: https://github.com/RRZE-HPC/likwid

[12] S.D.Hammond, C.T.Vaughan, and C.Hughes, "Evaluating the intel skylake xeon processor for hpc workloads," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*, July 2018, pp. 342–349.