Raexplore: Enabling Rapid, Automated Architecture Exploration for Full Applications

Yao Zhang, Prasanna Balaprakash, Jiayuan Meng, Vitali Morozov, Scott Parker, and Kalyan Kumaran
Argonne National Laboratory
{yaozhang, pbalapra, jmeng, morozov, sparker, kumaran}@anl.gov

Abstract

We present Raexplore, a performance modeling framework for architecture exploration. Raexplore enables rapid, automated, and systematic search of architecture design space by combining hardware counter-based performance characterization and analytical performance modeling. We demonstrate Raexplore for two recent manycore processors IBM BlueGene/Q compute chip and Intel Xeon Phi, targeting a set of scientific applications. Our framework is able to capture complex interactions between architectural components including instruction pipeline, cache, and memory, and to achieve a 3–22% error for same-architecture and cross-architecture performance predictions. Furthermore, we apply our framework to assess the two processors, and discover and evaluate a list of architectural scaling options for future processor designs.

1. Introduction

Over 20 years ago, supercomputer pioneer Seymour Cray famously made an analogy on computer design "If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens?" Today, we see both types of computers in the marketplace, and computer architects face an even wider spectrum of design choices on core complexity, memory hierarchies, parallelism, and special-purpose accelerators. Furthermore, the processor design landscape is becoming increasingly more dynamic, as we see mainstream processors both trickling up (e.g., ARM, DSP, GPU) and trickling down (e.g., Atom, Xeon Phi) in their design space to meet the demands for a range of emerging applications in scientific computing, data analytics, gaming, wearable devices, computer vision, etc.

For a big-picture view of this background, Figure 1 sketches today's processor landscape and macro trends in terms of single-thread performance and throughput performance. In Figure 2, we select eight representative processors and position them in a multi-dimensional design space in terms of their architectural features. The main observation is that the design space is vast in terms of both high dimensionality and large dynamic range for each dimension. We list eight major architectural features (dimensions) ranging from core complexity to memory hierarchies, not to mention other relatively minor features such as branch prediction, prefetching, and memory management. We use the ratio between the highest and lowest value to measure the span of the dynamic range of each feature (dimension). The observed span ranges from 4× up to 78×.

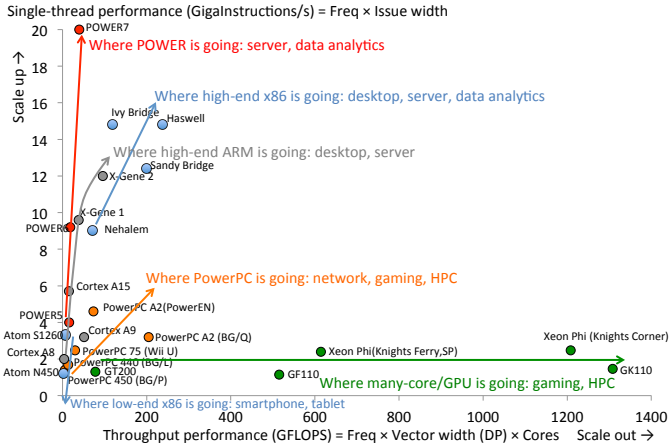


Figure 1: Processor landscape and trends. Processors in the same family/catergory are color coded.

Fundamentally, architecture design is driven by applications. Architectural evaluation and comparison for a diverse set of current processors are challenging because they often require significant human efforts to port applications to different architectures [8, 31]. Studying the performance of future processors is even more challenging, as the hardware is not yet available. While commonly used simulation-based techniques could provide highly accurate results, they are prohibitively slow to handle the combinatorial explosion of design choices in a multi-dimensional space and thus often limited to studying kernels and benchmarks rather than larger programs, mini-apps, and even full applications [3, 39, 41, 42].

In this work, we aim to address this architecture design challenge by developing Raexplore (**R**apid **a**rchitecture **explore**, pronounced as *ray-xplore*), a performance modeling framework to reduce the needs for application porting in architectural comparison, and to serve as a fast, first-order architecture explorer to complement slower but more accurate simulation-based techniques. In particular, we make the following contributions in this paper. First, we develop a methodology that combines experimental performance characterization and analytical performance modeling to enable rapid and systematic architecture exploration. Second, we develop analytical models for two recent manycore processors IBM Blue Gene/Q compute chip and Intel Xeon Phi. We show that our models could capture complex interactions between architectural components including instruction pipeline, cache, and memory, and achieve a 3–22% error for same-architecture

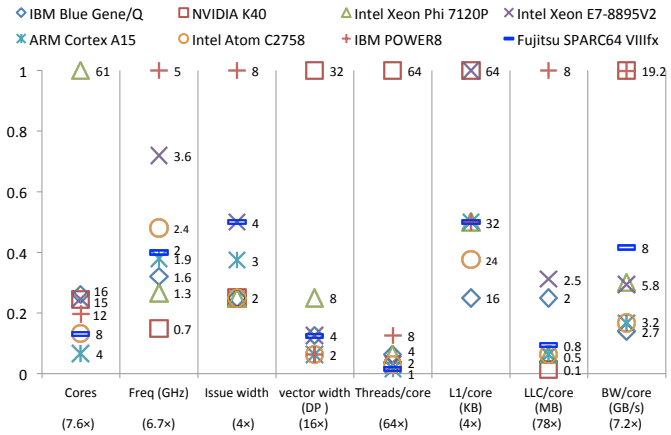


Figure 2: Mutli-dimensional processor design space. For each architectural feature, the numbers are normalized between 0 and 1 based on the highest value. The numbers to the right of each marker are absolute values. The numbers in parentheses below the horizontal axis are the ratios between the highest and lowest value, which measure the span of the dynamic range of each feature (dimension) for the eight processors.

and cross-architecture performance predictions. Third, using our framework, we analyze processor performance for a set of scientific applications, and suggest and evaluate a list of architectural design choices.

The rest of the paper is organized as follows. Section 2 and Section 3 describe our performance modeling methodology and the developed analytical models. Section 4 presents the experiments to validate our performance models. Section 5 and Section 6 apply our framework to analyze processor performance for a set of scientific applications and explore the design space for future architectures. Section 7 and Section 8 respectively discuss related work and the software release status of Raexplore. Section 9 concludes and describes future research directions.

2. Methodology

We have three goals in mind in developing a performance modeling methodology: (1) handle real-world large programs/applications, instead of kernels or benchmarks, as it is not common in practice that a single kernel dominates the application runtime; (2) explore architecture design space rapidly; and (3) capture complex effects and interactions of major architectural features and predict performance accurately. To this end, we develop a methodology that combines experimental hardware counter-based performance characterization and analytical performance modeling. The hardware counter-based approach provides a fast way to characterize performance for large applications on an existing baseline architecture. To project performance for future or different architectures, we develop analytical models that take in baseline performance characteristics and produce performance predictions. Analytical modeling reduces the process of architecture

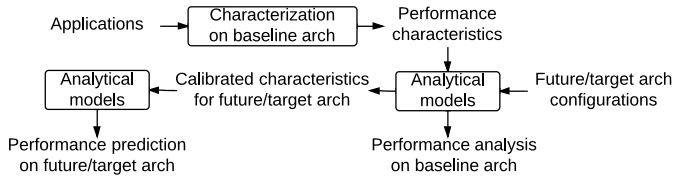


Figure 3: Performance modeling framework.

exploration to a matter of merely evaluating a set of mathematical formulas, enabling fast search of the vast design space.

Figure 3 shows our performance modeling framework. It first takes in a set of targeted applications and characterizes their performance on a baseline architecture. The performance characteristics are a set of performance events measured by hardware counter-based tools such as PAPI [38], Intel VTune [25], and IBM HPM [21]. The performance characteristics are then calibrated for a target architecture configuration using analytical performance models. The target could be either a future design of the baseline architecture, or a different current architecture (e.g., Blue Gene/Q as baseline and a hypothetical next Blue/Gene processor as target, or Blue Gene/Q as baseline and Xeon Phi as target). Finally, the analytical models produce performance analysis for the baseline architecture and performance predictions for the future/target architecture. Note that the analytical models include both the models for performance analysis (e.g., to derive the time of instruction execution, memory access, and their overlap) and the models for performance characteristics calibration to account for differences in architecture features such as cache size and instruction latency.

In order to project from a baseline to target architecture, analytical models should be able to capture the architectural changes in instruction pipeline and memory hierarchy. The wider the architectural difference is, the more challenging to model it. We list three types of baseline/target scenarios in the order of increased challenge: (1) project within the same processor line (e.g., from BGP to BGQ[1], from NVIDA Fermi to Kepler GPU), (2) project across similar architectures (e.g., from BGQ to Xeon Phi), and (3) project across different architectures (e.g., from BGQ to GPU, from BGQ to x86 multicores). For example, to project from BGQ to GPU, we need to model the GPU's hardware mechanism to coordinate massively parallel threads and their collective memory access. Projection across different architectures is further complicated by ISA and compiler differences (Section 4.4). In this work, we select two relatively similar architectures for our study: BGQ and Xeon Phi. We model their major architecture features, but currently leave out the software aspects such as compiler differences and thread management overhead.

In comparison with a pure analytical modeling approach [40], which relies on static program analysis for model inputs (e.g., operation count) and thus has difficulties in han-

[1] BGP and BGQ respectively stand for IBM Blue Gene/P and Blue Gene/Q compute chip.

Table 1: Comparison of performance study methodologies.

Methodology	Speed	Accuracy	Large applications	Future architecture
Experimental	✓	✓	✓	✗
Simulation	✗	✓	✗	✓
Analytical	✓	✗	✗	✓
Experimental + analytical	✓	✓	✓	✓

dling large applications and capturing program-hardware interactions (e.g., cache hit rate), our combined experimental and analytical approach jump-starts our models with accurate baseline performance characteristics, which have already accounted for compiler optimizations and hardware architecture effects. Complementary to simulation techniques [41], which are more accurate but several orders of magnitude slower than hardware execution, our approach could serve a fast, first-order architecture explorer. Statistical techniques and machine learning [26, 32] have been shown to be effective in handling the complexity of design space. However, because these techniques are driven by experimental/simulation data instead of hardware inner working mechanism, they are less explicative and insightful than our mechanism-driven analytical modeling approach. Furthermore, these techniques cannot predict performance for future processors with new features (e.g., a GPU memory coalescing feature that the processor training set does not cover), while the analytical approach could in principal model and incorporate such new features. Table 1 compares our methodology with existing approaches in terms of their capabilities.

3. Analytical performance models

We develop performance models for BGQ and Xeon Phi. We use a set of performance events (Table 2) monitored by hardware counters to characterize application performance on a baseline processor (Figure 3). To describe a baseline/target architecture configuration, we choose a set of hardware parameters that reflect major architectural features and by general practice have good performance impact (listed in Table 3). The architecture parameters are according to the references [19, 23, 24, 37]. Note that the bandwidth numbers are not their theoretical design values, but measured peak values using synthetic stream benchmarks [22, 37]. The inputs to our analytical models are the number of performance events measured by hardware counters and baseline/target architecture configurations; the outputs are analyzed and predicted runtime and its breakdown in terms of instruction execution and memory access.

Accurate performance models should be able to capture the effects and interactions between architectural components, and are the key to the success of our methodology. There are two major challenges. The first one is to deal with the complexity of the studied architecture, which has multiple cores, multiple pipelines, and multiple levels of memory hierarchy. We need deep understanding and analysis of performance to develop accurate models at the right level of abstraction. The second

Table 2: Events monitored by hardware counters.

Short names	Definition
time	execution time (cycles)
instInt	integer instructions
instFP	floating-point instructions
numAccess	memory reads and writes
hitsL1	L1 hits
hitsLLC	LLC hits
LD	LLC cacheline loads
ST	LLC cacheline stores

Table 3: Architecture parameters for BGQ and Xeon Phi 7120P. All latency values are in CPU cycles. L1p is the prefetch buffer on BGQ, which is between L1 and LLC.

Processor	Short names	Blue Gene/Q	Xeon Phi
Frequency	*freq*	1.6	1.24
Cores	*cores*	16	61
Integer inst latency	*IntL*	3	3
Floating-point inst latency	*FPL*	5	4
Max threads per core	*max TPC*	4	4
Inst streams per thread	*SPT*	1	2
L1 size (KB), latency	*sizeL1*, *latL1*	16, 3	32, 3
L1p size (KB), latency	*sizeL1p*, *latL1p*	4, 16	NA
LLC/L2 size (MB), latency	*sizeLLC*, *latLLC*	16, 42	30.5, 23
Mem BW (GB/s), latency	*BW*, *latDDR*	28, 213	177, 750

challenge is to deal with uncertainties that arise when hardware performance counters do not provide sufficient information required by our models (e.g., instruction- and memory-level parallelism, integer and FP instruction execution overlap). In this case, we need to develop upper and lower bounds of interested quantities. We approximate the unknown quantity using the average of its lower and upper bound. To improve the accuracy of such approximation, we divide the application to code blocks (individual functions or loops); because if the code blocks are fine-grained enough, the performance of each code block tends to be dominated by a single performance factor, which could be instruction execution, memory latency, or memory bandwidth. All code blocks together cover the whole application. In general, the finer granularity they are specified at, the more accurate the performance prediction is, at the cost of annotation effort (insert hardware counter monitors) and monitoring overhead. We will present both top time-consuming individual code blocks (together cover more than 90% of application) and aggregated performance breakdown of all code blocks (cover 100% of application) to guarantee representativeness; users can optionally examine the rest code blocks.

We now describe our performance models. As a reference, Table 4 lists the short names for performance metrics used in our models. For a given application, we divide it to code blocks and model the total application execution time as $timeTotal = \sum_{i=1}^{n} timeCodeBlock_i$.

Table 4: Short names for various performance metrics (in alphabetical order).

Short names	Definition
accessPerInst	memory accesses per instruction
activeCores	number of active cores
aIL	average instruction execution latency (cycles)
aML	average memory access latency (cycles)
bwCC	bandwidth per core per cycle
CPI	cycles per instruction
ILP	instruction level parallelism
instC	effective number of instructions per core
instIntC	integer instructions per core
instFPC	floating-point instructions per core
IPC	instructions per cycle
MLP	memory level parallelism
MPC	memory accesses per cycle
timeBW	memory bandwidth time (cycles)
timeCodeBlock	runtime for a code block (cycles)
timeInst	instruction execution time (cycles)
timeInstInt	integer instruction execution time (cycles)
timeInstFP	floating-point instruction execution time (cycles)
timeLat	memory latency time (cycles)
timeMem	memory access time (cycles)
timeOverlap	overlap time between timeInst and timeMem (cycles)
timeTotal	total application runtime (cycles)
SPT	instruction streams per thread
TPC	threads per core

$timeCodeBlock = timeInst + timeMem - timeOverlap$, where $timeInst$ is the instruction execution time, $timeMem$ is the memory access time, and $timeOverlap$ is the overlap time between instruction execution and memory access. Ideally, we want memory access time to be completely hidden (overlapped by instruction execution) using hardware features such as caching and simultaneous multithreading. However, this is often not the case in reality due to cache misses and the lack of instruction parallelism.

While $timeOverlap$ is not directly measurable, it could be estimated on the baseline architecture as $timeOverlap_{base} = timeInst_{base} + timeMem_{base} - timeCodeBlock_{base}$, where $timeCodeBlock_{base}$ is the measured time, and $timeInst_{base}$ and $timeMem_{base}$ are modeled time on the baseline architecture. For a target architecture, we assume $timeOverlap$ scales along with $timeInst$ and $timeMem$: $timeOverlap = \lambda \times timeOverlap_{base}$, where the scaling factor λ is estimated as the average of the time scaling ratio of instruction execution time and memory access time $\lambda = avg(\frac{timeInst}{timeInst_{base}}, \frac{timeMem}{timeMem_{base}})$.

The following subsections describe the models of the instruction pipeline and memory subsystem that are respectively used to estimate $timeInst$ and $timeMem$.

3.1. Instruction pipeline

We model the instruction execution time, which includes the pipeline stalls due to instruction dependencies and structural hazards, but excludes the stalls due to dependencies on memory access (assuming zero memory latency). Section 3.2 will separately model the memory access time.

Both BGQ and Xeon Phi feature simultaneous multithreading (SMT), where multiple threads could be executed in an interleaved fashion to increase the instruction-level parallelism (ILP) and to hide memory latency. Both the BGQ and Xeon Phi core have two instruction pipelines: one supports (vector) floating-point (FP) instructions, and the other does not. We will refer to the two pipelines in loosely defined terms as the integer pipeline and the FP pipeline; we will also refer to all general-purpose instructions (including control flow and load/store) as integer instructions. The integer pipeline on both BGQ and Xeon Phi support vector loads/stores instructions. The FP pipeline of Xeon Phi is actually versatile, as it can execute all other general-purpose instructions as well.

In terms of utilizing the two pipelines, BGQ could simultaneously issue one integer instruction and one FP instruction to the two pipelines, but these two instructions have to be from two different threads. On Xeon Phi, a thread could issue two instructions in one cycle to both the integer and FP pipelines subject to certain instruction pairing rules, but a thread cannot consecutively issue instructions in back-to-back cycles (in the next cycle, a different thread has to take the turn to issue instructions). Effectively, both BGQ and Xeon Phi require at least two threads to fully utilize both pipelines. One advantage of Xeon Phi is that, in the case of using a single thread per core, Xeon Phi pipeline observes more ILP than BGQ, because each Xeon Phi thread has two instruction streams per thread, while a BGQ thread has only one instruction stream; however, we have not observed this ILP advantage of Xeon Phi in the case of multiple threads per core.

We model the instruction execution time as $timeInst = \frac{instC}{IPC}$, where IPC is instructions per cycle (assuming zero memory latency as discussed earlier), and $instC$ is the effective number of instructions taking into account the overlap between the execution of integer and FP instructions (effectively we treat two pipelines as a single pipeline in our abstract processor model). In an ideal situation of sufficient instruction-level parallelism (ILP), instruction execution is fully pipelined and $IPC = 1$. In reality, ILP is limited by instruction dependency and contention for functional units.

The effective number of instructions $instC$ depends on the degree of execution overlap of integer and FP instructions. A complete overlap means a better utilization both integer and FP pipelines. However, this is not possible in reality due to limited instruction and thread parallelism. As discussed earlier, both BGQ and Xeon Phi require at least two threads to fully utilize the two pipelines. If the baseline and target architecture use the same number of threads per core, $instC = \alpha * instC_{base}$, where α is the factor that takes into account ISA and compiler differences; if the target architecture uses a different number of threads per core (TPC), we estimate $instC$ using the following conditional function, which essentially assumes maximum integer and FP instruction overlap if using more than two threads per core, minimum overlap if using one thread per

core, and an average overlap if using two threads per core.

$$instC = \begin{cases} instC_{max} & \text{if } TPC = 1 \\ avg(instC_{max}, instC_{min}) & \text{if } TPC = 2 \\ instC_{min} & \text{if } TPC > 2 \end{cases}$$

$$instC_{max} = sum(instIntC, instFPC)$$

$$instC_{min} = max(instIntC, instFPC)$$

instIntC and *instFPC* are respectively the number of integer and FP instructions per core, calculated as $instIntC = \beta \times \frac{instInt_{base}}{activeCores}$ and $instFPC = \gamma \times \frac{instFP_{base}}{activeCores}$, where β and γ are scaling factors to account for ISA and compiler differences.

We model instructions per cycle as $IPC = min(1, \frac{ILP}{aIL})$, where *aIL* is average instruction latency. Note that the maximum *IPC* is 1. This equation models the workings of the instruction pipeline. In an ideal situation, $IPC = 1$, when there is sufficient *ILP*, that is $ILP \geq aIL$. In the worst case, $IPC = \frac{1}{aIL}$, when $ILP = 1$. *aIL* is calculated as a weighted average of integer and FP instruction latencies $aIL = \frac{IntL \times instIntC + FPL \times instFPC}{instIntC + instFPC}$. *ILP* is derived from ILP_{base} taking into account the differences in threads per core (*TPC*) and instruction streams per thread (*SPT*),

$$ILP = \begin{cases} ILP_{base} + SPT - SPT_{base} & \text{if } TPC = 1 \\ ILP_{base} + TPC - TPC_{base} & \text{if } TPC > 1 \end{cases}, \text{ where}$$

$$ILP_{base} = aIL_{base} \times IPC_{base}$$

$$aIL_{base} = \frac{IntL_{base} \times instIntC_{base} + FPL_{base} \times instFPC_{base}}{instIntC_{base} + instFPC_{base}}.$$

We estimate IPC_{base} as the average of its lower and upper bound. At the lower bound, there is no overlap of integer and FP instruction execution; at the upper bound, there is a complete overlap, except in the case of $TPC = 1$ when there is no overlap. Also note that *IPC* cannot exceed 1.

$$IPC_{base} = avg(IPC_{base(min)}, min(1, IPC_{base(max)}))$$

$$IPC_{base(min)} = \frac{instC_{base(min)}}{timeInst_{base}}, \; IPC_{base(max)} = \frac{instC_{base(max)}}{timeInst_{base}}$$

$$instC_{base(max)} = sum(instIntC_{base}, instFPC_{base})$$

$$instC_{base(min)} = \begin{cases} sum(instIntC_{base}, instFPC_{base}) & \text{if } TPC = 1 \\ max(instIntC_{base}, instFPC_{base}) & \text{if } TPC > 1 \end{cases}$$

We estimate $timeInst_{base}$ as the average of its lower and upper bound respectively using a maximum and minimum ILP_{base} because ILP_{base} is not directly measurable by hardware counters. Note that $timeInst_{base(max)}$ cannot exceed $timeCodeBlock_{base}$, the measured execution time for this code block.

$$timeInst_{base} = avg(timeInst_{base(min)}, timeInst_{base(max)})$$

$$timeInst_{base(min)} = instC_{base(min)} \times CPI_{base(min)}$$

$$CPI_{base(min)} = max(\frac{aIL_{base}}{ILP_{base(max)}}, 1) = 1, \; ILP_{base(max)} \geq aIL_{base}$$

$$timeInst_{base(max)} = min(instC_{base(max)} \times CPI_{base(max)}, timeCodeBlock_{base})$$

$$CPI_{base(max)} = max(\frac{aIL_{base}}{ILP_{min}}, 1), \; ILP_{base(min)} = TPC_{base}$$

Finally, $instC_{base} = IPC_{base} \times timeInst_{base}$, which we use to calculate instruction time on the target architecture $timeInst = \frac{instC}{IPC}$. We can also estimate the integer and FP instruction overlap as $timeInstOverlap = timeInstInt + timeInstFP - timeInst$, where $timeInstInt = \frac{instIntC}{IPC}$ and $timeInstFP = \frac{instFPC}{IPC}$.

3.2. Memory subsystem

The memory performance could be either latency bound or bandwidth bound. Therefore, we model memory access time as $timeMem = max(timeLat, timeBW)$, where *timeLat* is the sum of memory access latency for all memory references and *timeBW* is the time to transfer all memory traffic (including prefetch traffic) over the memory bus.

$timeBW = \frac{(LDC_{base} + STC_{base}) \times lineSizeLLC}{bwCC}$, where *LDC*, *STC*, and *bwCC* are respectively loads per core, stores per core, and bandwidth per core per cycle, $LDC_{base} = LD_{base}/activeCores$, $STC_{base} = ST_{base}/activeCores$, $bwCC = \frac{BW}{activeCores \times freq}$.

$timeLat = \frac{numAccessC}{MPC}$, where *numAccessC* is the number of memory accesses per core and *MPC* is memory accesses per cycle. For the instruction pipeline, instruction level parallelism (*ILP*) directly impacts pipeline stalls and thus instructions per cycle (*IPC*). Similarly, memory access is also pipelined, and memory-level parallelism (*MLP*) directly impacts memory accesses per cycle (*MPC*), $MPC = min(1, \frac{MLP}{aML})$, where *aML* is average memory access latency, calculated as a weighted average of access latency to all levels of memory hierarchy, $aML = \frac{\sum_i latencyMemLevel_i \times numAccessMemLevel_i}{numAccessC}$. Note that $MPC = 1$, in an ideal situation of sufficent *MLP*, that is $MLP \geq aML$. *MLP* is calculated as

$$MLP = MLP_{base} + (ILP - ILP_{base}) \times accessPerInst_{base}$$

$$accessPerInst_{base} = \frac{numAccessC_{base}}{instC_{base}}$$

$$MLP_{base} = MPC_{base} \times aML_{base}, \; MPC_{base} = \frac{numAccessC_{base}}{timeLat_{base}}$$

We estimate $timeLat_{base}$ as the average of its lower and upper bound (note that $timeLat_{max}$ cannot exceed $timeCodeBock_{base}$). At the lower bound of $timeLat_{base}$, we use the minimum memory latency ($MPC = 1$); at the upper bound, the maximum memory latency is bound by average access latency to all memory hierarchies.

$$timeLat_{base} = avg(timeLat_{min}, timeLat_{max})$$

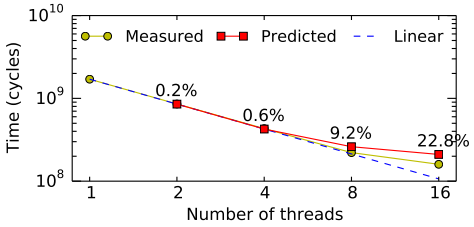
$$timeLat_{min} = \frac{numAccessC_{base}}{MPC_{base(max)}} = numAccessC, \; MPC_{base(max)} = 1$$

$$timeLat_{max} = min(\frac{numAccessC_{base}}{MPC_{base(min)}}, timeCodeBlock_{base})$$

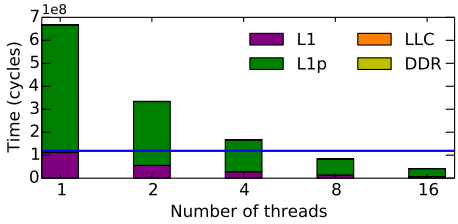
$$MPC_{base(min)} = \frac{MLP_{base(min)}}{aML_{base}} = \frac{1}{aML_{base}}$$

$$aML_{base} = \frac{\sum_i latencyMemLevel_{base(i)} \times numAccessMemLevel_{base(i)}}{numAccessC_{base}}$$

We use a well-known empirically observed power law relation to estimate the effects of cache size on miss rate $missRate = missRate_{base} \times (\frac{cacheSize}{cacheSize_{baseline}})^{-0.5}$. Hartstein et al. use a statistical model to analytically explain why the power law relation is obeyed [20]. To account for the effects of cache contention among multiple threads, we allocate an evenly divided portion of cache to each thread. Although our framework allows more sophisticated cache and contention models [2, 9, 10, 18] to be incorporated, we observe power law approximation and uniform allocation provide sufficiently accurate results, as we will show in Section 4. Regarding modeling cache coherency, our baseline measurements do include the effect of coherence traffic and we assume a linear scaling of this effect in performance prediction; our framework is extensible for advanced coherency models to predict the non-linear effect and the impact of changed coherence protocols.

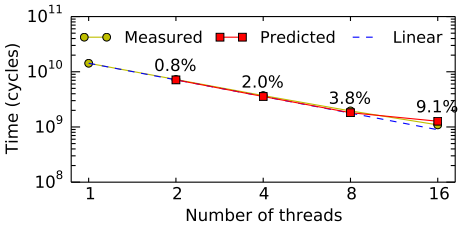


(a) Total time. Blue dashed line indicates linear time scaling.

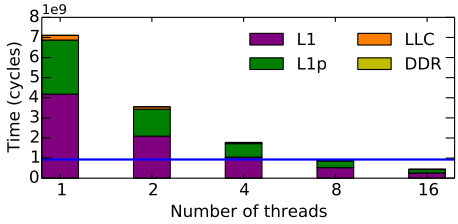


(b) Predicted memory latency time. Blue line indicates predicted memory bandwidth time.

Figure 4: Threads scaling performance prediction for *add2s*.



(a) Total time. Blue dashed line indicates linear time scaling.



(b) Predicted memory latency time. Blue line indicates predicted memory bandwidth time.

Figure 5: Threads scaling performance prediction for *dp*.

4. Model validation

We validate our models for (1) same-architecture performance prediction for threads scaling, cache contention, and simultaneous multithreading on BGQ, and (2) cross-architecture performance prediction from BGQ to Xeon Phi.

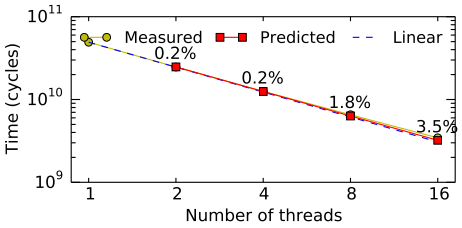
4.1. Threads scaling

We use our models to predict the runtime of executions on BGQ that use up to 16 threads based on the performance characterization for an execution that uses a single thread. We select several code blocks from a fluid dynamics code Nekbone [14], that solves a Poisson equation using a conjugate gradient method. The code blocks are representative of different scaling performance. Figures 4, 5, and 6, show the measured and predicted runtime respectively for code blocks named *add2s*, *dp*, and *grad*.

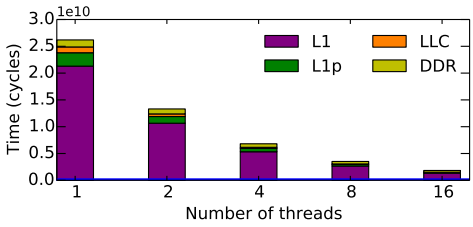
The prediction errors at 16 threads range from 3–22%. For example, in Figure 4a, the predicted time follows well with the measured time, and the performance stops scaling linearly at 4 threads. This is because the memory latency time becomes smaller than the memory bandwidth time at 4 threads (Figure 4b), which changes the performance from latency-bound to bandwidth-bound. Similarly, the linear performance scaling stops at 8 threads for *dp* (Figure 5); and for *grad*, the performance continues to scale linearly up to 16 threads because memory latency has always been the bottleneck, and it scales linearly with the number of threads (Figure 6).

4.2. Cache contention

We use our cache models based on power law approximation and uniform allocation to predict cache hit rate for 2-threads-per-core and 4-threads-per-core cases based on the performance characterization for the 1-thread-per-core case. Both baseline and target processor are BGQ. Table 5 lists the prediction results for four code blocks. Our simple cache model works very well and achieves a cache hit rate prediction error between 0.3% and 3.48%. For example, for the code block *dp*, we accurately predict the cache hit drop due to the contention of multiple threads.



(a) Total time. Blue dashed line indicates linear time scaling.



(b) Predicted memory latency time. Blue line indicates predicted memory bandwidth time.

Figure 6: Threads scaling performance prediction for *grad*.

Table 5: Predict L1 cache hit rate for contending threads.

Threads per core	Hit rate	*grad*	*add2s*	*glsc*	*dp*
1	measured	0.9573	0.9973	0.9919	0.9515
2	measured	0.9229	0.9781	0.9794	0.9224
	predicted	0.9396	0.9962	0.9885	0.9314
	error	1.78%	1.82%	0.93%	0.97%
4	measured	0.9118	0.96	0.9789	0.9086
	predicted	0.9146	0.9946	0.9838	0.903
	error	0.31%	3.48%	0.50%	0.62%

4.3. Simultaneous multithreading (SMT)

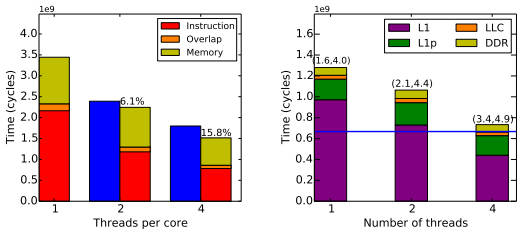
We use our models to predict SMT performance for 2-threads-per-core and 4-threads-per-core cases based on the performance characterization for the 1-thread-per-core case. Both baseline and target processor are BGQ. Figures 7, 8, and 9 show the measured and predicted runtime respectively for code blocks *grad*, *add2s*, and *dp*. The prediction errors for 4 threads per core range from 13.2–19.2%.

Take code block *grad* for example (Figure 7a). As we use more threads per core, both instruction execution time (red column) and memory access time (yellow column) decrease. The reduction of the instruction time is due to the increased *ILP* and the overlap of integer and FP instructions. The reduction of the memory access time is due to the increased *MLP*, despite the slight increase of average memory access latency *aML* due to the cache contention of simultaneous threads (Figure 7b).

In contrast, the total runtime for code block *add2s* does not reduce much at more threads per core (Figure 8a). This is because the majority of runtime is taken by the memory access time, which is bound by bandwidth and does not change with the number of threads per core (Figure 8b). The runtime at 4 threads per core actually increases rather than decreases. Further examination reveals that more dynamic instructions are executed, most likely due to the extra OpenMP thread management overhead. For *dp* (Figure 9a), from 2 to 4 threads per core, the runtime does not decrease as much as for *grad* because the memory access time becomes bandwidth-bound from latency-bound at 2 threads per core (Figure 9b).

4.4. Cross-architecture performance prediction

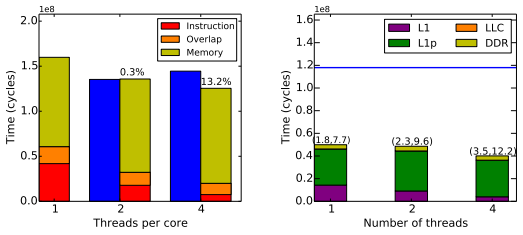
We use the performance characterization on the baseline processor BGQ to predict the performance on a target processor Xeon Phi. Figure 10 shows measured and predicted performance for three code blocks. We compare the prediction results for three models: “naive", “model”, and “with inst diff”. The “naive” model simply scales the runtime according to the difference in dynamic instruction count caused by the compiler and ISA differences. Both “model” and “with inst diff” use our performance models; “model” does not take into account dynamic instruction count difference, while “with inst diff” does. We examined the sources of the instruction



(a) Total time. Blue columns are measured time. At 1 thread per core, measured time = predicted time. The percentage number above a column is prediction error.

(b) Predicted mem lat time. Blue line indicates predicted mem BW time. The two numbers above a column are (MLP, aML).

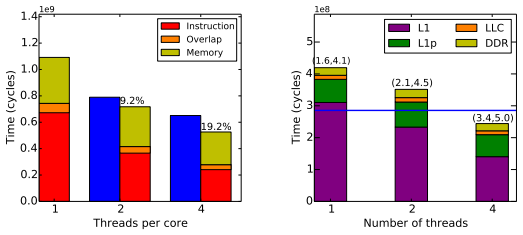
Figure 7: Predict SMT performance for *grad*.



(a) Total time. Blue columns are measured time. At 1 thread per core, measured time = predicted time. The percentage number above a column is prediction error.

(b) Predicted mem lat time. Blue line indicates predicted mem BW time. The two numbers above a column are (MLP, aML).

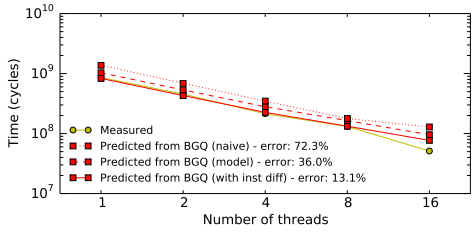
Figure 8: Predict SMT performance for *add2s*.



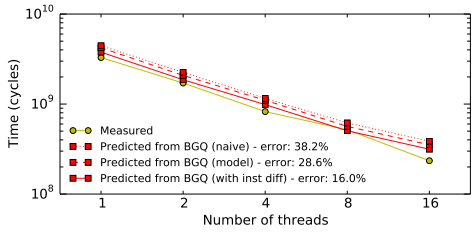
(a) Total time. Blue columns represent measured time. At 1 thread per core, measured time = predicted time. The percentage number above a column is prediction error.

(b) Predicted mem lat time. Blue line indicates predicted mem BW time. The two numbers above a column are (MLP, aML).

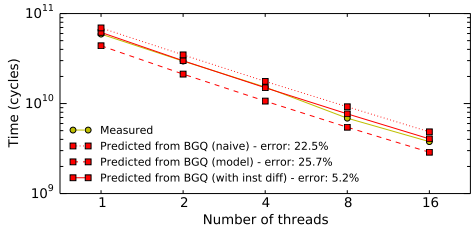
Figure 9: Predict SMT performance for *dp*.



(a) add2s



(b) glsc



(c) grad

Figure 10: Performance prediction from BGQ to Xeon Phi. In the legend, we show the average error for the predictions.

count difference and discovered they are mostly from compiler-generated instructions for prefetching and vector load/store pack/unpacking. For different code blocks, the observed instruction count for Xeon Phi could be up to 20% less and up to 40% more than that for BGQ. Overall, "model" produces significantly more accurate predictions than "naive"; "with inst diff" further reduces the errors to 5–16%.

One difficulty in cross-architecture performance prediction lies in a different dynamic instruction count resulted from ISA and compiler differences. We currently measure the dynamic instruction count on Xeon Phi to gauge the impact of this factor. Nevertheless, we do not require users to measure instruction count on target platforms; this is just an optional extra step to improve accuracy, especially for projections across very different architectures. Without it, we can still achieve reasonable accuracy and provide performance insights. Future work could use compiler techniques to estimate the instruction count change. Furthermore, we expect much smaller ISA and compiler differences in projecting performance within a processor line (e.g., BGP to BGQ, Xeon Phi to its next generation).

4.5. Summary

We have showed our models are able to capture complex effects and interactions of architectural components and performance factors including instruction pipeline, cache, memory bandwidth, and number of threads. Our models achieve good accuracy across a variety of validation experiments including threads scaling (3–22% error), cache contention (0.3–3.48% error), SMT performance (13–19% error), and cross-architecture prediction (5–16% error).

5. Performance Analysis

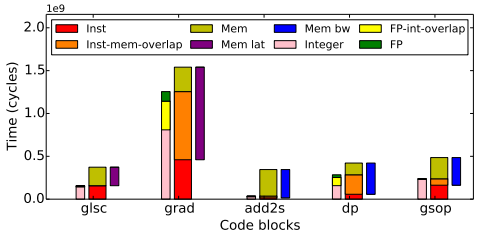
We demonstrate our models in analyzing processor performance for a set of applications. In our analysis, we provide instruction and memory time breakdown. For instruction time, we further break it down to integer and FP instruction time; for memory time, we further break it down to latency and bandwidth time (the latency time could be further divided to time to different levels of memory hierarchy). This type of analysis is different from performance characterization using raw hardware counter data [16] in that it processes the raw hardware counter-based performance characteristics with our models and provides important information on performance bottlenecks of the studied processor.

5.1. Application performance analysis

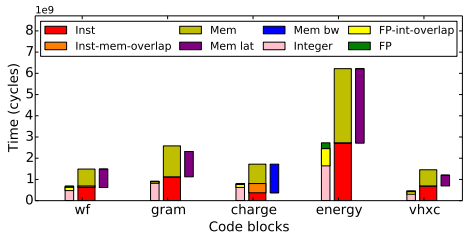
For our study, we select five scientific codes from the CORAL benchmarks [1]: Nekbone, Qbox, LULESH, AMG, and UMT. The CORAL benchmarks are formed according to the mission needs of the U.S. Department of Energy and currently used by three national labs (Oak Ridge, Argonne, and Livermore) to evaluate and design future architectures. All application codes have been parallelized using both MPI and OpenMP. In our experiments, we always select a combination of MPI tasks and OpenMP threads to fully utilize all cores of a processor and to minimize the time to solution. Note that this is a node-level study on processor and memory architecture, and we run MPI tasks on cores within a processor (all MPI communication traffic are included in the memory traffic).

Figure 11 shows the performance analysis for major code blocks in Nekbone and Qbox on BGQ. Two major observations are: (1) there is no single code block that takes more than 50% of the total application time, and (2) different code blocks observe different performance bottlenecks in integer/FP pipeline, memory latency, and memory bandwidth. Note that the memory performance for a code block could be either latency-bound or bandwidth-bound, so the memory time is completely taken by either latency time or bandwidth time.

Figure 12 shows the performance analysis for all five applications on BGQ. For each application, we aggregate the instruction and memory timings of all code blocks to derive the application-level performance analysis. The major observations are: (1) with the exception of Nekbone, most applications are bound more by memory latency time than by



(a) Nekbone



(b) Qbox

Figure 11: Performance analysis on BGQ for code blocks in Nekbone and Qbox. For each code block, we show three columns. The middle column shows instruction execution and memory access time breakdown; the left column show the integer and FP instruction execution time breakdown; the right column shows the memory latency and bandwidth time breakdown.

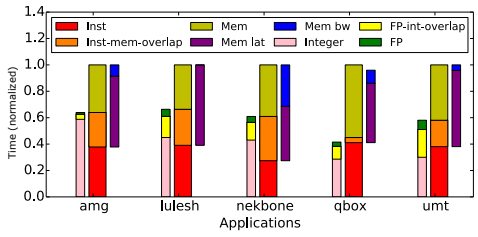


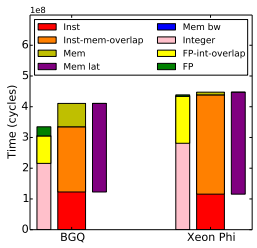
Figure 12: Performance analysis on BGQ for five applications. For detailed explanations on clustered columns, refer to the caption of Figure 11.

memory bandwidth time, which suggests cache improvements will benefit the performance; (2) most applications spend the majority of instruction execution time in processing integer instructions, which suggests adding more integer pipelines in a core would benefit the performance.

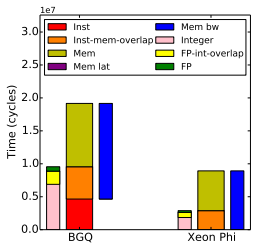
5.2. Architecture comparison

To compare the architecture features between BGQ and Xeon Phi, we analyze the performance of two code blocks that are respectively representative for two scenarios: (1) instruction execution (and memory latency) bound, and (2) memory bandwidth bound. Figure 13 shows the performance analysis results. To compare on a per-core basis, we have scaled the timings according to the core count difference between BGQ and Xeon Phi.

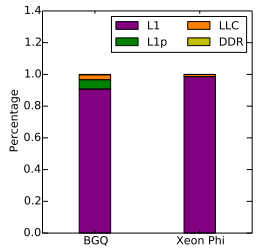
For code block *grad*, BGQ and Xeon Phi have comparable performance (Figure 13a). The performance of *grad* is mostly



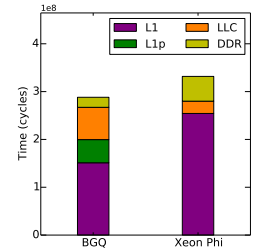
(a) Total time breakdown for *grad*.



(b) Total time breakdown for *add2s*.



(c) Memory hit distribution for *grad*.



(d) Memory latenty time breakdown for *grad*.

Figure 13: Performance comparison between BGQ and Xeon Phi for code blocks *grad* and *add2s*.

instruction execution bound and memory latency bound, and most of the memory latency is hidden (overlapped with instruction execution). The slightly longer instruction time on Xeon Phi is mostly due to the increased dynamic instruction count as a result of compiler-inserted memory prefetch instructions. The memory latency time on Xeon Phi is slightly higher, mostly due to its longer DDR access latency (Figure 13d and Table 3). Xeon Phi has a larger L1 cache (32 KB) than BGQ (16 KB), which results in a higher L1 cache hit rate (Figure 13c) and more time spent in accessing L1 and less time in LLC (Figure 13d).

For code block *add2s*, Xeon Phi performs about 2× better than BGQ (Figure 13b). This is mainly because *add2s* is mostly memory bandwidth bound, and Xeon Phi has 1.7× more per-core bandwidth than BGQ (Table 3).

In summary, BGQ and Xeon Phi have comparable instruction pipeline and cache performance. Although Xeon Phi has a larger L1 cache, this advantage is offset by its longer DDR latency. Xeon Phi has 1.7× more bandwidth per core than BGQ, which translates to almost the same ratio of real performance benefit for bandwidth-bound program.

6. Architecture Exploration

We demonstrate Raexplore in exploring architecture scaling options for BGQ. The studied architectural features include core count, L1 and LLC size, and memory bandwidth. By scaling these features in both directions (up and down), this type of study has a two-fold purpose: (1) evaluate the design balance of a current/baseline processor, and (2) explore scaling opportunities for its potential future design.

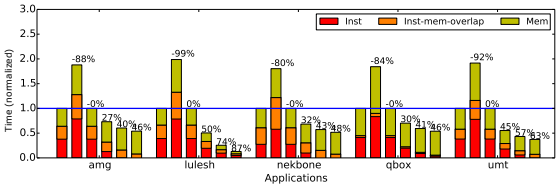


Figure 14: Core scaling. For each application, the 6 columns represent the total execution time for (baseline, 0.5, 1, 2, 4, 8), where the numbers in the brackets represent the scaling factor. The number above a column is the runtime difference from the baseline. Blue line indicates measured time for the baseline processor.

6.1. Core scaling

We scale the BGQ core count (16) to see its performance impact. As shown in Figure 14, if we reduce the number of the cores to half, the runtimes of all applications are almost doubled. If we double the number of cores, the runtime reduction is between 27% (AMG) to 50% (LULESH). A further increase of core count will have diminishing performance return because the performance becomes more and more memory bound (by either memory latency or bandwidth) as shown in Figure 14. This means the memory resources including cache and bandwidth need to keep up to accommodate more cores.

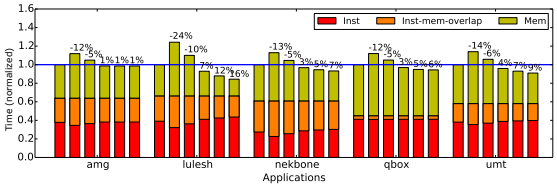
Overall, the core count is in a good balance with the rest of the architecture and leans towards the underdesign side. However, since the design is also constrained by chip area and power, increasing core count may not be possible. Note that an overdesign would mean changing (either increase or decrease) the core count does not affect performance much, and an underdesign would mean changing it would affect performance near linearly.

6.2. L1 cache scaling

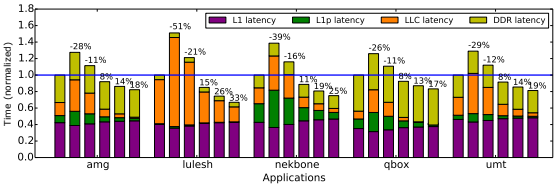
We scale the baseline L1 cache size (16 KB) to see its performance impacts. As shown in Figure 15b, the memory latency time is very sensitive to the L1 size; the larger the cache, the more L1 hits and L1's contribution to the total latency time, but the less LLC's contribution. We see diminished returns of increasing the L1 size, as LLC's contribution becomes less and less. The improved latency time translates to a overall runtime reduction of up to 12% for LULESH and 6% on average for all applications if we increase the L1 size by 4× (Figure 15a). AMG shows a minimum performance improvement with increased L1 size. Our investigation reveals that if we double the L1 size, the memory time of AMG becomes primarily bandwidth bound (Figure 16) and thus no longer affected by the L1 size. Overall, the designed L1 size is in a good balance with the rest of the architecture, and increasing it will also give considerable performance benefit.

6.3. Last level cache (LLC) scaling

Figure 17 shows how scaling BGQ's LLC (L2) size (1 MB/core) will impact the performance. For our applications, the LLC size seems overdesigned. Reducing the LLC size by



(a) total runtime.



(b) Memory latency time breakdown.

Figure 15: L1 cache scaling. For each application, the 6 columns represent the total execution time for scaling factor set (baseline, 0.25, 0.5, 2, 4, 8).

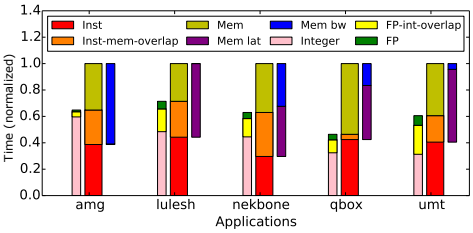
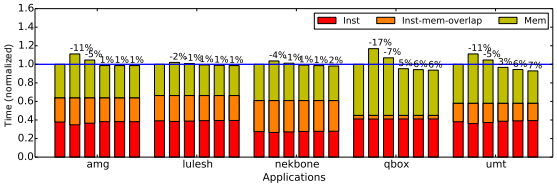


Figure 16: Time breakdown for increasing L1 by 2×.

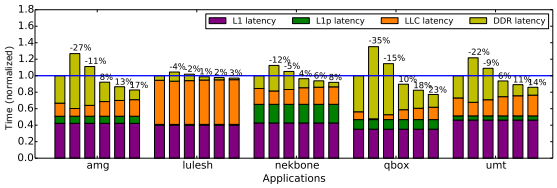
half only gives a small performance penalty of up to 7% for Qbox and 3.8% on average for all applications; doubling it only gives a slight performance increase between 1-5%.

6.4. Bandwidth scaling

Figure 18 shows how memory bandwidth scaling will impact BGQ's performance. If we decrease the bandwidth by half, the



(a) total runtime.



(b) Memory latency time breakdown.

Figure 17: LLC size scaling. For each application, the 6 columns represent the total execution time for scaling factor set (baseline, 0.25, 0.5, 2, 4, 8).

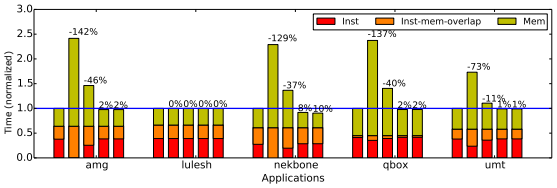


Figure 18: Bandwidth scaling. For each application, the 5 columns represent the total execution time for scaling factor set (baseline, 0.25, 0.5, 2, 4).

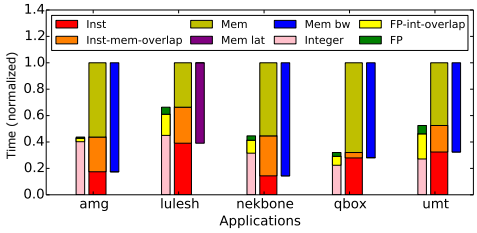


Figure 19: Time breakdown for decreasing bandwidth by half.

total runtime will increase significantly except LULESH, this is because the memory time changes to bandwidth bound from latency bound for all applications except LULESH as shown in Figure 19. On the other hand, doubling the bandwidth will not result in much performance benefit except with Nekbone, because the memory time of all applications except Nekbone is primarily bound by memory latency (Figure 12). The designed bandwidth is just in the right balance with the rest of the architecture.

6.5. Constraint-based exploration

We demonstrate our modeling framework in exploring architectural tradeoffs under chip area constraint. The hypothetical scenario for our study is: to develop a future processor based on BGQ with a 2× chip area (transistor) budget. How should we optimally allocate the chip area to cores and LLC? While a straightforward scaling option is to double both the cores and LLC, we also include other options which all keep the total chip area same (based on our measurement on the BGQ die photo; one BGQ core roughly takes the same area as 1 MB of LLC). For the future processor, we hypothetically increase the L1 cache size by 4× to 64 KB, and increase the memory bandwidth by 3× (anticipating new technologies such as stacked memory). We assume the higher L1 size and memory bandwidth do not significantly affect the chip area.

Figure 20 shows the predicted time for the different core count and LLC size options. The (56 cores, 8 MB LLC) option turns out to be the sweet spot for most applications, which is on average 14% better than the straightforward scaling option (32 cores, 32 MB LLC). This suggests that having more cores would benefit more than having more LLC. However, we should note that power is the main design constraint in today's processors, and having too many cores may exceed the design power envelope.

We have demonstrated our performance modeling framework for evaluating the design balance and exploring future scaling options for BGQ. In summary, the designed core count and L1 cache size are in good balance with the rest of the architecture and lean towards the underdesign side. The LLC size seems overdesigned. The bandwidth is balanced well with the rest of the architecture. For a future processor based on BGQ, more cores, together with correspondingly larger L1 cache and higher bandwidth, will continue to scale the performance, while the LLC size could be kept same or even shrink. Note that these recommendations should be taken with two caveats in mind: (1) they are specific to the selected applications of our interests, and (2) they should be considered along with the design constraints in chip area and power.

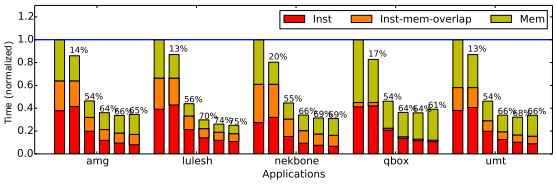


Figure 20: Core-LLC tradeoff exploration. For each application, the 5 columns represent the total execution time for (cores, LLC (MB)) set {(16, 16), (16, 48), (32, 32), (48, 16), (56, 8), (60, 4)}.

7. Related Work

In this work, we develop a performance modeling framework. By combining hardware counter-based performance measurements and analytical processor models, we reduce the process of architecture exploration to a matter of evaluating a set of mathematical formulas, which enables rapid and systematic search of processor design space for large programs and even full applications.

We see several works that are most similar in spirit to ours. Luo et al. [5, 33] use hardware counter data combined with analytical models to estimate the overlap of instruction execution and memory access for out-of-order processors. However, the focus of their work is to analyze the performance of existing processors, rather than exploring architecture for future processors. Saavedra and Smith [40] build machine and program execution models to estimate execution time for arbitrary machine/program combinations. Their technique relies heavily on static program analysis (assisted with runtime profiling) and thus has difficulties in taking into account compiler optimizations and dynamic program behaviors; in contrast, we use hardware counter data as inputs to our models so that these issues are easily and automatically handled. Carrington et al. [7, 43] build a modeling framework to predict application performance on future systems by combining simulation traces and machine profiles (collected by micro-benchmarks). In comparison, our approach relies on fast hardware counter-based profiling instead of several orders of magnitude slower program simulation. Because hardware counter data do not provide complete performance information, it requires more reasoning ability of our models to develop lower and upper

bounds of unknown factors. Krishna et al. [30] estimate upper performance bounds of applications through static program analysis. The advantages of their technique are the ease of use and not requiring runtime profiling. However, it is at the cost of not considering dynamic program behaviors; their technique also uses relatively simple hardware models.

There have been studies on analytical models of specific architecture features such as cache [2, 18, 45] and superscalar pipeline [13, 28]. Their focus is on the performance prediction of exact architecture features, while ours is on a methodology and an implemented framework to enable rapid, automated architecture search, as well as a demonstration for a set of applications on two recent processors.

Orthogonal and complementary to analytical modeling, simulation-based techniques have seen a great deal of recent developments. To speed up simulation (or reduce the number of simulations in architecture exploration), a variety of methods have been proposed including efficient parallelization [41], combined analytical modeling [6, 39], application abstraction [17], statistical sampling [46], and machine learning [26, 27, 29].

Domain-specific languages for modeling program behaviors are also developed [34, 35, 44]. Such languages require user-written code annotations or skeletons and need to be combined with hardware performance models to make a performance prediction. Other related works include very high-level bound-based Roofline performance modeling [47] and model-guided compiler and program optimizations [11, 12, 15, 36]. These studies require detailed, often manual analysis to model algorithm/program behaviors; they also have an application focus and use relatively simple processor models.

8. Software Release

Raexplore is implemented in Python (with plotting features using matplotlib). We will release Raexplore as an open-source tool. We have also developed a web application for it with features for users to manage and share architecture configurations and application profiles. Raexplore currently accepts hardware counter data from Intel VTune and IBM HPM, but is made modular and extensible for other profiling tools (e.g., PAPI [38]) or architecture simulators (e.g., M5 [4]), as well as integrating hardware models for other processors (e.g., GPU, DSP, ARM, and x86 multicores).

9. Conclusion

We have developed a novel performance modeling methodology that combines experimental and analytical approaches to overcome their shortcomings and gain the merits of both: fast, accurate, insightful. As a result, this hybrid approach is able to model, predict, and analyze the cross-architecture performance for full applications in little time. To our knowledge, such capability cannot be achieved by current technologies that are purely based on measurement, analytical modeling, or simulation, yet this capability is highly desired in co-designing next-generation manycore processors driven by a set of applications. The paper describes our first step proposing this methodology and focuses on trending power-efficient BGQ/Xeon Phi-style architectures for a set of HPC applications.

The great promise of analytical modeling is that it expresses the relation between performance and architectural components in mathematical formulas and thus allows a formal basis for computer architects to reason about and optimize architecture design. By combining with experimental evaluation, we made this approach practical for large applications. We envision our performance modeling framework could serve as a platform for future researchers to build a model library for a variety of both conventional and novel processors and to analytically and systematically compare their strengths and weaknesses targeting various applications. In addition to performance models, it would also be highly desired to combine them with chip area and power models and thus mathematically formulate the architecture design problem as a performance optimization problem with resource constraints.

Acknowledgment

We thank John Owens, Rong Ge, Andrey Vladimirov, and the anonymous reviewers for their comments and suggestions. We also thank James Collins for his careful editing of the manuscript. This work is supported by the U.S. Department of Energy under contract DE-AC02-06CH11357 and the LDRD X-PECT project 2013-213-NO. The research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] "The coral benchmarks," https://asc.llnl.gov/CORAL-benchmarks/.

[2] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM Trans. Comput. Syst.*, vol. 7, no. 2, pp. 184–215, May 1989.

[3] K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin, and J. Sancho, "Using performance modeling to design large-scale systems," *Computer*, vol. 42, no. 11, pp. 42–49, Nov 2009.

[4] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, Jul. 2006.

[5] K. W. Cameron, Y. Luo, and J. Scharzmeier, "Instruction-level microprocessor modeling of scientific applications," in *High Performance Computing*. Springer, 1999, pp. 29–40.

[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 52:1–52:12.

[7] L. Carrington, N. Wolter, A. Snavely, and C. B. Lee, "Applying an automated framework to produce accurate blind performance predictions of full-scale HPC applications," in *In Proceedings of the Department of Defense Users Group Conference*, 2004.

[8] J. Carter, L. Oliker, and J. Shalf, "Performance evaluation of scientific applications on modern parallel vector systems," in *Proceedings of the 7th International Conference on High Performance Computing for Computational Science*, 2007, pp. 490–503.

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 340–351.

[10] X. E. Chen and T. Aamodt, "Modeling cache contention and throughput of multiprogrammed manycore processors," *IEEE Trans. Comput.*, vol. 61, no. 7, pp. 913–927, Jul. 2012.

[11] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010, pp. 115–126.

[12] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Rev.*, vol. 51, no. 1, pp. 129–159, Feb. 2009.

[13] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 3:1–3:37, May 2009.

[14] P. Fischer, J. Kruse, J. Mullen, H. Tufo, J. Lottes, and S. Kerkemeier, "Nek5000–open source spectral element CFD solver," 2008, https://nek5000.mcs.anl.gov/.

[15] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the performance of an algebraic multigrid cycle on hpc platforms," in *Proceedings of the International Conference on Supercomputing*, 2011, pp. 172–181.

[16] K. Ganesan, L. John, V. Salapura, and J. Sexton, "A performance counter based workload characterization on Blue Gene/P," in *Proceedings of the 37th International Conference on Parallel Processing*, 2008, pp. 330–337.

[17] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *Proceedings of the 16th International Symposium onHigh Performance Computer Architecture (HPCA)*, Jan 2010, pp. 1–12.

[18] F. Guo and Y. Solihin, "An analytical model for cache replacement policy performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 1, pp. 228–239, Jun. 2006.

[19] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, Mar. 2012.

[20] A. Hartstein, V. Srinivasan, T. Puzak, and P. Emma, "On the nature of cache miss behavior: Is it $\sqrt{2}$?" *The Journal of Instruction-Level Parallelism*, vol. 10, pp. 1–22, 2008.

[21] IBM, "Hardware performance monitor (HPM) toolkit users guide," 2014.

[22] Intel, "Stream on Intel Xeon Phi coprocessors," 2013, http://software.intel.com/sites/default/files/article/370379/streamtriad-xeonphi-3.pdf.

[23] ——, "Intel Xeon Phi coprocessor system software developers guide," 2014.

[24] ——, "Intel Xeon Phi core micro-architecture," 2014, http://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture.

[25] ——, "VTune performance analyzer," 2014.

[26] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee, "An approach to performance prediction for parallel applications," in *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, 2005, pp. 196–205.

[27] E. Ipek, S. A. McKee, K. Singh, R. Caruana, B. R. d. Supinski, and M. Schulz, "Efficient architectural design space exploration via predictive modeling," *ACM Trans. Archit. Code Optim.*, vol. 4, no. 4, pp. 1:1–1:34, Jan. 2008.

[28] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," *SIGARCH Comput. Archit. News*, vol. 32, no. 2, pp. 338–, Mar. 2004.

[29] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra, "Using predictive modeling for cross-program design space exploration in multicore systems," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007, pp. 327–338.

[30] S. Krishna Narayanan, B. Norris, and P. D. Hovland, "Generating performance bounds from source code," in *Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, 2010, pp. 197–206.

[31] J. Krueger, D. Donofrio, J. Shalf, M. Mohiyuddin, S. Williams, L. Oliker, and F.-J. Pfreund, "Hardware/Software co-design for energy-efficient seismic modeling," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 73:1–73:12.

[32] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07, 2007, pp. 249–258.

[33] Y. Luo, O. M. Lubeck, H. Wasserman, F. Bassetti, and K. W. Cameron, "Development and validation of a hierarchical memory model incorporating CPU- and memory-operation overlap model," in *Proceedings of the 1st International Workshop on Software and Performance*, 1998, pp. 152–163.

[34] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, "GROPHECY: GPU performance projection from CPU code skeletons," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 14:1–14:11.

[35] J. Meng, V. A. Morozov, V. Vishwanath, and K. Kumaran, "Dataflow-driven GPU performance projection for multi-kernel transformations," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 82:1–82:11.

[36] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs," in *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*, Jun. 2009, pp. 256–265.

[37] V. Morozov, K. Kumaran, V. Vishwanath, J. Meng, and M. Papka, "Early experience on the Blue Gene/Q supercomputing system," in *IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, May 2013, pp. 1229–1240.

[38] PAPI team at the University of Tennessee Knoxville, "PAPI: Performance application programming interface," 2014, http://icl.cs.utk.edu/papi.

[39] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero, "On the simulation of large-scale architectures using multiple application abstraction levels," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 36:1–36:20, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2086696.2086715

[40] R. H. Saavedra and A. J. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," *ACM Trans. Comput. Syst.*, vol. 14, no. 4, pp. 344–384, Nov. 1996.

[41] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 475–486.

[42] J. Shalf, D. Quinlan, and C. Janssen, "Rethinking hardware-software codesign for exascale systems," *Computer*, vol. 44, no. 11, pp. 22–30, 2011.

[43] A. Snavely, N. Wolter, and L. Carrington, "Modeling application performance by convolving machine signatures with application profiles," in *Proceedings of the IEEE International Workshop on the Workload Characterization*, 2001, pp. 149–156.

[44] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 84:1–84:11.

[45] G. Sun, C. J. Hughes, C. Kim, J. Zhao, C. Xu, Y. Xie, and Y.-K. Chen, "Moguls: A model to explore the memory hierarchy for bandwidth improvements," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 377–388.

[46] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, Jul. 2006.

[47] 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.