Compute Bottlenecks on the New 64-bit ARM

Adam Jundt[*], Allyson Cauble-Chantrenne[*], Ananta Tiwari[*†], Joshua Peraza[*], Michael A. Laurenzano[*§], Laura Carrington[*†]

[*] EP Analytics, Inc.
{adam.jundt, allysonc, ananta.tiwari, joshua.peraza, michaell, laura.carrington}@epanalytics.com

[†]Performance Modeling and Characterization Laboratory, San Diego Supercomputer Center

[§]University of Michigan

ABSTRACT

The trifecta of power, performance and programmability has spurred significant interest in the 64-bit ARMv8 platform. These new systems provide energy efficiency, a traditional CPU programming model, and the potential of high performance when enough cores are thrown at the problem. However, it remains unclear how well the ARM architecture will work as a design point for the High Performance Computing market. In this paper, we characterize and investigate the key architectural factors that impact power and performance on a current ARMv8 offering (X-Gene 1) and Intel's Sandy Bridge processor. Using Principal Component Analysis, multiple linear regression models, and variable importance analysis we conclude that the CPU frontend has the biggest impact on performance on both the X-Gene and Sandy Bridge processors.

Keywords

ARMv8, X-Gene, Sandy Bridge, power, performance, modeling

1. INTRODUCTION

System designers and HPC centers are at a crossroads on how to build the next generation Exascale systems. In order to deliver Exaflop performance, they'll need systems that are capable of high performance, require minimal energy, and provide a known programming environment. The trifecta of power, performance, and programmability has remained elusive. Currently, most systems are able to hit two of the targets, e.g. focusing on performance and programmability (a traditional, scaled up x86 system) or performance and power (an accelerator based system) [22].

A potential third option is now hitting the market. ARM vendors are looking into getting their shot in the HPC community, and with improved double-precision and SIMD support many customers have been drawn in by the potential of a low power system with ease of programmability. Systems like HP's Moonshot [17], which features APM's X-Gene 1 processors, and Cavium [8] are now available. Yet it remains unclear if ARMv8 will serve as a useful design point for the HPC community and can provide the crucial element of performance as well. To that end, we study what architectural elements dictate power and performance behaviors on the first commercial implementation of ARMv8, an Applied Micro X-Gene 1. We also present the same analysis for an Intel Xeon E5-2670v1 (Sandy Bridge), a popular design point found in many large-scale HPC systems today. Our study utilizes a set of over 300 test cases that include micro-benchmarks, application benchmarks, and production-level large-scale HPC applications. These test cases are drawn from different computational domains and are meant to cover the types of calculations that are prevalent in large-scale computing workloads.

E2SC2015, November 15-20, 2015, Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-3994-0/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2834800.2834806

We characterize each of the test cases using performance hardware counters. These counters measure important micro-architectural events that report the application's interaction with key architectural elements. For example, the number of load and store operations can be used to quantify an application's memory activity, while the number of branch operations can be used to assess an application's usage of the branch prediction unit in the CPU. Usually, the number of performance counters exposed to the ISA is in the hundreds and not all counters correlate to the power and performance aspects of the system. In addition, some counters measure the same phenomenon (e.g., L2 cache misses counter and L3 cache accesses counter). In order to focus our attention on only the relevant counters (and therefore, relevant architectural elements), we use a combination of dimension reduction techniques and statistical machine learning techniques. Dimension reduction is achieved using Principal Component Analysis (PCA).

The reduced set of predictors is supplied to a machine learning algorithm, which builds models that predict an application's power and performance based off of hardware counter events. The use of a statistical machine learning model is motivated by the fact that often these modeling techniques support calculating the relative influence of the input metrics on the predicted values. We use this feature to discover which performance counters affect the predicted values the most and use that information as the basis for determining what architectural components have the greatest effect on performance and power. To simplify the presentation of the final analysis, we categorize the reduced set of hardware counters to classes of our own devising. For example, we group events that relate to the reservation station, instruction fetch/decode and frontend stalls into the *frontend* group (more details appear in Section 3.3).

Our results show that the CPU frontend has the biggest impact on performance on both the ARMv8 X-Gene and Sandy Bridge processors. The frontend and memory usage split the importance of dynamic power usage on the X-Gene; on the Sandy Bridge architecture, cache related activities have the biggest impact on power.

The rest of the paper follows: in Section 2 we describe the setup for our platforms, applications, and measurement techniques. Section 3 reports our method for modeling and discovering the power and performance factors on the two systems, and in Section 4 we present our findings. Related work is presented in Section 5 along with the conclusion in Section 6.

2. METHODOLOGY

We select a large set of computational kernels, benchmarks, and mini applications from diverse computational domains for the analysis on two platforms. Each platform is instrumented to measure the power draw for different computations as well as capture the characteristics of the computation to be used in the model development. The specific platforms, applications, and measurement tools and techniques used for data collection are described in this section.

2.1 Platforms

In this work, we use an 8-core Intel Xeon E5-2670v1 (Sandy Bridge) [18], a processor characteristic of the typical processor found in HPC compute nodes at the time of this writing (it appears in 5 of the top 15 systems in the current top500 list [3]). We turn off Turbo-Boost and hyper-threading on the Sandy Bridge processor. We also make use of the first generation 64-bit ARMv8 implementation, the recently released Applied Micro 883208-X1 [4], consisting of an 8-core X-Gene processor. A detailed listing of the configurations of the processors is presented in Table 1.

2.2 Applications

We use 303 unique configurations of diverse applications, benchmarks, and kernels. Workloads typical to high performance computing (NAS Parallel Benchmarks (NPBs) [5], polybench [21], Mantevo [23], Trinity [24] and Coral [2]), computer architecture (applications from SPEC CPU2006 [25] and PARSEC [7]), and high performance embedded computing (HPEC) (Coremark [9] and the DARPA PERFECT benchmarks [6]) are selected to be representative of high performance applications. Where applicable, we evaluate both single precision and double precision calculations. The input configurations for each application are summarized in Table 2.

2.3 Power and Performance Measurement

To measure power on the ARMv8, we collect cartridge-level power from the internal integrated Lights Out (iLO) facilities [35]. The cartridge-level power is only updated every 15 seconds, however, so we run each application in a loop for at least 160 seconds, removing the first and last 20 seconds of measurements before averaging. This ensures that we only measure application power. To measure power on the Sandy Bridge, we use a WattsUp meter [1] to collect wall power, and each application is run in a

TABLE 1: SYSTEM DETAILS FOR SANDY BRIDGE AND X-GENE

	Intel Sandy Bridge	**Applied Micro X-Gene 1**
Instruction Set Architecture	X86_64 (64-bit)	ARMv8 (64-bit)
ISA Design	CISC	RISC
Server	Fully Integrated	SOC
CPU	Intel E5-2670v1 Sandy Bridge	APM 883208-X1 X-Gene
Data Cache (n-way shared)	32KB L1(1) 256KB L2(1) 20MB L3(8)	32KB L1(1) 256KB L2(2) 8MB L3(8)
Instruction Cache (n-way shared)	32KB L1(1)	32KB L1(1)
Memory	32GB 1600 MHz	64GB 1600 MHz
FP/Vector Support	AVX (256-bit)	Neon (128-bit)

loop for at least 60 seconds. In order to separate out anything besides the dynamic CPU power (e.g. motherboards, fans, I/O devices, etc.) we subtract out idle server power from the application's power measurements. Runtime is measured as the end-to-end execution time of the application and all processes are pinned to cores to prevent process migration. Performance and power measurements are averaged over 3 runs.

2.4 Hardware Counters

In order to discover key architectural features that impact power and performance on the two architectures, we collect (in separate runs from the power/performance measurement runs) every hardware performance monitor available through the Linux perf interface for each executed kernel or benchmark. Each system has a different group of available performance monitors; there are 403 monitors available on the Sandy Bridge and 127 on the ARMv8. These performance counters show detailed characteristics about architectural features including on- and off-chip caches, branch predictions and outcomes, CPU frontend and backend, TLB, power management settings, on-chip network and memory controller.

3. MODELS

A high level overview of our analysis for discovering the key architectural features affecting performance and power on the two test bed systems is presented in Figure 1. Hardware counter data collected for all applications, benchmarks, and kernels are first processed to eliminate counters that measure very low frequency hardware events. The hardware counter data dimension is then further reduced using combined PCA and clustering analysis (Section 3.1). The remaining hardware counters are used to build statistical models to rank each counter's impact on power and

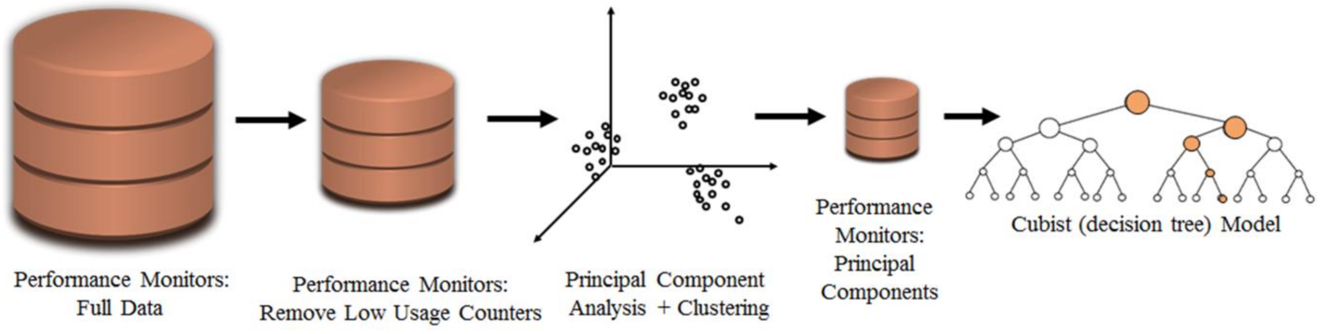
Performance Monitors: Full Data
Performance Monitors: Remove Low Usage Counters
Principal Component Analysis + Clustering
Performance Monitors: Principal Components
Cubist (decision tree) Model

FIGURE 1: MODEL STEPS [15]

TABLE 2: DESCRIPTIONS OF WORKLOADS

Domain	Suite	Applications	Input/Configuration
HPC	NAS Parallel Benchmarks	block tridiagonal, conjugate gradient, embarrassingly parallel, fourier transform, integer sort, multigrid, lower-upper gauss-seidel, scalar penta-diagonal	A, B, and C
	PolyBench	adi^2, atax^2, bicg^2, cholesky^2, covcol^1, dct^2, doitgen^2, dsyr2k^2, dsyrk^2, dynprog^2, fdtd2d^2, fdtdapml^2, gemver^2, gesummv^2, gramschmidt^1, jacobi2dimper^2, matmulinit^1, mm^1, mvt^2, seidel^2, ssymm^2, stencil3d^2, strmm^2, strsm^2, swim^2, tce^2, tmm^2, trisolv^2	SP and DP; ^{1}L1, L2, and L3 ^{2}L1, L2, L3, and MM
	Mantevo	CoMD, miniMD	SP and DP ; 2000 steps
	Trinity	GTC^1, MiniFE^2, MiniGhost^3, stream^1	SP and DP; ^{1}default ^{2}120x120x120 ^{3}500 steps
	Coral	AMGmk, MILCmk	default; SP and DP
Arch.	SPEC CPU2006	perlbench, bzip2, gcc, bwaves, games, milc, zeusmp, cactusADM, leslie3d, namd, gobmk, soplex, calculix, hmmer, sjeng, GemsFDTD, h264ref, tonto, omnetpp, astar, xalancbmk	reference
	Parsec	blackscholes, bodytrack, facesim, ferret, fluidanimate, swaptions	native
HPEC	coremark	matrix, linked list, state machine, cyclic redundancy check	size=666; iter=2e5
	PERFECT	SAR: pfa-interp1^6,pfa-interp2^6, bp^4 PA1: dwt53^5, 2d convolution^5, histogram equalization^5 STAP: outer product^6, system solve^6, inner product^6 WAMI: lucas kanade^6, debayer^6, change detection^6 OTHER: 1d fft^6, 2d fft^4, sort^6	^{4}small ^{5}medium ^{6}large

performance (Section 3.2). We then group the hardware counters into related bins, e.g. memory, branch, etc., to help identify the key architectural performance and power features on each system (Section 3.3).

3.1 Model Input

All of the hardware counters, power, and performance data described in Section 2 are input for the models. The model outcome is performance or power. In order to reduce the number of predictors to determine which architectural features contribute to performance and power, counters that measure rare hardware events (defined as events that occur less than once per thousand instructions averaged overall all 303 applications) are removed from further consideration. We also remove any variables that directly imply performance, like the number of CPU cycles. This reduces the number of performance counters from 403 to 230 on the Sandy Bridge and 127 to 58 on the X-Gene.

To further reduce the dimensionality of the hardware counter data, we make use of Principal Component Analysis (PCA) [10, 33], a statistical method that reduces sets of variables that may be correlated into a smaller number of non-correlated, *principal components*. Used effectively, PCA can greatly reduce the number of input variables with little to no loss to the quality of the resulting model. Similar to the methods found in [15], we make use of PCA and K-means clustering [11, 12] to discover the principal components from the remaining hardware counters after a first round data filtering. Our method first selects N principal components that are used to create an N-element vector for each of the input variables. Each vector is a linear combination of principal components that make up the input variable. Next, we use K-means clustering to discover clusters of input variables that have similar principal components. We then select a single input variable from each of the clusters as a representative of that group and discard all other non-selected input variables. This reduces the number of input variables that are fed into the models to 78 on the Sandy Bridge and 21 on the X-Gene.

3.2 Model Creation and Variable Importance

Using the reduced set of counters as input, we train machine learning models to predict performance and power. Model problem formulation for the performance model is provided in Equation 1.

$$Performance = f(C_1, C_2, \cdots, C_n) \qquad (1)$$

In Equation 1, C_1, C_2 ... C_n are performance counters. In this paper, we use the Cubist [14] model to approximate function *f*.

A Cubist model consists of a tree of linear regression models; final predictions are made using linear models at the terminal nodes of the tree. The path to the terminal node is determined by rules in the intermediate nodes, which are also based on linear models built using predictors used in previous steps. Compared to other machine learning approaches such as neural nets, Cubist models are easier to interpret and have the capability to encapsulate complex non-linear relationships between input variables.

From the resulting Cubist model, we can rank how important each variable is by how often it used in the model.

3.3 Generalizing Hardware Counters

To simplify the presentation of our results in terms of the architectural units rather than raw counter descriptions, we develop a counter classification scheme. In total, we have nine bins and provide brief descriptions for each bin below.

BRANCH (BNCH): events related to counts of branch instructions and branch prediction outcomes, such as the number of conditional branches or the number branch mispredictions.

CACHE: events relating to the cache, including access, hit, and miss counts.

FP+SIMD (FP): floating-point events, including single- and double-precision operations, as well as vectorized (SIMD) and non-vectorized operations.

FRONTEND (FE): CPU frontend events, including those touching on the reservation station, instruction fetch, instruction decode, and frontend stalls.

INSNMIX: events that indicate the mix of instructions being executed, such as dispatched micro-ops.

MEMORY (MEM): events relating to memory, including the number of loads and stores as well as any related stalls, such as resource stalls caused by a full store buffer.

PWR STATE (PWR): cycle counts during different CPU power states (Sandy Bridge only).

RESOURCE STALLS (RESSTL): Backend Stalls [32] (ARM only).

TLB: Translation Lookaside Buffer (TLB) events, including access, hit and miss counts.

3.4 Overfitting Avoidance

One of the key hazards of using machine learning techniques is overfitting. Overfitted models have no predictive value for anything beyond the training set that was used to develop the models. To avoid overfitting, we use the *k*-fold cross-validation method. In this method, the training dataset is randomly partitioned into *k* subsets. *k* models are then created, each of which uses *(k-1)* subsets to train the model and the held out subset to test the model. The model with the minimum error is selected and used as the final model.

To further ensure that the predictive capability generalizes to cases that are not in the training set, we use "out-of-sample" model validation by randomly dividing up our test cases into non-overlapping training and test sets. We take the additional step to ensure the applications with multiple configurations/input sets are in either the training subset or test subset, but not both. This ensures that the model's accuracy is not influenced by the prior knowledge of mostly identical benchmark with a different memory footprint. The model is then trained on the training subset and validated on the test set.

4. ARCHITECTURAL BOTTLENECKS

In this section, we present how well Cubist models predict the performance (Section 4.1) and power draw (Section 4.2) of an application given a set of hardware counters (Figures 2A and 2B) and the impact of the architectural features on performance and power (Figures 3 and 4).

4.1 Performance Results

We train our models on 75% of the applications and test on the remaining 25%. The models are trained and tested on the same 75/25 break-down across architectures to ensure fairness in model creation and validation. We normalize the performance by the number of billions of instructions executed. The resulting median error is 7.36% and 1.26% for the ARM and Sandy Bridge performance models respectively (Figure 2A).

The importance of each architectural feature on performance on the ARMv8 can be found in Figure 3A. Two features stand out on the ARMv8 system: the frontend and the branch unit. Looking more closely at the counters in the frontend bin, we find that they describe how often units are not given instructions, which dictates how well the application uses the rest of the CPU. For example, applications that rarely utilize the floating-point/SIMD unit would produce high counts for the hardware counter that counts the number of cycles the floating-point unit is not issued an instruction. Meanwhile, the branch unit contains counters that deal with branch instructions and branch prediction. The performance of the application is dependent on the branch prediction accuracy and the misprediction cost.

Figure 3B shows the variable importance ranking for the Sandy Bridge architecture. Performance is dominated by the frontend for the Sandy Bridge as well. Many of the Sandy Bridge counters in the frontend bin are stalls, like full reservation stations which can hurt performance because the backend is not executing instructions fast enough to keep the pipeline moving.

The BNCH bin, which consists of counters that describe interaction of applications with the branch unit, ranks in the top three in both architectures. To understand why the branch unit is a bottleneck, we created a benchmark, similar to one created in [31], that loops around a branch which is controlled by an array of booleans. We initialize the size of the array to 32K booleans, and run the benchmark inside another loop 16K times. The pseudocode is presented below:

```
initialize array of booleans, DIRECTIONS
do 16K times:
  for each element in DIRECTIONS:
     if true:
       increment counter 1
     else:
       increment counter 2
```

We can test the accuracy of the branch predictor on each architecture by playing with the initialization of our array and collecting the number of branch instructions and mispredictions

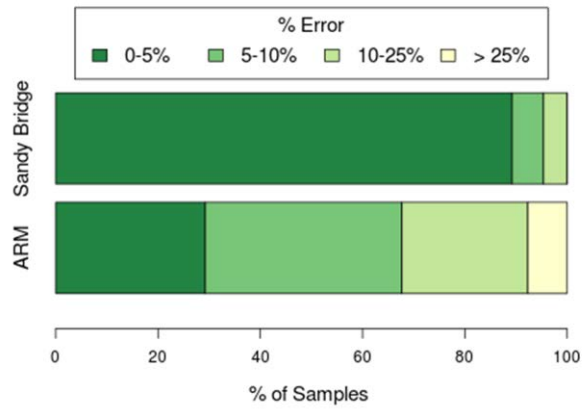

FIGURE 2A: PERFORMANCE MODEL RESULTS

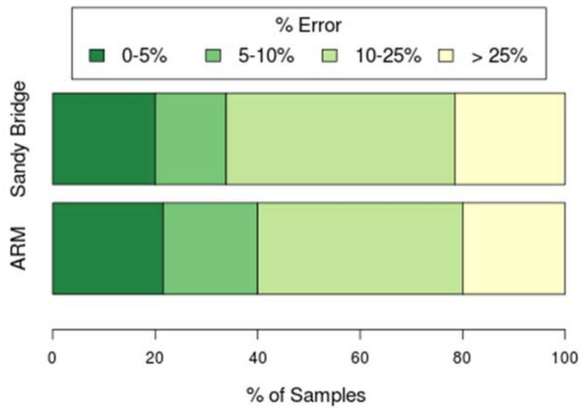

FIGURE 2B: POWER MODEL RESULTS

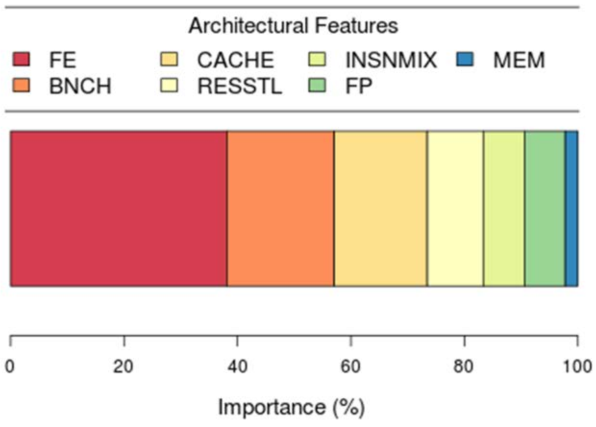

FIGURE 3A. ARM PERFORMANCE VARIABLE IMPORTANCE

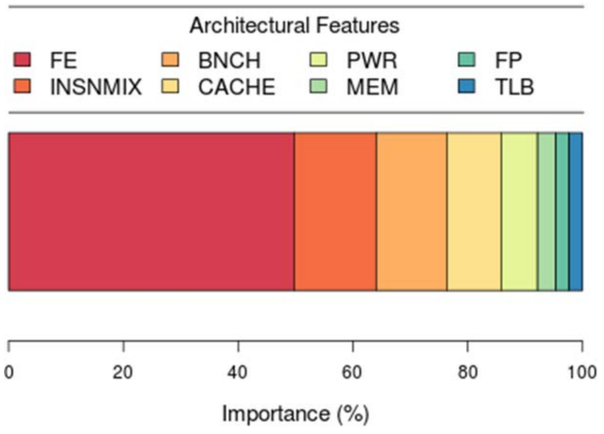

FIGURE 3B. SANDY BRIDGE PERFORMANCE VARIABLE IMPORTANCE

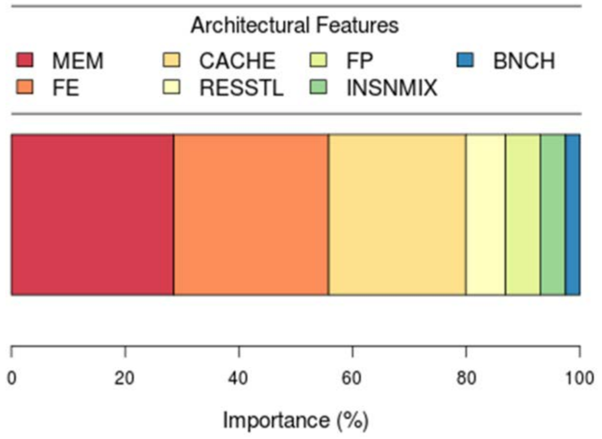

FIGURE 4A. ARM VARIABLE IMPORTANCE

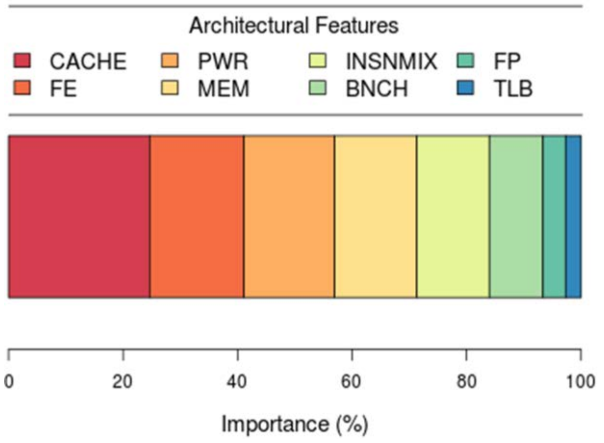

FIGURE 4B. SANDY BRIDGE POWER VARIABLE IMPORTANCE

and the number of total cycles. To get the misprediction rate, we divide the number of mispredictions by the number of branches.

If we initialize all the values of the directions array to TRUE, so that the branch is never taken, we expect the misprediction rate to be 0%. When we run this on both architectures, we find that both are able to do this. If we randomly generate the array, the branch predictor should have a 50% chance at guessing the correct direction. This is true on the ARMv8, but interestingly the Sandy Bridge does better than random chance, resulting in a misprediction rate of 46%.

To understand why, we increase the directions array size to 64K and repeat the experiment. As before, an array of TRUE values has a misprediction rate of 0% on both architectures. This time, however, the randomly-generated array mispredicts 50% of the time for both the ARMv8 and the Sandy Bridge. Branch predictors are designed to recognize certain patterns, and in the case of this benchmark, the same pattern is repeated 16K times. If the pattern is short enough, it can "remember" it and predict better than random. In the case of the Sandy Bridge, it takes an array of 64K booleans to break the branch predictor. Our results are summarized in Table 3.

Our benchmark can also us an estimate of the misprediction cost. Since initializing the boolean to hold only true values has a misprediction rate of 0%, it represents a "perfect" run of the benchmark. We calculate the average cycle cost of a branch misprediction with the following formula:

$$cost = \frac{(cycles_{random} - cycles_{true})}{mispredictions_{random}} \quad (2)$$

Where $cycles_{random}$ is the number of cycles the benchmark takes when the directions array is initialized randomly, $cycles_{true}$ is the number of cycles the benchmark takes when the directions array is initialized to all true values, and $mispredictions_{random}$ is the number of mispredictions that occur when the directions array is randomly initialized. The ARMv8 and Sandy Bridge have average misprediction costs of 26 and 23 cycles respectively.

In all, the ARMv8's branch predictor appears to be less accurate than the Sandy Bridge and has a higher misprediction cost. As such, the branch unit can be more of a bottleneck on the ARMv8

TABLE 3: MISPREDICTION RATES (LOWER IS BETTER)

Array Values	**Array Size**	**ARMv8**	**Sandy Bridge**
All True	32K	0%	0%
All True	64K	0%	0%
Random	32K	50%	46%
Random	64K	50%	50%

than on the Sandy Bridge.

4.2 Power Results

We select the same 75/25 training and test set breakdown for the power models. The resulting models have a median error of 13.92% for the ARMv8 and 15.4% for the Sandy Bridge. Error rates are shown in Figure 2B.

The important characteristics are a little different for the dynamic power models. Memory in particular is a bin that is not relatively important for performance but is the most important feature in dynamic power on the ARMv8. Looking more closely at the counters in this bin, we find that it is dominated by accesses to the memory bus. This is interesting because memory access costs are generally very high, so the ARMv8 must either be good at hiding the latency, or the costs are relatively unimportant when compared to the frontend and the branch unit. Otherwise, it should be just as important to performance. The use of the memory bus, though, implies reads and writes to memory, which could have two effects on power. The first is that a data transfer consumes power, so the more reads and writes to the memory bus, the more power is consumed. The second effect is that memory could have a power-saving state for when it is idle. Reads and writes to memory would mean that memory would not be able to be in the power-saving state.

One way to compare the effect of power that memory has on the Sandy Bridge versus the X-Gene is to run LMbench [30], a benchmark for determining memory bandwidth. Running this on both systems we find that the per-core memory read bandwidth is 1.7X smaller on the ARMv8. This could imply that memory needs to be active 1.7X longer than Sandy Bridge for transferring the same amount of data, which would increase the power-usage.

The frontend is still an important feature as well. As discussed in the previous section, this bin contains counters that describe which parts of the system are not being given work. One possible explanation for this is the hardware components could consume more power while actively being used, which indicates that applications that are able to keep all pipelines active would consume more power than those that stall [34].

On the Sandy Bridge, the cache is the most important factor in dynamic power. The cache will consume power by doing data transfers. Since it does not seem to have as much of an impact on performance, we guess that costs of cache misses are hidden by the hardware. One way this could happen is if data is prefetched before the miss can even happen. How often data needs to be transferred throughout the cache hierarchy due to prefetching will affect how much power is consumed by the cache, but will have less of an effect on performance, assuming the prefetcher uses a good algorithm for the set of applications.

5. RELATED WORK

Given the recent attention of ARM in the server and scientific communities, there has been a growth in research on ARM's power and performance capabilities [16, 26, 27, 28, 29, 31].

Padoin, et al. [1] compare the performance and energy trade-offs between a 32-bit ARM processor and Sandy Bridge processor on the NAS Parallel Benchmarks. The authors report that while the ARM processor requires far less power, it is not more energy efficient than the Sandy Bridge processor due to its lower performance.

Abdurachmanov, et al. [19] conducted a usability, performance, and power study of the ARMv8, Xeon Phi, and Intel Xeon processors for software specific to interest to CERN (CMSSW). Using all available threads on the reported systems, they find that the ARMv8 had the lowest performance, used the lowest amount of energy, but was not more energy efficient that the Xeon.

Laurenzano, et al. [20] reported the performance and energy efficiency of the ARMv7 architecture on a range of HPC computational benchmarks. They find that the key bottlenecks of the architecture came from FP/SIMD computations and interactions with the memory subsystem.

Our work differs from these earlier studies in several ways. We focus our attention on the latest available ARM platform, the 64-bit ARMv8. We also use statistical modeling to isolate the key architectural elements of the up and coming ARM system and the established Sandy Bridge processor, allowing for the potential of hardware designers and programmers to consider methods to increase performance and energy efficiency.

6. CONCLUSION

The recent release of the first commercially available implementation of ARMv8, Applied Micro's X-Gene 1, has shown promise already with its improved double precision and SIMD support. By utilizing over 300 test cases that span workloads from multiple computational domains typical to HPC and collecting performance hardware counters from each test, we create models to discover which architectural elements dictate power and performance behaviors on the X-Gene and Sandy Bridge processors.

Our results show that the CPU frontend and branch predictor has the biggest impact on performance on the ARMv8 X-Gene processor, where the frontend and cache dictate performance on the Sandy Bridge. Conversely, the memory and frontend are the key architectural factors for energy usage on the X-Gene compared to the Sandy Bridge where the cache dominates power. Advancements in the compiler and software stacks for the ARM 64-bit architecture, which have only had a short time to evolve, will mitigate the effects of some of the performance bottlenecking components. Further, aggressive technology scaling and optimization of circuits that will likely be featured in other implementations of the ARMv8 architecture should help alleviate some of the bottlenecks as well.

Acknowledgements

This work was supported in part by the Air Force Office of Scientific Research under AFOSR Award No. FA9550-12-1-0476 and by DoE SBIR Award No. DE-SC0013164.

7. REFERENCES

[1] E. L. Padoin, L. L. Pilla, M. Castro, F. Z. Boito, P. O. A. Navaux, and J.-F. Mehaut. Performance/energy trade-off in scientific computing: the case of arm big. little and intel sandy bridge. IET Computers & Digital Techniques (CDT), 2014.

[2] CORAL Benchmark Codes. https://asc.llnl.gov/CORAL-benchmarks/, 2013. [Online; accessed 31-August-2015].

[3] Top 500 Supercomputing Sites. http://top500.org/, 2014. [Online; accessed 31-August-2015].

[4] AppliedMicro. X-Gene. https://www.apm.com/products/data-center/x-gene-family/x-gene/, 2015. [Online; accessed 31-August-2015].

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The nas parallel benchmarks. International Journal of High Performance Computing Applications, 1991.

[6] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo. PERFECT (Power Efficiency Revolution For Embedded Computing Technologies)

Benchmark Suite Manual. Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013. http://hpc.pnnl.gov/projects/PERFECT/.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In Parallel Architectures and Compilation Techniques (PACT), 2008.

[8] Cavium. ThunderX ARM Processors. http://www.cavium.com/ ThunderX_ARM_Processors.html, 2015. [Online; accessed 31-August -2015].

[9] E. M. B. C. (EEMBC). CoreMark: an EEMBC Benchmark. https://www.eembc.org/coremark/, 2015. [Online; accessed 31-August-2015].

[10] I. Jolliffe. Principal component analysis. John Wiley & Sons, Ltd, 2002.

[11] R. O. Duda and P. E. Hart. Pattern classification and scene analysis. Vol. 3. New York: Wiley, 1973.

[12] C. Ding and X. He. K-means clustering via principal component analysis. Proceedings of the twenty-first international conference on Machine learning. ACM, 2004.

[13] N. R. Draper, H. Smith, and E. Pownell. Applied regression analysis. Vol. 3. New York: Wiley, 1966.

[14] RuleQuest Research. Data mining with cubist. http://rulequest.com/ cubist-info.html, 2012. [Online; accessed 31-August-2015].

[15] L. Porter, et al. Making the Most of SMT in HPC: System-and Application-Level Perspectives. ACM Transactions on Architecture and Code Optimization (TACO) 11.4 (2015): 59.

[16] P. Stanley-Marbell and V. C. Cabezas. Performance, power, and thermal analysis of low-power processors for scale-out systems. In Workshop on High-Performance, Power-Aware Computing (HPPAC), 2011.

[17] N. Hemsoth. Moonshot Moves HPC Closer to ARM's Reach. http://www.hpcwire.com/2014/09/29/moonshot-moves-hpc-closer-arm-reach/, 2014. [Online; accessed 31-August-2015].

[18] Intel. Intel Xeon Processor E5-2670. http://intel.ly/1jjxoQE, 2012. [Online; accessed 31-August-2015].

[19] D. Abdurachmanov, B. Bockelman, P. Elmer, G. Eulisse, R. Knight, and S. Muzaffar. Heterogeneous high throughput scientific computing with apm x-gene and intel xeon phi. Journal of Physics: Conference Series. Vol. 608. No. 1. IOP Publishing, 2015.

[20] M. A. Laurenzano, A. Tiwari, A. Jundt, J. Peraza, W. A. Ward Jr, R. Campbell, and L. Carrington. Characterizing the performance-energy tradeoff of small arm cores in hpc computation. In European Conference on Parallel Processing (Euro-par), 2014.

[21] L.-N. Pouchet. The Polyhedral Benchmark Suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench/, 2012. [Online; accessed 10-April-2015].

[22] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In International Symposium on Computer Architecture (ISCA), 2012.

[23] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Sandia National Laboratories Technical Report, 2009.

[24] M. Cordery, B. Austin, H. Wassermann, C. Daley, N. Wright, S. Hammond, and D. Doerfler. Analysis of cray xc30 performance using trinity-nersc-8 benchmarks and comparison with cray xe6 and ibm bg/q. In Workshop on High Performance Computing Systems: Performance Modeling, Benchmarking and Simulation (PMBS). 2014.

[25] J. L. Henning. Spec cpu2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, 2006.

[26] M. F. Cloutier, C. Paradis, and V. M. Weaver. Design and analysis of a 32-bit embedded high-performance cluster optimized for energy and performance. In Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014.

[27] M. Jarus, S. Varrette, A. Oleksiak, and P. Bouvry. Performance evaluation and energy efficiency of high-density hpc platforms based on intel, amd and arm processors. In Energy Efficiency in Large Scale Distributed Systems (EE-LSDS). 2013.

[28] Z. Ou, B. Pang, Y. Deng, J. K. Nurminen, A. Yla-Jaaski, and P. Hui. Energy-and cost-efficiency analysis of arm-based clusters. In Cluster, Cloud and Grid Computing (CCGrid), 2012.

[29] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez. Tibidabo: Making the case for an arm-based hpc system. Future Generation Computer Systems, 2014.

[30] LMbench. http://sourceforge.net/projects/lmbench/, 2012. [Online; accessed 31-August-2015].

[31] A. Tiwari, K. Keipert, A. Jundt, J. Peraza, S. Leang, M. Laurenzano, M. Gordon, and L. Carrington. Performance and Energy efficiency analysis of 64-bit arm using gamess. In Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2015. To Appear.

[32] ARM X Gene Events. http://bit.ly/1LmDdXQ, 2015. [Online; accessed 31-August-2015].

[33] C. Croux, P. Filzmoser, and M. Oliveira. Projection-pursuit estimators for robust principal component analysis. Technical Report TS-04-4, Vienna University of Technology, Austria, 2004.

[34] A. Baniasadi and A. Moshovos. Instruction flow-based front-end throttling for power-aware high-performance processors. In Proceedings of the international symposium on Low power electronics and design (ISLPED), 2001.

[35] Server remote management with HP Integrated Lights Out (iLO), 2015. http://bit.ly/1NOjMsC. [Online; accessed 31-August-2015].

[36] WattsUp? Meters. https://www.wattsupmeters.com.