# Long and Winding Road Towards Efficient High-Performance Computing

<table>
<thead>
<tr>
<th>Journal:</th>
<th><em>Proceedings of the IEEE</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Manuscript ID</td>
<td>0125-SIP-2017-PIEEE.R1</td>
</tr>
<tr>
<td>Manuscript Categories:</td>
<td>Special Issue Paper</td>
</tr>
<tr>
<td>Date Submitted by the Author:</td>
<td>n/a</td>
</tr>
<tr>
<td>Complete List of Authors:</td>
<td>JALBY, William; Universite de Versailles Saint-Quentin en Yvelines UFR des Sciences, Computer Science; Exascale Computing Research, LI PARAD KUCK, David; INTEL, SSG MALONY, Allen; University of Oregon, Computer Science MASCELLA, Michel; CEA, DRF/DSV MAZOUZ, Abdelhafid; BULL/ATOS, HPC POPOV, Mihail; Universite de Versailles Saint-Quentin en Yvelines UFR des Sciences, Computer Science</td>
</tr>
<tr>
<td>Keyword:</td>
<td>Computer architecture, Computer performance, Design methodology, Parallel architectures, Power measurement</td>
</tr>
</tbody>
</table>
Long and Winding Road Towards Efficient High-Performance Computing

William Jalby, David Kuck, Allen D. Malony, Michel Masella, Abdelhafid Mazouz, and Mihail Popov

Abstract—The major challenge to Exaﬁlop computing, and more generally, efﬁcient high-end computing, is in ﬁnding the best ”matches” between advanced hardware capabilities and the software used to program applications, so that top performance will be achieved. Several benchmarks show very disappointing performance progress over the last decade, clearly indicating a mismatch between hardware and software. To remedy this problem, it is important that key performance enablers at the software level – autotuning, performance analysis tools, full application optimization – are understood. For each area, we highlight major limitations and most promising approaches to reaching better performance and energy levels. Finally, we conclude by analyzing hardware and software design, trying to pave the way for more tightly integrated hardware and software co-design.

Index Terms—Autotuning; performance evaluation tools; molecular dynamics; hardware design; benchmarking

I. INTRODUCTION

Peak performance progress has been steady and impressive during the last 20 years, giving hope for reaching the exaﬁlop level within the next decade. According to the TOP 500 ranking, which twice a year lists the most powerful systems, the fastest system reached 93 Petaflops running the LINPACK benchmark in November 2017 [1]. What does it mean with respect to our ability to achieve such results in future high performance computing (HPC) systems running real applications? It has been argued widely that LINPACK is not at all representative of real HPC applications; its computational characteristics stress only a few hardware components: mainly, L1 cache plus ﬂoating-point adders and multipliers. As a result, a companion benchmark was introduced in November 2014: HPCG (High Performance Conjugate Gradient) [2]. Interestingly, since then HPCG performance obtained by the fastest system has remained constant at around 0.6 Petaflops, while the best LINPACK performer has seen a 3× increase from 33 Petaflops to 93 Petaflops. Section II examines the past 10 years of benchmarking in more detail.

The large (over 100×) and increasing gap between these two benchmarks is annoying if we expect the Exaﬁlop race to have any real-world value. The ACM Bell HPC award performance is more representative of real applications and lies between these numbers, but the rate of performance increase seems to be slowing in recent years. Many realistic benchmarks and full application performance must be evaluated. Further, we need to understand what is going wrong. Is the hardware poorly designed? Are software (SW) tools inadequate for restructuring applications or even helping code developers redesign critical parts of application? Is it necessary to redesign current applications from scratch?

Basically, “software” and “hardware” are the main system optimization knobs. Simply put, “software” techniques range from fully-automated approaches, to manual code restructuring as guided by performance tools, to full application rewriting as a last resort. At the other extreme, “hardware” approaches involve next generation hardware/software co-design, where the application and the hardware can be simultaneously adjusted to improve their performance match. This process is becoming more flexible and complex with FPGAs (Field-Programmable Gate Array) embedded into processors and other SoC (system on chip) techniques that allow hardware to be “customized” to match application classes. A main difficulty is how to identify application classes that would benefit from hardware customization.

Ultimately, it makes sense to apply all of the approaches above in concert to reach maximum performance potential. Although this is easier said than done, we present perspectives on how autotuning and performance tools can contribute to the objective, listing the most promising directions and the most obvious needs (see Sections IV and V). For application redesign, it is more difﬁcult to provide a uniform view since application domains are quite different. For this reason, after reviewing Gordon Bell Award winners, we give an historical perspective on molecular dynamics code evolution (see Section VI). We selected this domain because it is rich in terms of code development and includes specialized hardware development (ANTON Machine). Finally, we will sketch the key ingredients of a methodology to improve hardware/software co-design. (See Section VII).

The major contributions of this paper are:

1) Optimization: We review state-of-the-art methods for optimizing applications, and new research paths to help reduce the sustained versus peak performance gap, while controlling energy consumed.

2) Developer input: We emphasize the importance of application developer input for analyzing and tuning performance and energy, and advocate semi-automatic (versus fully-automatic) approaches to assist code developers in optimization.

3) Hierarchical decomposition: We establish the importance of top-down application performance and energy decomposition, leading to analyses of speciﬁc loop types (codelets), the corresponding data types and sizes, and their combined effects on those speciﬁc HW nodes that limit the resulting computations.

4) HW/SW co-design: Using workload-wide codelets, data sets, and HW node details, we deﬁne optimization methods that are balanced between hardware and soft-
For over a decade, the SPEC benchmarks [3] have charted the performance of processors and SMP systems through a well-defined, thorough measurement protocol applied to multiple architectural variants, different run types, parallelism flavors, and scale. Although the SPEC benchmarks can be criticized for their inability to predict real workload performance, they provide useful information on system performance evolution. This section uses historical information to identify important performance trends and issues over the past decade.

To complement the SPEC performance data, we also consider theoretical arithmetic peak performance, which helps us understand sustained performance improvement across new systems. Moreover, comparing sustained and architectural peak performance allows us to quantify the gap between achieved performance and what the hardware can deliver.

We select the SPEC FP 2006 benchmarks as a reference. The SPEC measurements are performed on a whole system, so comparison between years includes changes to the processor, the OS, the runtime, and the compiler. This bundling of several factors together is reasonable because the evaluated source codes remain unchanged and we are interested in overall system evolution. Figure 1 presents the performance improvement for each year compared to the previous one from 2007 to 2016. Table I shows the 10 XEON-based server characteristics used to perform the measurements. To better emphasize the architectural evolution, we display both the CPU model and the corresponding key memory hierarchy numbers.

![SPECspeed and DP peak speed-up year per year](image)

<table>
<thead>
<tr>
<th>System</th>
<th>CPU</th>
<th>Year</th>
<th>Core DP (Gflops)</th>
<th>Freq (GHz)</th>
<th># Cores</th>
<th>L3 RAM (MB)</th>
<th>RAM (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harpertown (45 nm)</td>
<td>X5482</td>
<td>2007</td>
<td>12.80</td>
<td>3.20</td>
<td>8</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>Harpertown (45 nm)</td>
<td>X5492</td>
<td>2008</td>
<td>13.60</td>
<td>3.40</td>
<td>8</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>Gaisenstow (45 nm)</td>
<td>W5590</td>
<td>2009</td>
<td>13.32</td>
<td>3.33</td>
<td>8</td>
<td>8</td>
<td>48</td>
</tr>
<tr>
<td>Westmere-EP (32 nm)</td>
<td>X5680</td>
<td>2010</td>
<td>13.32</td>
<td>3.33</td>
<td>12</td>
<td>12</td>
<td>48</td>
</tr>
<tr>
<td>Westmere-EP (32 nm)</td>
<td>X5690</td>
<td>2011</td>
<td>13.88</td>
<td>3.47</td>
<td>12</td>
<td>12</td>
<td>96</td>
</tr>
<tr>
<td>Sandy Bridge-EP (32 nm)</td>
<td>ES-2690</td>
<td>2012</td>
<td>23.20</td>
<td>2.90</td>
<td>16</td>
<td>20</td>
<td>64</td>
</tr>
<tr>
<td>Ivy Bridge-EP (22 nm)</td>
<td>ES-2690 v2</td>
<td>2013</td>
<td>24.00</td>
<td>3.00</td>
<td>20</td>
<td>25</td>
<td>128</td>
</tr>
<tr>
<td>Haswell-EP (22 nm)</td>
<td>ES-2690 v3</td>
<td>2014</td>
<td>41.60</td>
<td>2.60</td>
<td>24</td>
<td>30</td>
<td>256</td>
</tr>
<tr>
<td>Haswell-EP (22 nm)</td>
<td>ES-4650</td>
<td>2015</td>
<td>33.60</td>
<td>2.10</td>
<td>48</td>
<td>30</td>
<td>512</td>
</tr>
<tr>
<td>Broadwell-EP (14 nm)</td>
<td>ES-2690 v4</td>
<td>2016</td>
<td>41.60</td>
<td>2.60</td>
<td>28</td>
<td>35</td>
<td>256</td>
</tr>
</tbody>
</table>

TABLE I

REFERENCE ARCHITECTURES FROM 2007 TO 2016.
features. Again, as observed in the period before 2011, other factors (RAM and L3 size increase) account for performance improvements. Note that 2015 offers an interesting hiccup due to the server reference machine operating at a lower frequency (2.1 GHz) compared to 2014 and 2016. This justifies both 2015 over 2014 lower and 2016 over 2015 higher values.

These SPECspeed measurements are very optimistic about core performance evolution because they correspond to unicore measurement. Using a single core to run the benchmark, it gives full access to shared resources such as L3 and RAM processor bandwidths. A more realistic evaluation is obtained by using SPECrate throughput measurements, another SPEC score for performance evaluation. Unlike SPECspeed, SPECrate is evaluated by simultaneously running a copy of the benchmark on each available core, thus stressing shared resources.

Figure 2 compares, for each SPEC FP benchmark, the overall speedup from 2007 to 2016 systems for both unicore SPECspeed and SPECrate normalized by the number of cores. Peak unicore FP performance has improved by a factor of 3.25× from 2007 to 2016, and 8 out of 17 codes have unicore speedups exceeding that threshold. As seen in Table I, this is due to L3 and RAM size increase. However, the situation is drastically different when we consider SPECrate per core: only one code marginally exceeds the FP arithmetic speedup threshold. One code has even a lower SPECrate per core in 2016 compared to 2007. 14 out of 17 codes achieve a speedup lower than 2× in 10 years, demonstrating an increasing gap between peak performance and performance on real codes on a unicore basis. However, it should be noted that at the processor level, the core count increase across the years (8 cores in 2008, 48 cores in 2015) has allowed a substantial increase in the standard SPECrate. The overall performance of a system can be approximated by the product, number of cores times unicore performance, hence our focus on unicore performance.

The SPEC metric SPECmpiM_base2007 evaluates the sustained parallel performance of the SPEC MPI 2007 benchmarks which use MPI message communication. For simplicity, we refer to this metric as SPECmpiM in this paper. A SPECmpiM_base2007 score for each benchmark is calculated as its execution time on a fixed historical reference machine divided by its execution time on the target machine. Similarly to SPECSpeed for serial codes, the SPECmpiM_base2007 score for the benchmark suite is defined as the geometric mean of the individual scores.

To evaluate performance scalability, we consider three different systems: SGI ICE X E5-2690 v2, SGI ICE X E5-2690 v3, and SGI ICE XA E5-2690 v4. That are based on compute nodes featuring the respectively presented CPU of years 2013, 2014 and 2016 in the Table I. They have the same number of nodes but different numbers of cores per node (see Table II). This illustrates the trend to increase the number of cores per node over time.

Figure 3 presents the performance scalability of the SPEC MPI 2007 codes run on medium (MPIM) datasets at node level. Blue dots represent the SPECmpiM metric while red triangles illustrate a perfect scaling per node (i.e., executing a benchmark on n nodes is n times faster than executing it on a single node). To quantify the SPECmpiM performance distribution per benchmark, we also include a violin plot for each evaluated configuration.

Since the number of cores per node increases over time, we should expect performance improvement. Yet, by looking between 2013 and 2016, we see relatively small SPECmpiM_base2007 performance improvement. This is illustrated by the blue dots that are at similar levels across the three architectures. We also note that the performance distribution of the benchmarks is little affected by architectural changes: violin shapes remain similar across the generations. Table II summarizes key numbers to emphasize the increasing gap between sustained and DP peak performance for parallel applications. The peak system performance of the 40-node systems has increased by 2.43× while SPEC performance has only increased by 1.28×. Providing high performance levels (Teraflops, Petaflops, and ultimately Exaflops) at reasonable cost clearly requires reducing this gap between sustained and peak performance.

### III. A Closer Look at Hardware Use

The above analysis, performed at the core and SMP system level, can be pursued further by looking at the individual hardware components in the machine. For most of the key components, we can associate abstract “nodes” (defined by Bandwidth, Size and Latency). Bandwidth and latency are widely used terms, with various interpretations for physical

<table>
<thead>
<tr>
<th>System</th>
<th># Nodes</th>
<th>Cores per node</th>
<th>SPECmpiM</th>
<th>Peak (Tflops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGI ICE X E5-2690 v2</td>
<td>40</td>
<td>20</td>
<td>100.3</td>
<td>19.2</td>
</tr>
<tr>
<td>SGI ICE X E5-2690 v3</td>
<td>40</td>
<td>24</td>
<td>111.6</td>
<td>39.94</td>
</tr>
<tr>
<td>SGI ICE XA E5-2690 v4</td>
<td>40</td>
<td>28</td>
<td>129.0</td>
<td>46.59</td>
</tr>
</tbody>
</table>

**TABLE II**

**KEY CHARACTERISTICS OF THE THREE 40-NODE TARGET SYSTEMS OF FIGURE 3.**
The marks are used for evaluating the maximum. These rates can be misleading by a factor of 10X, and because only one or two data sets are run on committee-defined code extracts, may not represent a given field very well. Such methods). Figure 4 presents saturation ratios (for the hardware nodes listed above) for 27 of the “hottest” YALES2 loops. All of the bandwidths were derived from nominal specifications, while all of the capacities were computed from hardware counter performance measurements. First, we observe a large variation in saturation ratios over the studied loops, justifying the need for per codelet study. Because typical applications share a lot of code that is written in a particular style, 10s to 100s of codelets may provide good coverage (this is true of SPEC codes). However, the range of node use in Figure 4 clarifies the inability of Linpack or HPCG (having only a few codelets) to reflect the realities of HPC computation.

Second, most nodes have low hardware utilization. This could be explained in part by a low vectorization ratio, but the use of throughput mode should still result in a favorable use of hardware resources. These low saturation ratios reflect poor processor-code match and underutilized components that will result in wasting static power. The traditional argument is that seldom-used HW is worthwhile because it is crucial when it is used (like an expensive fire engine). However, if shared memory SoC nodes can be specialized to such functions, they can be put to sleep (saving idle energy) until used, and can be designed specifically for given functionalities to provide high performance and saturation (consuming minimal dynamic energy).

Correlated with the notion of resource usage is the problem of defining computation rates. Three main approaches have been used:

1) Machine oriented rates, e.g. instructions per cycle, can be compared across any computations on any systems, but do not distinguish scalar (one word) from vector (8 or more words) instructions so can mislead by 8X or more. Normalization to byte rates would improve the situation.

2) Artificial benchmark rates reduce a set of computations to a single average number. Fig. 3 shows that this can be misleading by a factor of 10X, and because only one or two data sets are run on committee-defined code extracts, may not represent a given field very well. Such
numbers are widely used as indices across many systems in marketing, but not for machine selection by experts.

3) **User-controlled, domain-specific rates** can give solid insights across systems. This is seldom done, although standard domain-specific data sets are used (e.g., autonomous vehicle simulations of one car changing lanes or a pedestrian crossing; car crash simulations of large/small vehicles etc.). Such simulations can produce pixel or frame rates, crash simulations per hour (resp.), etc. Quality tools should promote easy domain-specific rate definition and collection to aid users, even allowing multiple rate types per application.

### IV. **Automated Tuning** (Autotuning)

In recent years, there has been considerable research in automated performance tuning (more commonly known as autotuning) to address complex performance optimization problems and to overcome the time-consuming and error-prone process of iterative manual code updating based on performance measurement and analysis. In general, autotuning is a methodology for improving the performance of a program based on an automated procedure for empirical- or model-driven performance optimization. An important objective of autotuning is to maximize performance and productivity without sacrificing portability.

Conceptually, autotuning targets a space of possible versions of a program (called variants), and searches the space systematically to identify which variant delivers better performance. Autotuners differ in both how they generate program versions and what strategies they apply to explore the space for higher performing outcomes. How program variants are produced is an aspect of what code forms or parameters are being modified by the autotuning systems. Autotuners can work by changing compiler settings, utilizing alternative algorithms, applying code transformations, controlling execution parameters, etc. Because the performance space is potentially complex and large, autotuning systems can seldom evaluate the entire space and must apply a search procedure based on partial evaluation.

The motivation for autotuning technology comes from the increased complexity of the execution platform (e.g., multiple cores, deep memory hierarchies, shared resources, heterogeneity) making parallel programming for performance more challenging. Autotuning allows customized optimizations (as a function of the target architecture and code), enabling potential performance gains over generic strategies. Autotuning for multiple objectives, such as performance and power, is also possible. Generally, the production of program variants is automated. The variants are then executed, measured, and compared. Based on the analysis, the one with the best performance is selected, or the search continues and more variants are evaluated. The key issues are identifying and generating the search space, pruning the search space to manage evaluation cost, and measuring/analyzing performance. Depending upon the target (from libraries to code restructuring to full applications), autotuners differ greatly in their techniques.

#### A. Optimizing Compilers / Profile Guided Optimization

Compilers have from the beginning integrated performance models to select the best generated code. To compensate for the lack of dynamic information at compile time, compilers also generate multiple versions of code with embedded tests in order to select dynamically the best code version. A step further in that direction is to use profile-guided optimization (PGO), where the code is instrumented to collect runtime behavior (branches, calls, and so on), then run to gather the target information, and finally to feed this information back to the compiler to fine-tune path optimization, unrolling strategies, etc. PGO is implemented both in research compilers (such as LLVM [6]) and in industry compilers (e.g. INTEL C and FORTRAN Compilers).

#### B. Compiler Switch Optimization and Iterative Compilation

Empirically, “playing” with compiler switches has been a traditional recipe for “benchmarkers” who had little time to optimize a customer application directly. More formally, iterative compilation applies a string of compiler transformations or flags to a program, evaluates their impact by executing the resulting code, and eventually looks for a better string [7]. The number of flags per transformation string to be explored can become large very quickly: assuming $K$ compiler flags available, properly setting a string of $N$ flags leads to exploring $K^N$ combinations.

Due to the search space size, early efforts targeted speeding up exploration. Techniques included genetic algorithms [8] and reuse of past searches performed on a training set with clustering. Genetic algorithms navigate through the search space using a fitness function to drive selection, crossover, and mutation operations.
space by mimicking the natural selection process. On the other hand, clustering uses code characterization to build code classes for which good strings of flags have been predetermined via testing on a reference (training) set of codes [9]. The notion of clustering/characterization was pushed further by using performance counters to generate code “performance” signatures and associating with each signature a string of compiler flags predetermined on a training set of codes [10]. Combining feedback-directed optimization and iterative compilation allows outcomes from evaluation of code variants to inform the compiler and refine its code generation strategy, as illustrated by the autotuning framework CHiLL [11, 12].

Autotuning frameworks based on code transformation can work at different language levels: from domain-specific languages (DSL), to intermediate representations, to assembly code. The Orio [13] autotuning framework is a source-to-source transformation tool that uses code annotations to inform optimizations, parameterized code generation, and automated empirical execution of code variants on a target machine. It can support alternative search heuristics, with the user being able to specify preferred search strategies.

One interesting gauge of iterative compilation potential is to analyze the impact of compiler flag optimization on the SPECFP benchmark. SPECSpeed numbers for FP benchmarks come in two flavors: baseline (using standard optimization flags) and optimized (special flags being manually inserted by experts). Figure 5 shows speedup obtained by flag optimization (over baseline) on the SPEC FP codes during the last 10 years. Remarkably, only a third of the codes achieve real performance improvements. More interesting, these speedups are obtained repetitively every year.

We analyzed three codes (games, tonto, and povray) for which optimized versions were significantly better than baseline. Between 2012 and 2015, the flags of the optimized versions were close to identical to the original configuration, furthermore all of them included profile-guided optimization. We also performed a global analysis over the full SPECFP2006 database containing over 140,000 benchmark executions, about 10 years × 850 systems on average per year × 17 codes. On average, the optimized versions provided a 5% gain over baseline version. One of the limitations of the optimization approach is that flags are applied over the whole code. An approach with specific flags for every loop could generate more improvement at the expense of a more expensive search.

C. Libraries

Historically, libraries were the first targets for autotuning because they focused on a specific set of functions, drastically simplifying the space of variants to be explored. PHiPAC (Portable High Performance ANSI C) [14] and ATLAS [15] first popularized the notion of a self-tuning library, in this case basic linear algebra subroutine (BLAS) libraries. The general idea was to parameterize the algorithm implementations based on hardware characterization to generate multiple code variants from automated code transformations and data size selection. The variants could then be executed, measured, and fed back into the PHiPAC or ATLAS search engine.

In a similar fashion, OSKI [16] and BeBOP [17] targeted sparse BLAS and its indirect, irregular memory access and low computational intensity. Here tuning is important because exploiting matrix structure can dramatically affect execution performance. However, knowledge of the target architecture is also required to make the right code transformations. OSKI benchmarks the target architecture to determine the high-performance variants. Workload information and matrix properties are then used to evaluate models and select program-specific data structures and code.

FFTW [18] used similar techniques to generate optimized FFT versions for different architectures. SPIRAL [19] extended the space of variants explored by moving from declarative algorithm representation for signal processing, to high-level language rewriting for final code generation. The general idea is that a library is generated and tuned for a specific problem case and a specific target platform. High optimization levels can be achieved with full knowledge of the algorithm, application-specific problem, and machine. SPIRAL is particularly interesting because it introduces first domain knowledge in the performance optimization process and second a high level algorithmic representation of the search space.

D. Autotuning Limitations and Promising Directions

We see four major limitations to autotuning. First, relying on runs to drive the optimization makes the optimization itself dependent upon the parameters of the run (e.g., inputs). Second, the space of transformations is somewhat limited to the compiler technology available. Third, searching through the variants is challenging due to space size. Finally, the granularity of tuning is a major issue. Optimizing entire applications is different from just focusing on a library or code region. Let us look at more details in these four roadblocks.

Since autotuning relies on measurements, two issues arise simultaneously. First, measurements can be only performed on existing hardware platforms, the cost of using simulators being prohibitive during the search phase. This prevents autotuning from being used to explore hardware space design. Second, the “best” variants selected through runs will be the best for a given dataset input. This suggests that the autotuning outcome (ultimately and correctly) can only be determined by a user who knows the target platform and exactly the representative and significant datasets of interest. In the worst case, there might be as many code variants as dataset inputs.

Beyond measurements, the second key limitation of autotuners is the space of variants explored. Compiler switches are limited and do not fully exploit all of the potential power of the compilers. An interesting approach proposed in [20] is to use generative programming to implement architecture-aware optimizations for linear algebra kernels. Another approach is presented in tools like Active Harmony [21] which apply search strategies within a space of possible combinations, for instance, to support application-level tuning by simultaneously evaluating user-defined tunable parameters. These could be both higher-level parameters, such as those affecting data structure distribution and concurrency granularity, as well as code-level controls, such as loop unrolling, blocking, and...
scheduling. With each variant, performance is monitored and fed back into the search algorithm until convergence.

The third limitation, is that the number of variants increases search complexity. To deal with this problem there has been significant interest in combining machine learning techniques with autotuning. In a way, this combines the best of empirical- and model-driven approaches by using methods to create statistically-derived models from data that can then be used to guide search [22]–[24]. The potential benefits are in reducing the search space complexity by modeling the relationship among the factors that contribute to it. It should be emphasized that the goal function is in general highly non linear due to the architectural interactions resulting in performance inflections. This may force a larger sampling of the search space to discover highly performant variants.

The fourth major limitation is the granularity at which the autotuner is operating. For instance, operating at the whole application level is far from optimal due to the heterogeneity of codelet behavior. Ideally, autotuning systems should operate on individual loops, allowing different loops to use different tuning parameters. However, this drastically increases the search space. To overcome this difficulty, Fursin et al. [25] propose dynamically selecting the best loop variant during execution, to exploit the fact that the same loop is executed millions of times in a real application and several variants can be tested across these millions of executions. This can drastically reduce search cost (several variants are explored through a single application run), but the remaining number of full application runs needed is costly.

A more promising approach is offered by CERE [26] which is an application code extraction tool that can be used to generate codelets as proxies to tune compiler and runtime parameters for different code regions. One of the key characteristics of CERE is its ability to preserve execution context, so that code fragments executed (in vitro execution) after their extraction have a similar behavior to that within the application (in vivo execution). The cost of measuring variants is dramatically reduced because instead of running the whole application, only extracted codelets have to be executed [27], [28].

Autotuning frameworks are only as powerful as their ability to generate “high value” code variants and to manage the experimentation and search complexity. The former depends a great deal on what the autotuner knows about the optimization space and how well it can effectively produce variants to test. While most code transformations are within the scope of current compilers, knowledge of the target hardware is harder to get. The latter depends on how well the search strategy can reason about the experimental results thus far to guide the choice of the next experiment to conduct. This requires knowledge of the interplay of performance factors in a complex hardware/software platform, so an analysis must be carried out of per experiment cost versus search intelligence. More promising techniques are incorporating architectural models (to better interpret experiment data) as well as code and data set isolation (to better reduce testing cost).

V. OPTIMIZATION WITH PERFORMANCE ANALYSIS TOOLS

Parallel performance optimization practice has long been dominated by the iterative process of developing the code, measuring its performance on a target platform, analyzing the measured data to identify inefficiencies, and modifying the code to improve performance. Evolving HPC technology over the last 30 years has focused much of the parallel performance tools community’s attention on developing instrumentation and measurement infrastructure for performance observation purposes. Indeed, significant advancements have resulted in probe-based and sample-based instrumentation [29]–[33], access to high-resolution timers and hardware counters [34]–[36], and parallel profiling and tracing measurement [37]–[43] that can scale to large HPC machines. Although the performance data captured in these measurements primarily report metrics for low-level events that occur on threads and processes during parallel execution, parallel programming technologies have advanced to expose performance interfaces that provide access to runtime events and performance factors of relevance to parallel execution semantics. Presently, the state-of-the-art performance analysis tools can process large parallel profile and trace data [37]–[39] generated from performance experiments and produce results that generally reflect basic properties of HPC application execution (e.g., time distribution, hardware behavior, load imbalance, synchronization waiting). However, there is much less support for automated reasoning about performance problems and guidance for performance improvement.

A. Performance Observation

One of the major challenges of performance evaluation tools is in determining what is of interest to measure in a program. Without information about specific events of interest to observe from the application developer, the tool only has the program code as a reference. Tools can observe the program code both statically (prior to execution) and dynamically (during execution). Static performance analysis tools allow detailed interrogation of program code (source to binary), across multiple levels of code transformation/generation [30], [38], [44]. The general objective for performance purposes is to understand features and characteristics of the code and to have data that relates the different code forms and types. Static performance analysis is important for the mapping of performance properties and measurements to static program structures, control and operations.

Dynamic performance observation measures a program’s execution using techniques that can be classified generally as sample-based and probe-based [29]. Sample-based methods rely on observation of a program’s state when it is interrupted. The state is essentially what can be interrogated when the program is halted, which includes the program counter (PC), calling stack, and any other observable program information. This information provides a program context for interpreting the measured performance data, including time (sample period) and hardware counters. Sample-based methods attribute performance data to the program code location and callpath as identified by the context.
Probe-based methods use instrumentation to insert probes into the code at specific locations to observe the execution. The instrumentation can occur from the source level to the binary level. Execution of a probe generates an event that captures program context and performance data at that time. The context in this case can include the code location, the callpath, and any other information the probe code provides. Performance data is equivalent, but the attribution is more exact, versus statistical in sample-based methods. However, probe-based methods have to be managed carefully to avoid huge unmanageable traces and large overhead.

In both sample-based and probe-based methods for performance measurement, the program code is used to place the performance data in the structural context of the program. Sample-based methods are lower-level in that they rely on static analysis to map PC information back to the program code, whereas probe-based methods can carry more semantics with events. For this reason, it is potentially more difficult to reconstruct higher-level functional representations of application performance using sample-based methods alone.

B. Performance Analysis, Experimentation, and Diagnosis

In general, the problems addressed by the HPC performance tools community (academic and industrial) are fundamentally complex and technically difficult. Nevertheless, many of the tools are robust, portable across modern parallel computing platforms, many are available in open source forms, and have developed user communities [33], [36]–[39], [41]–[43], [45], [46]. While the present tools are of practical importance, the challenge of parallel performance diagnosis and optimization remains elusive. A critical area of concern is the lack of knowledge necessary to correlate performance observed to what is known of the underlying architecture and to understand performance relative to the computational model of the application. Most performance tools are informed by only what can be gleaned from the application code or observed during execution. Hence, the application is seen by the tool as a “black box” from the perspective of what parallel algorithms are used, how parallel/distributed data are allocated and managed, why parallel interactions occur, and what performance outcome is to be expected for the application and input.

Without this knowledge, performance measurement must focus on low-level actions that take place in the parallel execution, and performance analysis must attempt to infer high-level performance behavior.

An initial step in this direction is to build better tools for automating performance experimentation [47] and organizing the performance information produced from performance experiments and their analyses. Certain performance tools implement performance database technology [38], [48] already. However, it will be important to improve the performance management system to support metadata, data mining, analysis pipelines, and inferencing [49], if knowledge-based reasoning for performance diagnosis and optimization is to be realized [50]. In general, these systems must support multi-experiment performance analysis, since the ability to conduct comparative analysis is central to the methodology for understanding critical factors and their performance effects. In addition to the performance data, metadata is used to describe the performance experiments and encode information about the context in which the experiment was done. Moreover, it is important to structure and represent the information in the system to facilitate analysis, characterize results, and represent performance knowledge [31], [48], [49].

C. Limitations and Promising Directions

The first major limitation of performance analysis tools is the lack for most of them of any quality control process: numbers are generated without any way of assessing whether measurements results are meaningful. The situation is worsened due to lack of standards, making the output comparison of various tools extremely difficult. Even for simple information such as profiling, every tool has its own format. For the community it would be good to agree on some output standards and then also manage some sort of global repository where performance information obtained by various tools could be gathered on reference codes and machines. Furthermore there are fundamental limitations to accuracy which are not taken into account. In some cases, the performance data is difficult to attribute: matching hardware events and assembly instructions is far from being reliable [51]. Similarly, measurements, especially on large parallel systems, are inherently irreproducible.

Second, the level of information provided by the tools is much too low-level, most of the metrics gathered refer to hardware component usage (essentially specifying problems) while a code developer might not have the necessary architecture background or access to the right documentation to fully understand low-level metrics. Additionally some of these metrics are hard to interpret correctly: for example, shortage of physical registers is not necessarily associated with poor register allocation but more likely with long latency instructions. Finally, such hardware events do not allow in general to identify costly instructions and therefore optimization techniques aiming at working around such costly instructions cannot be used.

Third, for optimization purposes, it is important to link performance analysis tools to the main knobs that a programmer has available to tune their code such as compiler switches, programming pragmas or hand rewriting of code portions. For example, PerfExpert [52] aims directly at providing the users with solution prescriptions to performance problems. PerfExpert will first detect and diagnose the causes for performance bottlenecks in each procedure and loop of an application. Then it will try to apply pattern-based software transformations to the source code to remove a performance bottleneck. If this automatic step is unsuccessful, it will provide the user with performance analysis reports and suggestions for bottleneck remediation. In a similar manner, CQA [53] uses assembly static analysis to identify potential compiler deficiencies, and proposes suggestions for workarounds. However, it goes a step further by providing performance estimates for various code optimizations: partial FP vectorization, full vectorization, and so on. Because CQA is based on a static model, its speed is
In general, providing better support for user interaction and control during performance analysis will also lead to diagnosis and optimization benefits. For example, a tool providing what if scenarios could allow user input to specify performance projections. In this case, the tool must have some means to predict performance. Ultimately, a performance recommendation tool could guide the user through restructuring an application.

We developed a prototype tool along these directions (extending our previous VP3 [54] results) and Figure 6 provides typical output for two applications YALES2 and AVBP. Four transformations were targeted: 1) **FP arith Vectorization** assumes a partial vectorization of only arithmetic FP operations, 2) **Clean** refers to an “FP bound” code version where non FP operations are supposed to have a zero cost, 3) **Fully Vectorized** assumes that all of the operations have been vectorized, and 4) **L1 Blocking** refers to a perfect blocking where all data access will be performed out of L1. For every loop (L) and for every transformation (T), we generated automatically potential performance expected if transformation (T) was applied to loop (L). For YALES2 (upper part of Figure 6) the code developer can clearly see that both transformations (Clean and FP Arithmetic vectorization) will bring little performance improvement: 2% for Clean and 5% for FP vectorization. On the contrary, L1 blocking will pay off much more, up to 40% (if the 20 most profitable loops are transformed). Similarly, full vectorization can bring the performance gain to 60%. On AVBP (a combustion code developed at CERFACS), performance gains will be much more limited: 10% for L1 blocking, with most of the performance gain from the first 7 most profitable loops. Full vectorization will provide higher benefit closer to 20%, but will require vectorizing 20 loops. Such approaches could be extended to cover more transformations such as array restructuring or even cases of combined transformations (blocking + vectorization). One major difficulty is the trade off between the accuracy of the performance prediction and its cost [54].

A methodology to support optimization of parallel applications should incorporate a hierarchical approach that integrates program representations (static and dynamic) and performance data (models, measurements, analysis) in a manner to enhance performance understanding, problem diagnosis, and prediction. The methodology could be based on a principle of **performance archetypes** which described different levels of abstraction, from the high-level algorithms to low-level code. In general, performance archetypes would capture meaningful performance abstractions based on their semantics, describe what performance data is required to interpret an instance of the abstraction (what might be called a performance scenario), and specify the relationships to other abstraction levels.

The importance of performance archetypes is their ability to constrain the performance analysis to what is known about performance effects associated with the archetype. Because
archetypes are tied to computational models as well as application code, the ability to predict performance based on what if scenarios will take advantage of both. While there has been success in HPC performance modeling for prediction purposes, there is a severe lack of integration of such methodologies in standard performance tools that are applied routinely in application code development.

VI. APPLICATION: THE WEIGHT OF HISTORY

The success of supercomputing in the last decade might be reflected by the the list of Gordon Bell ACM prize winners shown in Table III. This offers a brighter perspective on the ability to achieve high sustained performance on real applications, as measured by the percentage of peak performance. Indeed, impressive progress of about 180× was achieved over the past 10 years, but only about 6× of that in the past 7 years. In the first 7 years, sustained performance efficiency was over 30% of peak, and up to 77%. But the past 3 years showed efficiency below 20%. Given our earlier dire observations, such performance gains were impressive, but required a full rewrite of the application, large data sets, and extensive optimization on the target platform. Although these successes are remarkable, the corresponding application codes are usually very focused proofs of concept, and not widely usable. Real applications must run with various data sets, and cover a range of physical problems. As massively parallel systems arose over the past 20 years, many real application computations have suffered from single digit efficiencies when producing the highest flop rates. Therefore it seems that we are facing two extremes: 1) benchmark codes corresponding to routine or oversimplified use of HPC equipment and 2) extreme (“tour de force”) demonstrations reserved for a very narrow (but strategic) use.

Section III gave several rate definitions, favoring domain-specific rates. But even this can be misleading, e.g. several molecular dynamics (MD) codes are competitive on a class of physical problems. However their algorithms and the basic physics modeled can vary widely as below, so NAMD and GROMACS in Spec may be stylized code with carefully chosen data sets that loosely characterize distinct real-world MD problems. In general, the physical problem and data set define types of results, and the algorithms used, programming language, and architecture all influence the rate; both the time and operation counts can vary substantially. Similar effects can be found in many HPC applications, so while contests like the ACM Bell award are interesting, the only relevant codes for use in tuning a given application or in HW/SW codesign are a high-coverage set of codelets and data sets for an entire workload. Next we discuss the evolution of codes in one important application area: molecular dynamics.

The aim of MD is to simulate the temporal evolution of a microscopic system at the molecular scale by numerically solving a set of discrete (usually Newtonian) equations of motion. In SPECFP 2006, 4 codes of 17 are closely related to molecular dynamics, a particular application of the many-body problem. It was first introduced by Alder and Wainright in the late 1950’s to investigate the properties of simple fluids, with the first MD simulations of realistic microscopic systems performed about 20 years later. The first simulation of liquid water using a realistic molecular model was done in 1974 by Stillinger and Rahman. The first simulation of a protein was done in 1979 by MacCannom, Gelin and Karplus (awarded the Nobel prize in 2013).

Since the very first MD simulations in the 1950’s, the method limitations have been the same, and can be summarized as “how to perform efficiently a reliable simulation of a complex molecular system.” Here, reliable corresponds to three different physical concepts. First, to ensure the stability of the Newtonian equations of motion at the molecular level, the time step must be adapted to the frequency of the fastest motion within the simulated system. That yields time steps at the femtosecond (10⁻¹⁵ sec) scale, a dozen orders of magnitude smaller than the common relaxation times corresponding to many biological processes, like protein folding. Reliable also refers to the accuracy of the model (usually called the force-field) used to simulate a molecular system. Only basic additive force fields based on the systematic truncation of long range inter-atomic interactions were used in the 1970’s. Now, the ongoing increase of available computational hardware efficiency allows one to consider sophisticated force fields without truncating any interaction, and by accounting for microscopic cooperative effects. For instance, molecular polarization (a nonpairwise microscopic interaction) is pivotal to simulate molecular systems at interfaces. Such systems play a key role in understanding atmospheric pollution phenomena and their effects on climate change, for instance.

To tackle the above limitations and because of the standardization of computer hardware, since the 1970’s huge efforts were devoted to improving MD methods and algorithms. For example, the development of the multiple time step algorithms sped up MD simulations by a factor up to 5. Grid-based methods devoted to the Ewald summation technique were another major methodological achievement of the 1990’s. In particular, the numerical complexity of these grid based methods (based on a heavy usage of FFTs) is O(NlogN). Used in conjunction with modern O(N) decomposition schemes, all the latter methods allow one to readily perform nanosecond (10⁻⁹ sec) simulations of sufficiently large molecular systems, at the computation-day scale using a few multicore CPUs. This suggests that the major source of MD code speedup arises more from new mathematical/numerical methods than from code implementations able to take advantage of modern CPU architectures. This is supported by the data shown on Figure 2.

Over 10 years, the popular MD SPEC code NAMD, achieved a speedup of less than 2 (using the same mathematical and numerical methods). Moreover the data of Figure 5 shows that both NAMD and the MD SPEC code GROMACS show difficulty in getting noticeable speedup from compiler flag
than standard force field approaches, but also because of
simulation, not only because of more computational efficiency
generation. Coarse graining a molecular system may speed up the
complexity reduction of the molecular system under investi-
\[ \mu \text{sec} \] sampling of the conformations of a molecular system from sub

to develop new and original physical approaches to address
relatively poor availability stimulated the academic community
accuracy of MD approaches.

approaches, like more and more sophisticated force-fields that
up to the millisecond scale. However, besides the poor avail-
\[ 0.1-1 \mu \text{sec} \] scale at an overall affordable cost. More impressively,
a modern GPU can run standard MD code simulations at the
computational architectures.

makes MD simulations less and less well suited for standard
field strongly related to HPC since its origin in the 1950's,
Table III). This means that the specificities of MD, a research
Bell ACM winner in the last decade in the MD field (see
(see the above discussions), the ANTON machine is the sole
i.e. the ANTON machine. Interestingly and not surprisingly
the situation is even worse for large molecular systems like the FFT-based grid methods
mentioned above, which are not well-suited for use on modern
massively parallel systems. That explains the attempts to
consider more efficient computational systems, like GPUs, and
even to build a special purpose machine for MD simulations,
i.e. the ANTON machine. Interestingly and not surprisingly
(see the above discussions), the ANTON machine is the sole
Bell ACM winner in the last decade in the MD field (see
Table III). This means that the specificities of MD, a research
field strongly related to HPC since its origin in the 1950's,
makes MD simulations less and less well suited for standard
computational architectures.

For typical molecular systems of size \( N \) less than 100,000,
a modern GPU can run standard MD code simulations at the
0.1-1\( \mu \)sec scale at an overall affordable cost. More impressively,
the ANTON machine allows one to reach simulation times
up to the millisecond scale. However, besides the poor avail-
ability of dedicated hardware like the ANTON machine, the
main limitation of both the GPU and ANTON is the diffi-
culty/impossibility of implementing new molecular modeling
approaches, like more and more sophisticated force-fields that
are expected to be the next major milestone to improve the
accuracy of MD approaches.

Interestingly, the advent of the ANTON machine and its
relatively poor availability stimulated the academic community
to develop new and original physical approaches to address
hard theoretical problems like protein folding. Highly efficient
parallel methods like the replica exchange ones allow intensive
sampling of the conformations of a molecular system from sub
\( \mu \)sec scale simulations, by using relatively modest computational
resources. Other promising approaches are those leading to a
complexity reduction of the molecular system under investiga-
tion. Coarse graining a molecular system may speed up the
simulation, not only because of more computational efficiency
than standard force field approaches, but also because of
smoothing the energy barriers of microscopic potential energy
surfaces that require long time intervals to generate reliable
MD simulations. Lastly, we quote the approach developed by
one of us [57], mixing a sophisticated force field to model
intra-molecular protein interactions, a coarse grained approach
to model the protein aqueous environment and a numerical
scheme to compute inter-atomic long range interactions very
popular in astrophysics but not in the MD field (i.e. the Fast
Multiple Method).

All these new physical methods are developed first on
standard single-socket or at most dual-socket CPU systems
using common programming languages (FORTRAN or C) by
research teams that are not high-end experts in computational
science and that are not evaluated/granted based on their
coding skills. Porting codes and new algorithms to modern
computational architectures (like Nvidia GPUs or Xeon Phis)
is usually a long effort for experienced teams (see among
other [58]–[60] in the MD case), that considerably slows down
the systematic assessment and implementation in popular MD
codes (and thus the diffusion) of these new methods. In
turn, that prevents a larger use of MD-based computational
approaches by academic research teams and industry. Hence
the difficulty in using available computational resources at
their best, primarily parallel HPC systems, is now the critical
issue that prevents the diffusion of new concept and ideas,
not the lack of large available computational resources. That
contradiction has to be solved by developing "user friendly"
analysis tools that will help all types of users to speed up the
development of efficient codes as well as to account for MD
specificities in hardware design.

### VII. Application Driven Hardware/Software Codesign

The discussion above focused on performance for fixed
system HW. In the past decade, we have hit a "power wall" that
has flattened clock speed growth. This has forced architecture
changes to rely on parallelism and functionality specialization
in the form of multicore chips, vector accelerators, multi-IP
SoCs, and FPGAs. The "power wall" also forces major focus on
the energy of computation (directly affecting operating
cost). The net effect is a need for new system quality metrics,

deeper and broader workload analysis, and more comprehen-
sive codesign methods. This section gives an overview of

<table>
<thead>
<tr>
<th>Year</th>
<th>Machine</th>
<th>Cores</th>
<th>Pflops obtained</th>
<th>Pflops peak</th>
<th>Obtained/Peak (%)</th>
<th>Computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
<td>BlueGene/L</td>
<td>131K</td>
<td>0.11</td>
<td>0.28</td>
<td>39</td>
<td>Micron-scale atomistic simulation of Kelvin-Helmholtz instability</td>
</tr>
<tr>
<td>2008</td>
<td>Cray XT4</td>
<td>31K</td>
<td>0.2</td>
<td>0.26</td>
<td>77</td>
<td>Simulations of disorder effects in high-Tc superconductors</td>
</tr>
<tr>
<td>2009</td>
<td>Cray XT5</td>
<td>147K</td>
<td>1.03</td>
<td>1.36</td>
<td>76</td>
<td>Ab initio computation of free energies</td>
</tr>
<tr>
<td>2010</td>
<td>Cray XT5-HE</td>
<td>200K</td>
<td>0.7</td>
<td>2.3</td>
<td>30</td>
<td>Direct numerical simulation of blood flow</td>
</tr>
<tr>
<td>2011</td>
<td>K computer</td>
<td>442K</td>
<td>3.08</td>
<td>7.07</td>
<td>44</td>
<td>First-principles calculations of electron states of a silicon nanowire</td>
</tr>
<tr>
<td>2012</td>
<td>K computer</td>
<td>663K</td>
<td>4.45</td>
<td>10.6</td>
<td>42</td>
<td>Astrophysical N-body simulation</td>
</tr>
<tr>
<td>2013</td>
<td>Sequoia</td>
<td>1.6M</td>
<td>11</td>
<td>20.1</td>
<td>55</td>
<td>Cloud cavitation collapse</td>
</tr>
<tr>
<td>2014</td>
<td>Anton 2</td>
<td>34K</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>Molecular dynamics</td>
</tr>
<tr>
<td>2015</td>
<td>Sequoia</td>
<td>1.6M</td>
<td>0.69</td>
<td>20.1</td>
<td>3</td>
<td>Implicit solver for complex PDEs</td>
</tr>
<tr>
<td>2016</td>
<td>Sunway TaihuLight</td>
<td>10.56M</td>
<td>7.85</td>
<td>125</td>
<td>6</td>
<td>Fully-implicit solver for nonhydrostatic atmospheric dynamics</td>
</tr>
<tr>
<td>2017</td>
<td>Sunway TaihuLight</td>
<td>10.56M</td>
<td>18.9</td>
<td>125</td>
<td>15</td>
<td>Nonlinear earthquake simulation</td>
</tr>
</tbody>
</table>

**TABLE III**

**SYSTEMS USED BY GORDON BELL WINNERS ACROSS THE LAST DECADE. NA MEANS NOT AVAILABLE: SHAW ET AL. [55] USED TIME TO SOLUTION RATHER THAN FLOPS.**
A. Introduction to System Use, Selection, and Design

Hardware-software interaction is a prominent complication among factors inhibiting progress. Many SW examples are used in making system design choices, but (mostly) other SW cases are used when buyers/users select systems; because designers cannot analyze entire customer workloads, mismatches grow in terms of cost and performance. Because various assumptions influence overall customer satisfaction, more effort must be made to understand them when defining the design-selection-use cycle.

To make the discussion tractable, we distinguish three related topics in order of increasing difficulty:

1. SW tuning for a given system either static (off-line and auto-tuning) or dynamic (on-line).
2. System selection for a given workload; algorithm choice for a given system.
3. HW/SW codesign.

When selecting among existing HW and SW choices, buyers/owners have many plausible options so they may be forced to simply "pay their money and take their chances". For them, progress depends on improving options and clarifying choices in terms of their specific workloads. When a current system becomes inadequate, people first want to choose a new SW/HW system from currently existing technologies. If that fails, their next step is to modify their workload and try again. Otherwise, they must deal with choices among emerging architectural options, usually making changes to the workload.

We advocate three principles as a basis for solving the overall problem:

- Use hierarchical system analysis to decompose the problem into relevant questions per hierarchy level.
- Use a SW model of the workload and a HW model of the architecture to allow fast projections from a set of measurements to many trade off options, thereby avoiding massive, infeasible benchmarking runs.
- Breaking the problem types into SW/HW mismatch cases allows design of specialized mitigation recommenders for expert use. For example, if vectorization may be useful over hundreds of loop nests in a typically flat workload profile (see Section V), a recommender focuses human experts on top potential payoff cases in rank order.

Hierarchical decomposition and fast modeling, presented in a single graphical coordinate system allows presentation of cogent options to SW, architecture and applications experts. Their combined intuition can make selections among similar cases, help rule out tool/measurement errors, and consider cost/effectiveness tradeoffs. They are also necessary to provide and adjust application and data set inputs for analysis. Then, automatic recommendations tailored to human-chosen directions can guide expert developers toward success.

The ultimate goal is improved system upgrades, SW tuning, and communication of needs and desired outcomes from users and buyers to system designers. This would replace the process of buying and then trying a system on full workloads, with massive trying before buying. It should reduce human work, improve workload coverage, and widen system options.

B. Computational Energy and Quality

We can describe any computer system running a workload in terms of hierarchical sets of HW nodes and SW constructs. For HPC, the HW hierarchy starts with exascale nodes (of say, 100 000 DMP nodes), goes down to multi-SMP nodes, and ends at the individual CPU level where nodes are arithmetic units, cache levels, and so on. The SW hierarchy starts with a workload, goes to the application level and stops at codelets that collectively comprise most of the workload running time. The SW can be regarded in terms of source code or binary generated by a compiler, together with a data set size D. When used in a given SW, we refer to combined $<codelet,D>$ as a code (codelet execution).

Our focus begins on microprocessors whose runtime use parameters can have double digit % effects on quality, performance and energy. Codesign efforts can achieve more, and can produce much higher gains per codelet when customized chips and apps are included. Because HPC system costs (capital and operating) increase much faster than parallel performance, optimizing architectures for codelet classes from the bottom up is a necessity. For example, a 10× cost saving may achieve high quality improvement at the sacrifice of relatively small performance loss. This can be viewed as a practical compromise between general purpose systems and single-purpose systems.

1) Quality: Performance has been the main driver of HPC for 50 years; in the past decade, that has changed to include energy. However, the big step to exascale performance will be impossible without energy reduction breakthroughs. To clarify goals, we define a few general metrics for codelet execution: $C$ will refer to “average” computational rate, $W$ to “average” power, $t$ to the time required by codelet execution and $E$ to the total amount of energy consumed. Maximizing $C$ is no longer the only concern, because pure performance optimization can lead to unacceptable energy cost. Symmetrically, only minimizing $E$ is not a viable approach. Therefore quality ($Q$) metrics combining both performance and energy must be used as a primary optimization target, in tuning SW for existing HW (see Sections VII-A and VII-B above). For example, $C/W[ops/joule]$ is a traditional design engineer’s metric (energy per op), but to capture performance rate (how fast the ops are executed) we extend this to $C/E[rate/joule]$. More generally, quality metrics will take the form $Q(\beta) = C/(W\beta^\beta)$, so $\beta$ can be chosen appropriately to reflect more importance on the energy or performance side. Mitigation can be done via HW or SW changes, and the tools we discuss help both areas.

2) Energy: The energy model of nodes within a chip [61] identifies both static power (more or less independent from the workload) and dynamic parts (highly dependent upon the workload). Table IV shows how each node of a microprocessor contributes to overall dynamic energy. We used two Haswell Xeon (HSW) based systems: a client CL E3-1270 v3 and a server (SR) E5-2630 v3. Note a $10\times$ dynamic energy range for...
Table IV
ENERGY COSTS EXPRESSED IN NANO-JOULE PER INSTRUCTION. FOUR
INSTRUCTION TYPES ARE USED: SS, SD, PS, AND PD. THE FIRST LETTER
REFERS TO SCALAR (S) OR PACKED I.E VECTOR (P). THE SECOND
LETTER REFERS TO SINGLE (S) OR DOUBLE (D) PRECISION.

<table>
<thead>
<tr>
<th>Energy (nJ) / HW node</th>
<th>HSW-CL (3.5GHz)</th>
<th>HSW-SR (2.4GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front-End (n/J/Inst)</td>
<td>0.11</td>
<td>0.08</td>
</tr>
<tr>
<td>Back-End Integer (n/J/Inst)</td>
<td>0.14</td>
<td>0.13</td>
</tr>
</tbody>
</table>

Back-End FP Arithmetics (nJ/Inst)

<table>
<thead>
<tr>
<th>Operation</th>
<th>HSW-CL (3.5GHz)</th>
<th>HSW-SR (2.4GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD (SS/SD) 32/64 bits</td>
<td>0.33</td>
<td>0.33</td>
</tr>
<tr>
<td>ADD (PS/PD) 128 bits</td>
<td>0.33</td>
<td>0.33</td>
</tr>
<tr>
<td>ADD (PS/PD) 256 bits</td>
<td>0.60</td>
<td>0.45</td>
</tr>
<tr>
<td>MUL (SS/SD) 32/64 bits</td>
<td>0.16</td>
<td>0.13</td>
</tr>
<tr>
<td>MUL (PS/PD) 128 bits</td>
<td>0.30</td>
<td>0.15</td>
</tr>
<tr>
<td>MUL (PS/PD) 256 bits</td>
<td>0.33</td>
<td>0.25</td>
</tr>
<tr>
<td>DIV (SS) 32 bits</td>
<td>3.76</td>
<td>4.19</td>
</tr>
<tr>
<td>DIV (SD) 64 bits</td>
<td>5.59</td>
<td>5.46</td>
</tr>
<tr>
<td>DIV (PS) 128 bits</td>
<td>5.21</td>
<td>4.80</td>
</tr>
<tr>
<td>DIV (PD) 128 bits</td>
<td>6.32</td>
<td>6.15</td>
</tr>
<tr>
<td>DIV (PS) 256 bits</td>
<td>11.83</td>
<td>11.20</td>
</tr>
<tr>
<td>DIV (PD) 256 bits</td>
<td>14.09</td>
<td>17.05</td>
</tr>
<tr>
<td>FMA (PS/PD) 256 bits</td>
<td>0.70</td>
<td>0.63</td>
</tr>
</tbody>
</table>

Back-End Load/Store (nJ/Inst)

<table>
<thead>
<tr>
<th>Operation</th>
<th>HSW-CL (3.5GHz)</th>
<th>HSW-SR (2.4GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD/L1 (SS/SD) 32/64 bits</td>
<td>0.30</td>
<td>0.15</td>
</tr>
<tr>
<td>LD/L1 (PS/PD) 128 bits</td>
<td>0.33</td>
<td>0.18</td>
</tr>
<tr>
<td>LD/L1 (PS/PD) 256 bits</td>
<td>0.45</td>
<td>0.34</td>
</tr>
<tr>
<td>STL1 (SS/SD) 32/64 bits</td>
<td>0.69</td>
<td>0.69</td>
</tr>
<tr>
<td>STL1 (PS/PD) 128 bits</td>
<td>0.69</td>
<td>0.69</td>
</tr>
<tr>
<td>STL1 (PS/PD) 256 bits</td>
<td>0.78</td>
<td>0.69</td>
</tr>
<tr>
<td>L2 (Read/Write) (nJ/64 bits)</td>
<td>2.96</td>
<td>2.12</td>
</tr>
<tr>
<td>L3 (Read/Write) (nJ/64 bits)</td>
<td>4.59</td>
<td>4.70</td>
</tr>
</tbody>
</table>

Fig. 7. Computed C/E, C and total E for SSE codes running on the HSW-CL machine.

4) Policy definitions: We introduce the idea of policies to express the need to combine user goals with designer constraints. Both users and designers want the same result, and agreeing on a formal policy definition provides up-front and design process clarity. Each policy specifies a constrained optimization process. This may succeed but may prove infeasible, making compromise necessary in the form of splitting one design problem into two, relaxing the policy, or some combination. By applying a similar run-time policy, designs can be more robust when accommodating new codexes.

All policies take the form of a constraint set that must be satisfied when optimizing the target quality metric. Two example policies are: constant power \( W_{sys} \) and constant capacity \( C(\text{rate}) C_{sys} \). More precisely, constant power is an upper bound, while constant capacity is a lower bound. Run time policy enforcement as described above is different from procedures that merely prevent SW from violating key bounds without knowledge of running apps (HW or SW governors). This paper focuses on \( Q = C/E \) where the variables can be evaluated for a codelet or workload.

A policy algorithm may be applied to a multi-core system or to several such systems, by run-time selection among \(<\text{freq, instset, prefetch, #cores}>\) options. The multisystem application of a policy can partition a workload among several distinct system types (i.e. the workload assignment problem). An example of policy use in system design will be given in Section VII-E.

C. HW and SW change

One summary point from the above is that both architecture/HW and SW need to be “in play” both in system selection and code design and for architecture co-design. Another is that flexible selection should be much easier than current practices. Finally, better tools are needed to support new approaches. This may be obvious, but what specific steps can be taken to support these ideas in mitigating today’s basic problems?
• To get facts, systems must be measurable. Many HW counters are available, some are hard to understand, and key ones are sometimes lacking.
• If one has perfect HW counters and SW tools that summarize their results, then quality and mitigation consequences will still be useless for many SW developers (lack of HW understanding) and even for HW designers (lacking understanding of SW consequences).
• Finally, fast (equation level) models are needed, that give real-time responses at the system/application level.

We will skip the details of the first point and focus on the second below. The third is the hardest, so this paper will only provide a summary. Today, fast models are vague. Researchers usually give up on first principle approaches, resorting to “machine learning” approaches that are error prone. One reason for this is the first two points above — they do not really have adequate measurements, and they do not deeply understand many resulting numbers. An alternative is a fast cycle-level simulator [63], approaching single digit errors, and resolution to the codelet and architectural node level. Another interesting approach is the development of parametrized analytical performance models built using compiler framework (ExaSAT [64]): they achieve high speed at the expense of accuracy and generality (they are targeting a very specific regular type of computations).

For designers the problems are deeper. They rely on cycle-accurate simulators with little error, but are impossibly slow running, and often contain hard-to-change architectural assumptions (e.g. fixed cache levels). Furthermore, even with close compiler-team cooperation, studying whole workloads in forms well-matched to architectural changes is impossible for tight design cycles. A real-time solution to multi-core chip codesign would have to examine major fractions of workloads (up to tens or hundreds of thousands of codelets) together with many architectural tradeoffs, in months, not years.

In summary, better SW access to better counters is needed to drive fast modeling, in a measure once, model often approach. Extensive measurement runs of current systems can be used many times. An important assumption here is that the measurements are only useful in modeling systems of a “similar” nature. However, most single-core architectures have evolved to similar forms that are targeted at various broad workload types. This approach has been demonstrated in [65].

The problem we are addressing is to decide which key codelets are quality outliers from any reasonable modification of a single core. In other words, for normal codelets in any market, future systems should have standard cores that are designed to give acceptable Q values. For all outlier codelets, we will focus first on radical SW change (new algorithm, data structure, etc.). If that is infeasible, one needs a special purpose (SP) radical core redesign (super divide engine, simple or no cache, etc.), i.e. SoC IP, FPGA. The interconnect (on chip or off) is a major issue. We will give a new approach to understanding the feasibility of finding a HW/SW match satisfying a quality goal, and with some modifications should be useful for interconnect (requiring appropriate HW counters). The key point is that infeasible situations can be mitigated by relaxing Q restrictions (tools can show by how much), or by removing difficult applications (after codelet change has already been exhausted). Radical steps may be necessary, but pre-Silicon quality understanding is the goal for setting customer expectations and defining next generation HW/SW research goals.

D. Tools needed

The technical ideas below can be used in codesign software tools to overcome various current weaknesses. In order of necessity, next generation tools must:

1) Give general indications of acceptable or unacceptable Q behavior.
2) Point to specific locations for HW/SW improvement; codelet line numbers and system nodes in need of improvement.
3) Accurately rank the magnitudes of potential computational quality gains.
4) Give reasonable absolute magnitude estimates of the highest quality gains.

For serious developers, tools with all four features would be easy to use, and their reliability would instill confidence. Features 1-3 alone provide adequate effectiveness, but 4 is necessary for users to judge the effort/benefit aspects of engaging in a mitigation activity. The magnitudes of desired improvements determine the feasibility of specific types of change. Modest improvements may be feasible using HW or SW uncore changes, but 10× or more improvements imply dealing with multicore issues.

1) Generalized saturation and intensity: While HW/SW interactions are complex, a few reasonable assumptions allow extending the above to system-level analytics that provide top-down insights. For this, we define system saturation and intensity norms and relate them back to improving C, E and C/E. Corner cases excluded by the simplifications may then be captured by tools derived from this approach. System saturation S and system intensity I are generalizations of traditional performance metrics defined as sums of relative saturation (C_i/C_max) and normalized intensity (C_max/C_mean) over all nodes i, where maximum and mean are computed over all performance limiting nodes. Saturation measures how much of the provided HW is used by a computation, and intensity measures how well the system node use is balanced. In general, we do not want to pay for HW that is never used (unsaturated computations), nor do we want to put in expensive HW that is very busy while other nodes are idle.

2) Tool summary: A tool hierarchy can start with a quality plot of go/no-go indicators. For example, several views of the data in Fig. 7 could be used to show that imposing Q > 3 will be a major effort on codelet tuning while imposing Q > .5 only implies optimizing two codelets. However, it also shows that performance for the 7 codelets with Q > 3 is above C = 8, but that dropping to Q > 0.3 will cut performance in half for the majority of codelets. Thus, one can focus on quality and performance design expectations. The next lower level question is why are quality and performance so bad in certain cases? What specific high level information can be provided? The tool set is focused on solving one or
another specific problem. Linkages between problems complicate global solutions, and as now, require human decision-makers. To begin, required expertise varies across teams, and this influences solution directions. Then, tradeoffs among future needs require human input. Finally, human intuition can prioritize options, for example, should one first vectorize or parallelize a set of codexes, as there are underlying tradeoffs.

E. SoC/ FPGA

Among HW/SW codesign problems, SoC IP blocks are simple examples, required to deal with little run-time variety. To design systems with the highest perf/energy quality, we use quality metric \([C/E]\) subject to Silicon (Si) area, in a policy driven by the following equation:

\[
MaxQ = Max(C/E) \text{ subject to Si area}
\]

Combining \(Q\) and area is an example of how to link applications directly with HW design. Traditionally, a good design start is to squeeze into a chip a maximum number of general purpose cores, each with min power. If this leads to inadequate computational quality, the next step is to specialize some cores to workload codecex needs (in the style of currently adding a set of GPUs). Consider analyzing HPC center workloads to find key codecex outliers from codecexes that produce high \(Q\) computations on simple, low energy General Purpose (GP) cores. This would provide architectural guidelines to HW designers (in satisfying the equation above), and allow compiler designers to define recognition algorithms for such outliers.

In a natural evolution from today’s uniform core configurations, the resulting HW socket would have multicore shared memory plus local caches, together with several distinct core (or IP) types, as well as distinct memory units containing logic to preprocess data accesses for long-stride arrays and pointer/indirect (e.g. graph algorithm) data, which waste expensive cache line accesses. Besides GP cores it would have LP (Limited Purpose) and SP (Special Purpose) cores.

At a high level, suppose we produce: 256 low power, workload-customized basic cores to cover most codelets encountered, plus 32 SP and LP cores to be chosen to match a given workload’s outliers. Removing outlier codelets from their scope, simplifies the basic GP cores. We assume thousands of sockets, so millions of cores, approaching exascalpe magnitudes. By dealing with codelet distinctions at the socket level, the rest of the system can handle accessing data and hiding physical latency at higher architectural levels.

GP cores could have a cheap divider, but one LP core (for codelets dominated by performance limiting divide operations) would have a very fast divide ALU. Further, the ideal LP core for a scaling operation (multiplying an array by a constant) has only a multiplier, and the best fit for a codelet computing the sum of arrays has only an adder. Both are streaming codelets with no data reuse. The LP core we use would have both fast + and * but no divide, with a limited depth (e.g. 2 level) cache hierarchy tailored to no-reuse streaming. Idle power waste would be limited to unused arithmetic but not to cache levels that actually slow the computation, saving energy compared to present systems.

The next step beyond specialized LP cores would be dedicated SP cores that are more specialized to the algorithm level, e.g. perhaps the entire Anton pipeline for inter-atomic force calculations [55], providing 100 X performance gains at relatively low energy.

VIII. CONCLUSION

The implied reference to the Beatles song “The Long and Winding Road” (from their “Let It Be” album) in the title of our paper could easily have been replaced by the Grateful Dead’s album title “What a Long Strange Trip It’s Been.” Indeed, there is a mysterious, adventurous, and even enlightening quality to trying to make computers run as fast as possible. The days of easy performance gains by waiting for the next microprocessor generation are long gone. In its place we are left with a realization that high-efficiency HPC will come more from integrative performance technologies, knowledge-driven codesign, user-informed objectives, and new or revised physical and numerical approaches. Our hope is that this will ultimately lead to computing breakthroughs at exascale and beyond. However, it is important to remember that scientific, engineering and human discoveries productivity lies at the heart of our HPC fascination. As the Beatles “Revolution” lyrics say, “We all want to change the world.”

The history of the world’s fastest computers can be traced through many decades of mostly scientific and engineering computation topics. The name of the subject has varied: supercomputing, HPC and Greek names for increasing flops-exponents (megaflops to petascale and exascale) computing in other words exposcale computing. There are well-known dilemmas in the modern exposcale race (which sometimes takes on exposition- or fair-like qualities). The dilemmas arise in ignoring that most technical people seek energy-efficient throughput, some have real-time and energy constraints, and only a few seek maximum performance. While exposcale enthusiasts often justify their efforts in terms of maximum performance as well as technology transfer, the latter claim has several practical flaws:

- Exposcale success is usually defined narrowly, using oversimplified benchmarks or applications that under-represent the real scientific and engineering computing world in terms of algorithm types and data sizes.
- Computational quality for users and owners of fast systems usually includes both performance and cost (up-front capital plus operating energy costs), combined in diverse quality (Q) metrics and operating policies.
- Computer system design successes require balancing nearly-overwhelming engineering details. Software tools for analyses, tradeoffs, and HW/SW codesign are all too weak. We need higher-level, fast tools to eliminate many details and raise the level of intuitive decision-making.

This paper has surveyed a number of these issues, analyzing historical data up to the present, and has pointed out realistic trends. It also made some suggestions for improved ways of comparing existing systems and computations, as well as
techniques for mitigating weaknesses in computations that do not meet quality expectations.

ACKNOWLEDGMENT

The authors would like to thank Emmanuel Oseret and David Wong for evaluating saturation ratios, and Peter Tang for Anton comments. We also thank Mohammed-Salah Ibnamar for his fruitful comments. Finally, we would like to thank the anonymous reviewers for their feedback.

This work has been conducted thanks to the continuous support of Exascale Computing Research (CEA, INTEL, UVSQ) and Tocqueville Fullbright sponsoring of Allen Maloney for a 6 months visit at USVQ.

REFERENCES


