A Principled Kernel Testbed for Hardware/Software Co-Design Research

Alex Kaiser, Samuel Williams, Kamesh Madduri, Khaled Ibrahim,
David Bailey, James Demmel, Erich Strohmaier
Computational Research Division
Lawrence Berkeley National Laboratory

Abstract

Recently, advances in processor architecture have become the driving force for new programming models in the computing industry, as ever newer multicore processor designs with increasing number of cores are introduced on schedules regimented by marketing demands. As a result, collaborative parallel (rather than simply concurrent) implementations of important applications, programming languages, models, and even algorithms have been forced to adapt to these architectures to exploit the available raw performance. We believe that this optimization regime is flawed. In this paper, we present an alternate approach that, rather than starting with an existing hardware/software solution laced with hidden assumptions, defines the computational *problems* of interest and invites architects, researchers and programmers to implement novel hardware/software co-designed *solutions*. Our work builds on the previous ideas of computational dwarfs, motifs, and parallel patterns by selecting a representative set of essential problems for which we provide: An algorithmic description; scalable problem definition; illustrative reference implementations; verification schemes. This testbed will enable comparative research in areas such as parallel programming models, languages, auto-tuning, and hardware/software co-design. For simplicity, we focus initially on the computational problems of interest to the scientific computing community but proclaim the methodology (and perhaps a subset of the problems) as applicable to other communities. We intend to broaden the coverage of this problem space through stronger community involvement.

1 Introduction

For decades, computer scientists have sought guidance on how to evolve architectures, languages, and programming models in order to improve application performance, efficiency, and productivity. Unfortunately, without an overarching direction, individual guidance is inferred from the existing software/hardware ecosystem, and each group often conducts their research independently assuming all other technologies remain fixed. Architects attempt to provide micro-architectural solutions to improve performance on fixed binaries. Researchers tweak compilers to improve code generation for existing architectures and implementations, and they may invent new programming models for fixed processor and memory architectures and computational algorithms. In today's rapidly evolving world of on-chip parallelism, these isolated and iterative improvements to performance may miss superior solutions in the same way gradient descent optimization techniques may get stuck in local minima.

To combat this tunnel vision, previous work set forth a broad categorization of numerical methods of interest to the scientific computing community (the seven *Dwarfs*) and subsequently for the larger parallel computing community in general (13 *motifs*), suggesting that these were the problems of interest that researchers should focus on [1, 2, 9]. Unfortunately, such broad brush strokes often miss the nuance seen in individual kernels that may be similarly categorized. For example, the computational requirements of particle methods vary greatly between the naive but more accurate direct calculations and the particle-mesh and particle-tree codes.

In this paper, we present an alternate methodology for testbed creation. For simplicity we restricted our domain to scientific computing. Superficially, this is reminiscent of the computational kernels in Intel's RMS work [12]. However, we proceed in a more regimented effort. We commence with the enumeration of problems, proceed by providing not only reference implementations for each problem, but more importantly a mathematical definition that allows one to escape iterative approaches to software/hardware optimization. To ensure long term value, we augment each with both a scalable problem generator and a verification scheme. By no means is the list of problems complete. Rather, it constitutes a sufficiently broad yet tractable set for initial investigation.

2 An Evolved Testbed

For a testbed to have long-term and far-reaching value, it must be free and agnostic of existing software and hardware. Today, the underlying semantics of memory and instruction set architecture leech through into benchmarks and limit the ability for researchers to engage in truly novel directions. Languages and programming models should not expose the details of an architectural implementation to programmers, but rather allow the most natural expression of an algorithm. When operating on shared vectors, matrices, or grids, the dogmatic load-store random access memory semantics may be very natural and efficient. However, when operating on shared sets, queues, graphs, and trees, programmers are often forced to create their own representations built on an underlying linear random access memory using loads, stores, and semaphores. This should not be.

To ensure we did not fall prey to the tunnel vision optimization problem, we made several mandates on our evolved testbed. To that end, we strived to stay away from the conventional wisdom that suggests that parallelization and optimization of an existing software implementation is the challenging problem to be solved. Rather we believe the starting point is not code, but a problem definition expressed in the most natural language for its field. The Sort benchmark collection [31] initiated by Gray exemplifies the future vision for our reference kernel testbed. The sort benchmark definitions are based on a well-defined problem, include a scalable input generator, multiple metrics for assessing quality (for instance, sort rate for a terabyte-sized dataset, amount of data that can be sorted in a minute or less, records sorted per joule, and so on), and finally a verification scheme. Framing the benchmark objectives as an open challenge, rather than providing an optimized implementation of a particular approach, has led to novel algorithmic research and innovative engineered sort routines. We believe similar results can be attained in other fields.

Although this argument may sound vague, we found the textbook taxonomy to describe problems illustrative. The *"solution"* is the efficient co-design of software and hardware to implement a *"problem"* described in a domain-specific mathematical language (*e.g.* numerical linear algebra, particle physics, spectral analysis, sorting, etc.). The veracity of the solution is determined via an accompanying verification methodology specified in the same domain-specific mathematical language. We may provide *"hints"* to the solution in the form of reference and optimized implementations using existing languages, programming models, or hardware. The quality of the solution is based on the performance, energy, cost (amortized by reuse), and designer productivity.

In the following sections, we will describe and illustrate this process of problem definition, scalable input creation, verification, and implementation of reference codes for the scientific computing domain. Table 1 enumerates and describes the level of support we've developed for each kernel. We group these important kernels using the Berkeley Dwarfs/Motifs taxonomy using a red box in the appropriate column. As kernels become progressively complex, they build upon other, simpler computational methods. We note this dependency via orange boxes. We must reiterate that by no means is our list comprehensive. For example, the finite difference methods listed in the structured grid section are easily understood and representative, but are often replaced by more complex methods (*e.g.* the finite volume and lattice Boltzmann methods) and solver acceleration techniques (multigrid, adaptive mesh refinement).

2.1 Problem Specification

After enumeration of a set of important numerical problems, we create a domain-appropriate high-level definition of each problem. To ensure future endeavors are not tainted by existing implementations, we specified the problem definition to be independent of both computer architecture and existing programming languages, models, and data types.

For example, numerical linear algebra has a well developed lexicon of operands (scalars, vectors, matrices, etc.) and operators (addition, multiplication, transpose, inverse, etc.). Although programmers are now accustomed to mapping such structures to the array-like data structures arising from the linear random access memories in computer architecture, such an end state is the product of decades of focused optimization of hardware and software. It is not an inherent characteristic or mandate in the problem definition.

Conversely, graph algorithms are often defined as operating on edges and vertices via set and queue operations. Programmers are often forced to map such operands and operators onto architectures optimized for linear algebra. Although such techniques have sufficed in the single-core era, parallelization of set and queue operations on shared random access memories via kludges like atomic operations is unnatural and error prone. By taking a step back to a high-level problem definition, we hope designers may free themselves of their tunnel vision and build truly novel systems adept at such computations.

Whenever possible, we specify the high-level parallel operations (for all, sum, etc.) to be independent of whether or not such constructs will create data dependen-

Kernel	Dense Linear Alg.	Sparse Linear Alg.	Structured Grids	Unstructured Grids	Spectral	Particles	Monte Carlo	Graphs & Trees	Sort	Definition	Reference	Optimized	Scalable Inputs	Verification
Scalar-Vector Mult.	🟥									✓	✓		✓	
Elementwise-Vector Mult.	🟥									✓	✓		✓	
Matrix-Vector Mult.	🟥									✓	✓		✓	
Matrix-Matrix Mult.	🟥									✓	✓		✓	
LU Factorization	🟥									✓	✓	✓	✓	✓
Symmetric Eigensolver (QR)	🟥									✓	✓		✓	✓
Cholesky Factorization	🟥									✓			✓	
SpMV (y=Ax)		🟥								✓	✓	✓	✓	✓
SpTS (Lx=b)		🟥								✓	✓		✓	✓
Matrix Powers ($y_k=A^k x$)		🟥								✓	✓		✓	✓
CG	🟧	🟥								✓	✓		✓	✓
KSM/GMRES	🟧	🟥								✓	✓		✓	✓
SpLU	🟧	🟥												
Finite Difference Derivatives			🟥							✓	✓		✓	✓
FD/Laplacian			🟥							✓	✓	✓	✓	✓
FD/Gradient			🟥							✓	✓	✓	✓	✓
FD/Divergence			🟥							✓	✓	✓	✓	✓
FD/Curl			🟥							✓	✓		✓	✓
FD/Solve PDE, Explicit			🟥							✓	✓		✓	✓
FD/Solve PDE, Implicit Iter.	🟧		🟥							✓	✓		✓	✓
FD/Solve PDE, Multigrid	🟧		🟥							✓	✓		✓	✓
There are a number of other important structured grid methods including lattice Boltzmann, finite volume, and AMR, that we have yet to enumerate representative kernels for.														
Although even within our community unstructured grids are commonly used, we have yet to enumerate any concise representative kernels.														
1D FFT (complex→complex)					🟥					✓	✓		✓	✓
3D FFT (complex→complex)					🟥					✓	✓		✓	✓
Convolution	🟧				🟥					✓	✓		✓	✓
Solve PDE via FFT	🟧		🟥		🟥					✓	✓		✓	✓
2D N^2 Direct						🟥				✓	✓		✓	
3D N^2 Direct						🟥				✓	✓		✓	
2D N^2 Direct (with cut-off)						🟥				✓	✓		✓	✓
3D N^2 Direct (with cut-off)						🟥				✓	✓		✓	✓
2D Particle-in-Cell (PIC)			🟧			🟥								
3D Particle-in-Cell (PIC)			🟧			🟥								
2D Barnes Hut						🟥		🟧		✓	✓		✓	✓
3D Barnes Hut						🟥		🟧		✓	✓		✓	✓
2D Fast Multipole Method					🟧	🟥		🟧						
3D Fast Multipole Method					🟧	🟥		🟧				✓	✓	
Quasi-Monte Carlo Integration							🟥			✓	✓		✓	✓
EP Summation							🟥			✓	✓			✓
Graph traversal								🟥		✓	✓		✓	✓
Betweenness centrality								🟥		✓	✓		✓	✓
Integer Sort									🟥	✓	✓		✓	✓
100 Byte Sort									🟥	✓	✓		✓	✓
Spatial Sort									🟥	✓			✓	
Our kernel selection predominantly reflects scientific computing applications. There are numerous other application domains within computing whose researchers should enumerate their own representative problems. Some of the problems from other domains may be categorized using the aforementioned motifs, some may be categorized into other Berkeley Motifs not listed above (such as branch-and-bound, dynamic programming), while others may necessitate novel motif creation.														

Table 1: Brief Overview of enumerated kernels with their mapping to Dwarfs. Check marks denote progress we've made towards a practical testbed for scientific computing. Note, orange boxes denote the mapping of supporting kernels to dwarfs.

cies when mapped to existing languages or instruction set architectures. This ensures we neither restrict nor recast parallelism. Moreover, we minimize the expression of user-managed synchronization in the problem specification.

2.2 Reference Implementation

To provide context as to how such kernels productively map to existing architectures, languages and programming models, we have proceeded by attempting to produce a reference implementation for each kernel. As a reminder, these should be viewed as "hints" designed to show how other designers have mapped a problem's operands and operators to existing hardware and software. Since we wanted such implementations to be illustrative, we tried to ensure they were the most straightforward implementation in the easiest to understand languages using familiar architectures. To that end, most of the linear algebra-oriented computations are written in MATLAB using array indexing to process matrices, rather than one-line library calls to compute the same kernel. This ensures that the kernel's computation is explicit and readable in the implementation and not hidden behind a library.

Unfortunately, MATLAB has limitations such as awkward facilities for graphs and tree programming, and does not permit low-level control of computations. For these reasons, our reference implementations of kernels such as the Barnes-Hut n-Body solver were written in pure C without any supporting library computations.

2.3 Optimization Inspiration

There is a dramatic performance gap between the performance that can be attained via productive programming (the most natural means of implementing the problem using existing languages, programming models and hardware) and the style needed to elicit high performance. The discrepancy in performance should not be viewed as the programmer's failing. Rather, it should be viewed as a lighthouse for future research into architecture, languages, and middleware.

There are decades of optimizations for each of the kernels we have enumerated. As such, it would be wasteful to try and recreate all of them. We will compile a list of known top-performing algorithmic strategies and optimizations associated with each kernel, as well as document autotuners, libraries, and benchmarks that are representative of specific problem sizes and programming models/languages. However, for a small subset of the kernels, we created a reference optimized implementation designed to illustrate the most common optimization techniques in sequential computation (such as maximizing data locality). For example, the optimized reference implementations of the LU and QR factorizations take the LAPACK-style cache blocking and BLAS-3 aggregation optimizations and distill them into compact C implementations. In the future, we may explore a different direction and produce a series of optimized versions that rather than explore sequential optimizations, explore parallelization and synchronization techniques.

Ultimately, for each kernel, we will compile a list of previous optimization work likely to be relevant in the future.

2.4 Scalability

The last decade has not only seen an order-of-magnitude increase in inter-SMP parallelism, but also a more challenging explosion in intra-SMP parallelism via SIMD, hardware multithreading, and multiple cores. This ever increasing parallelism constrains the fixed problem size benchmarks into a strong scaling regime. Although this might be appropriate for some domains, it is rarely appropriate in the field of scientific computing where weak scaling has been used to solve problems at petascale. Similarly, such a constraint may drive designers to solutions that, although they may be quite appropriate for the benchmark, are utterly unscalable and inappropriate for future problem scales.

To that end, we have created a scalable problem generator to accompany each computation. In some cases this generator may be nothing more than a means of specifying problems using the underlying method's high-level description language. In other cases, code is written to create input datasets. In either case, the problem size is independent of implementation or mapping to architecture.

In the linear algebra world, inspired by the Linear Algebra working note [11], we generate randomized matrices for LU and QR on the fly adhering to certain conditions than enable factorization. We apply similar techniques in the spectral problems by specifying the FFT input size, but randomizing initial values. The n-Body computations can be scaled simply by increasing the number of particles, and the computational challenges seen in the particle-tree codes can be altered by changing the spatial layout of particles and forces used.

Unfortunately, scalable problem generation can be challenging for kernels in which the problem configuration or connectivity is specified in the data structure. Often, sparse linear algebra research has been focused on fixed size matrices that have resided in collections for decades. We believe this tradition must be evolved into a form like [15] so that repositories of scalable matrix generators exist. We acknowledge that this is not universally applicable as some matrices are constructed from discrete real-world phenomena like connectivity of the web.

2.5 Solution Verification

One could view the high level problem definition as a means to verify the validity of a solution. In effect, one can compare the results from two implementations (reference and HW/SW co-design), checking for discrepancies on a bit by bit granularity. Unfortunately, such an approach may not always be appropriate. As problem size scales, execution of the reference implementation may not be feasible. Moreover, such a verification regime assumes that implementation and verification code don't contain a common error (matching, but incorrect result). Finally, this approach assumes that there is one true solution. We wish to create a verification methodology that is in some sense orthogonal to the problem definition.

In many cases, we construct problems whose solutions are known a priori or can be calculated with minimal cost. We verify the symmetric eigensolver by constructing randomized matrices with known eigenvalues. To obtain such a matrix, one forms a diagonal matrix D composed of the desired eigenvalues and a randomized orthogonal matrix Q. The test matrix is the product $Q^T DQ$. This 'reverse diagonalization' produces a randomized matrix with pre-determined eigenvalues, the eigenvalues of which can be selected to be as numerically challenging or clustered as the user desires. Finite difference calculations are verified by evaluating a function (*e.g.* $sin(xy)$) both symbolically and via the finite difference method. We may then compare the grid at a subset of the sampled points. Similarly, the result of the Monte Carlo integration kernel can be compared to analytical or numerical results in any dimension.

2.6 Solution Quality

The quality of a solution is multifaceted. Thus far, our group has primarily taken the rather narrow focus of optimization: time or energy to solution given a fixed architecture. Unfortunately, given a set of programmers unrepresentative of the community as a whole (we pride ourselves in our knowledge of architecture and algorithms), we likely minimize the programming and productivity challenges required to attain such performance. In the end, the quality of a solution must take into account not only performance or energy, but must engage the programmer community to determine how productive the solution is. Moreover, the solution must be evaluated on its ability to integrate with existing software and hardware.

3 Related Work

There is abundant prior work on defining microbenchmarks (*e.g.* LINPACK [29] for peak floatingpoint performance, pChase [28] for memory latency, STREAM [22] for memory bandwidth), benchmarks for evaluating specific architectural and programming models (*e.g.* HPC Challenge [16] for MPI and distributed-memory systems, ParBoil [27] and Rodinia [8] for GPUs and CUDA, PARSEC [6] for cache-based multicore, SPLASH [30] for shared-memory systems, and STAMP [23] for transactional memory implementations), benchmarks that are focused on a particular application-space (*e.g.* ALPBench [20] and MediaBench [19] for multimedia applications, BioPerf [3] for computational biology algorithms, Lonestar [18] for graph-theoretic and unstructured grid computations, NU-MineBench [25] for data mining, PhysicsBench [38] for physics simulations, NAS Parallel benchmarks [5] for scientific computing, and the HPCS SSCA benchmark suite [4] for Informatics applications), and large benchmark consortia (*e.g.* SPEC [32] and EEMBC [13]). From our perspective, we view existing benchmarks as reference implementations of one or more kernels (since the problem size, programming language, and algorithms are typically fixed in the benchmark definition). In fact, the Rodinia and the Parallel Dwarfs project [26] teams adopt the Berkeley 13-motif classification to describe the underlying computation in each of their benchmarks.

While all the aforementioned benchmarks serve the computing research community well, their typical usage is to generate a single performance number corresponding to a benchmark-specific metric. Our intent with creating the kernel reference testbed is to drive hardware-software co-design, leading to innovative solutions that can be potentially applied across application domains. Hence we emphasize that our reference and optimized implementations are only hints on how problems should be solved.

3.1 Example Uses

Consider application of this methodology with the simplifying restrictions that we we must use existing hardware, languages, and compilers. We may either embrace existing programming models or develop new ones, but we wish to explore alternate implementations of a particular kernel. In essence, this is the core thesis behind modern auto-tuners [7, 10, 14, 17, 21, 33–37]. The auto-tuner can implement a large number of functionally-equivalent variants and simply benchmark them to find the best. As its free of legacy code, auto-tuners are free to explore novel data structures or execution models.

Recently, we've extended this approach to encompass architectural exploration [24]. Using the highly configurable Tensilica processor generator, we simultaneously explored the microarchitecture, memory architecture, and software optimization space in order to find the most performant, power-efficient, or cost-effective HW/SW solution.

We believe in the future, such a test best could be used to explore the other fields listed in Table 2.

Benchmark Style	Micro-arch.	Compilers	Instruction Set	SW Optimization	Prog. Models	Languages	Memory Arch.	Algorithms
Fixed Binary	✓							
Fixed Source Code	✓	✓	✓					
Fixed Interface, but may optimize code	✓	✓	✓	✓				
Code-based problem definition	✓	✓	✓	✓	✓			
High-level problem definition	✓	✓	✓	✓	✓	✓	✓	✓

Table 2: Fields of research enabled by different styles of Benchmarks

4 Summary

In this paper, we presented a methodology for defining and constructing a domain-oriented problem testbed. As an example, we applied this methodology to the field of scientific computing and constructed a novel reference testbed. Unlike previous benchmarking efforts, we believe this testbed will enable a much broader research effort (Table 2). Moreover, we believe it will facilitate collaboration between researchers from different fields.

The reference testbed contains a concise set of kernels selected to span the most common and defining algorithmic and computational aspects of each problem domain. By its very nature it is meant to be extendible. An essential element of the testbed is a supporting framework of methods for generating problem specifications such as input data sets, illustrating reference implementations, and solution verification procedures.

The flexibility of such a domain-oriented problem testbed challenges researchers by removing a single fixed code base as starting point for iterative improvements in existing hardware or software solutions. Any *validated* hardware/software solution becomes a possible starting point. However, the drawback of having to select and possibly implement such a solution is greatly outweighed by the potential benefits of finding new co-designed hardware and software solutions not easily obtainable with traditional fixed code benchmark sets.

Acknowledgments

This work was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., and Yelick, K. The landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, University of California, Berkeley, Dec. 2006.

[2] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Lee, E., Morgan, N., Necula, G., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. The parallel computing laboratory at U.C. Berkeley: A research agenda based on the Berkeley view. Tech. Rep. UCB/EECS-2008-23, University of California, Berkeley, Mar. 2008.

[3] Bader, D., Li, Y., Li, T., and Sachdeva, V. BioPerf: a benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *Proc. IEEE Int'l. Symp. on Workload Characterization (IISWC)* (Oct. 2005), pp. 163–173.

[4] Bader, D., Madduri, K., Gilbert, J., Shah, V., Kepner, J., Meuse, T., and Krishnamurthy, A. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *Cyberinfrastructure Technology Watch* (Nov. 2006).

[5] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. The NAS parallel benchmarks. *Int'l. Journal of High Performance Computing Applications 5*, 3 (1991), 63–73.

[6] Bienia, C., Kumar, S., Singh, J., and Li, K. The PARSEC benchmark suite: Characterization and architectural implications. Tech. Rep. TR-811-08, Princeton University, Jan. 2008.

[7] Chandramowlishwaran, A., Williams, S., Oliker, L., Lashuk, I., Biros, G., and Vuduc, R. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)* (Atlanta, Georgia, 2010).

[8] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. IEEE Int'l. Symp. on Workload Characterization (IISWC)* (Oct. 2009), pp. 44–54.

[9] Colella, P. Defining software requirements for scientific computing, 2004. DARPA HPCS presentation.

[10] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. Stencil Computation Optimization and Auto-Tuning on State-of-the-art Multicore Architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), IEEE Press, pp. 1–12.

[11] Demmel, J., Marques, O., Parlett, B., and Vömel, C. A testing infrastructure for LAPACK's symmetric eigensolvers. Tech. Rep. 182, LAPACK Working Note, Apr. 2007.

[12] Dubey, P. A platform 2015 workload model: Recognition, Mining and Synthesis moves computers to the era of Tera. Tech. rep., Intel Corporation, 2005.

[13] The embedded microprocessor benchmark consortium. `http://www.eembc.org`.

[14] Frigo, M., and Johnson, S. G. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing* (1998), vol. 3, IEEE, pp. 1381–1384.

[15] Gahvari, H., Hoemmen, M., Demmel, J., and Yelick, K. Benchmarking Sparse Matrix-Vector Multiply in Five Minutes. In *SPEC Benchmark Workshop* (January 2007).

[16] HPC Challenge benchmark. `http://icl.cs.utk.edu/hpcc/`.

[17] Kamil, S., Chan, C., Oliker, L., Shalf, J., and Williams, S. An auto-tuning framework for parallel multicore stencil computations. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)* (Atlanta, Georgia, 2010).

[18] Kulkarni, M., Burtscher, M., Cascaval, C., and Pingali, K. Lonestar: a suite of parallel irregular programs. In *Proc. IEEE Int'l. Symp. on Performance Analysis of Systems and Software (ISPASS)* (Apr. 2009), pp. 65–76.

[19] Lee, C., Potkonjak, M., and Mangione-Smith, W. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. 30th ACM/IEEE Int'l. Symp. on Microarchitecture (MICRO 30)* (Dec. 1997), pp. 330–335.

[20] Li, M.-L., Sasanka, R., Adve, S., Chen, Y.-K., and Debes, E. The ALPBench benchmark suite for complex multimedia applications. In *Proc. IEEE Int'l. Symp. on Workload Characterization (IISWC)* (Oct. 2005), pp. 34–45.

[21] Madduri, K., Williams, S., Ethier, S., Oliker, L., Shalf, J., Strohmaier, E., and Yelick, K. Memory-efficient optimization of gyrokinetic particle-to-grid interpolation for multicore processors. In *Proc. SC2009: High performance computing, networking, and storage conference* (2009).

[22] McCalpin, J. Memory bandwidth and machine balance in current high performance computers. *IEEE CS TCCA Newsletter* (Dec. 1995), 19–25.

[23] Minh, C., Chung, J., Kozyrakis, C., and Olukotun, K. STAMP: Stanford transactional applications for multi-processing. In *Proc. IEEE Int'l. Symp. on Workload Characterization (IISWC)* (Sept. 2008), pp. 35–46.

[24] Mohiyuddin, M., Murphy, M., Williams, S., Oliker, L., Shalf, J., and Wawrzynek, J. A design methodology for domain-optimized, power-efficient supercomputing. In *Proc. SC2009: High performance computing, networking, and storage conference* (2009).

[25] Narayanan, R., Ozisikyilmaz, B., Zambreno, J., Memik, G., and Choudhary, A. MineBench: A benchmark suite for data mining workloads. In *Proc. IEEE Int'l. Symp. on Workload Characterization (IISWC)* (Oct. 2006), pp. 182–188.

[26] Parallel dwarfs. `http://paralleldwarfs.codeplex.com/`.

[27] The Parboil benchmark suite. `http://impact.crhc.illinois.edu/parboil.php`.

[28] Pase, D. The pChase memory benchmark page. `http://pchase.org/`.

[29] Petitet, A., Whaley, R., Dongarra, J., and Cleary, A. HPL - a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. `http://www.netlib.org/benchmark/hpl/`.

[30] Singh, J., Weber, W.-D., and Gupta, A. SPLASH: Stanford parallel applications for shared-memory. *SIGARCH Comput. Archit. News 20*, 1 (1992), 5–44.

[31] Sort benchmark home page. `http://sortbenchmark.org/`.

[32] SPEC benchmarks. `http://www.spec.org/benchmarks.html`.

[33] SPIRAL Project. `http://www.spiral.net`.

[34] VUDUC, R., DEMMEL, J., AND YELICK, K. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series* (June 2005), Institute of Physics Publishing.

[35] WHALEY, R. C., PETITET, A., AND DONGARRA, J. Automated Empirical Optimization of Software and the ATLAS project. *Parallel Computing 27(1-2)* (2001), 3–35.

[36] WILLIAMS, S., CARTER, J., OLIKER, L., SHALF, J., AND YELICK, K. Lattice Boltzmann simulation optimization on leading multicore platforms. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)* (Miami, Florida, 2008).

[37] WILLIAMS, S., OLIKER, L., VUDUC, R., SHALF, J., YELICK, K., AND DEMMEL, J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. SC2007: High performance computing, networking, and storage conference* (2007).

[38] YEH, T., FALOUTSOS, P., PATEL, S., AND REINMAN, G. Parallax: an architecture for real-time physics. *SIGARCH Comput. Archit. News 35*, 2 (2007).