Integrated Inference for Hardware-Software Efficiency: A Case Study in SpMV and Smart Memories

Benjamin C. Lee [1] *Mark A. Horowitz* [2]

Systems Architecture Integration Laboratory
Duke University
Technical Report No. TR-10-01

August 30 2010

[1] Electrical and Computer Engineering
Duke University
Durham, North Carolina 27708

[2] Electrical Engineering
Stanford University
Stanford, California 94305

Abstract

Energy efficiency is the fundamental challenge in computing. Dennard scaling has stopped, which means that Moore's Law provides more transistors but power densities increase with integration. These power densities, combined with Amdahl's Law, will also limit the efficiencies and tractability of future multi-core integration. Without process and parallelism to drive efficiency, we must rely on customization and integrated design. However, customization has become prohibitively expensive, primarily due to the challenge of integrated software and hardware design. Customization is facilitated by recent advances in software and hardware generators, which constrain a design space, parameterize the remaining degrees of freedom, and automatically produce functional implementations for any combination of parameter values.

To address these challenges, we propose using generators more effectively by creating an integrated design framework that synthesizes key interactions between hardware and software. We demonstrate a proof of concept for sparse matrix-vector multiply (SpMV) on an embedded processor hardware base, by using statistical regression modeling. With models that capture the highly non-monotonic SpMV performance topologies, we perform integrated optimization to demonstrate a performance gain of 5.0x (Mflop/s) while reducing the energy costs per operation by 10 percent (0.9x nJ/Flop).

1 Introduction

Energy efficiency is the defining challenge in modern computing. Mechanisms that delivered efficiency in the past will face fundamental limitations in the future. While Moore's Law projected increasing transistor densities, Dennard scaling provided guidelines for leveraging those densities at reasonable power: as device feature size F scaled, voltage V_{dd} and current I decreased linearly, power (IV_{dd}) decreased quadratically, and power density remained constant [5]. However, leakage current constrains threshold voltage scaling, which in turn constrains V_{dd} scaling. Although V_{dd} stopped scaling at 130nm [12], Moore's Law continued, leading to smaller devices and increased power densities.

Increasing power densities imply fundamental limits to multi-core integration, which promises energy efficiency via a larger number of less powerful

cores. Although Moore's Law and smaller transistors may enable more cores on a single chip, limits of Dennard scaling and increasing power densities make supplying power to those cores difficult. Moreover, Amdahl's Law and the challenges of extracting parallelism constrain the number of small cores useful in an application [11]. Thus, multi-core integration will experience rapidly diminishing marginal returns in energy efficiency due to constraints from both process technology and software parallelism.

Without technology and parallelism to drive efficiency, computing must rely on a third mechanism: customization and integrated design. Application specific integrated circuits (ASIC's) are several orders of magnitude more efficient than high-performance, general purpose architectures. However, customization has become prohibitively expensive, requiring long design times that incur high non-recurring engineering costs. These costs arise, primarily, from the challenge of integrated software and hardware design. While application experts possess deep domain knowledge and hardware engineers possess a broad array of design options, bridging the hardware/software interface produces an intractable number of degrees of freedom in the design.

These costs motivate generators, which constrain a design space, parameterize the remaining degrees of freedom, and automatically produce functional implementations with any combination of parameter values. Such generators have proven effective for both application tuning [26] and processor soft-cores [22]. However, these generators are restricted to one side or the other of the hardware/software interface. To use these generators most effectively, we must consider an integrated framework that synthesizes the key interactions between hardware/software and uses them to perform integrated optimization. Only with such an integrated framework can we quantify efficiencies from software, hardware, and the synergies between the two.

This paper outlines opportunities and challenges when integrating the use of software/hardware generators (Section 2). We demonstrate a proof of concept for sparse matrix-vector multiply SpMV, embedded processor architectures, and statistical regression modeling (Section 3). Statistical inference captures the highly non-monotonic SpMV performance topology as a function of application structure and processor architecture with less than 5 percent error (Section 4). Performance and energy models drive optimization, which lead to the following contributions (Section 5):

- **Application Efficiency:** Mechanisms that improve application performance are likely to improve energy efficiency. Better performance

through locality reduces energy overheads from data movement and reduce energy per useful operation. For SpMV, better locality improves Mflop/s by 1.6x while improving nJ/Flop by 0.6x through fewer expensive off-chip memory accesses.

- **Architecture Efficiency:** Architecture requires the traditional trade-offs between performance and energy. Larger resources compensate for poor application locality but incur a net increase in energy. For SpMV, additional resources improve Mflop/s by 2.7x but also increases nJ/Flop by 1.5x.

- **Integrated Efficiency:** Both application and architecture optimization are required for efficiency. Application tuning maximizes the utility of additional architectural resources, thereby amortizing their energy overheads over a larger number of useful operations. For SpMV, simultaneous optimization enables a 5.0x increase in Mflop/s with net reduction in energy of 0.9x nJ/Flop.

- **Serialized Optimization:** While simultaneous optimization across application and architecture parameters is ideal, serialized optimization incurs modest penalties for SpMV. These penalties are more likely if the architecture is optimized before the application. Moreover, such penalties may increase as parameter choice exerts greater impact on efficiency. In SpMV, the greatest penalties of approximately 20 percent were observed for the most aggressively tuned matrices.

Collectively, these results motivate new thinking on computing efficiency that spans the hardware/software interface. Architects cannot afford to ignore application tuning, a part of the abstraction layer where performance and energy efficiency are correlated. Application designers increasingly rely on hardware knowledge to extract performance. Integrating hardware/software design enables efficiency through customization.

2 Generators and Customization

Generators reduce the cost of customization, which include long design times and incur high non-recurring engineering costs. They apply judicious constraints on the software and hardware design spaces to limit the degrees of freedom and to reduce the scope of analysis. Today's generators provide two

parallel flows, one for software and one for hardware. We propose wrapping these generators in a third flow to perform inference and optimization across the hardware/software interface. Thus, automation and inferential modeling enable tractable optimization over the remaining degrees of freedom.

Software Generators. A choice of algorithm and kernel constrains the space of software implementations. Code parameterization takes the remaining degrees of freedom and creates a design space such that an instantiation of any point in the space can be automatically generated. For example, given a sparse matrix kernel with a defined data structure (e.g., compressed sparse row), a parameterized code generator automatically reorganizes that basic data structure into blocks of various sizes with various locality benefits. In addition to parameters that enhance performance and energy, codes might also parameterize precision, using narrow width operands or fewer iterations in an iterative heuristic, to reduce energy at the expense of accuracy.

With this parameterization, code generators automatically generate any point in the parameterized space. Drawing from experiences in related work, code generation will likely be domain-specific since domain expertise is required to construct a reference implementation. Furthermore, most of these systems use a source-to-source compiler. For example, generators in numerical linear algebra parameterize and extend a baseline code by generating C code using C code (e.g., ATLAS [29], OSKI[26]) or generating C++ code from a meta-language (e.g., PetaBricks [1]). Sketching compilers combine a reference code and an incomplete sketch of a variant code to infer a complete implementation of the variant code [21]. In future, any integrated hardware/software framework should be extensible enough to leverage a mix of these or other generator strategies.

Moreover, these generators should include profiling and verification collateral. Profiling provides insight into the generated code, which will be required for integrated inference and optimization. Code verification is increasingly important as generators begin to explore trade-offs in precision and energy efficiency. Verification collateral quantifies accuracy and assesses its impact on, for example, numerical stability, which may affect convergence properties of iterative solvers or heuristics.

Hardware Generators. Hardware generators implement a flow that parallels the one for software. The generator defines a system, which constrains hardware components and their points of interaction. Like the software generators, this system is likely to be effective for a particular application domain, and will use architecture parameterization to define/enumerate remaining degrees of freedom.

Execution model, instruction fetch, processor functional units, and mem-

ory hierarchy could be parameterized, enabling customization for applications with varying degrees of parallelism and locality. For example, one can exploit parallelism through superscalar units, speculative execution, SIMD, and VLIW. The cache hierarchy could be parameterized using normal cache parameters (line/fetch size, associativity, size, levels, etc) to optimize for varying kinds of locality and bandwidth requirements.

The hardware generator should be capable of automatically generating any point in the parameterized hardware space. How the system generates each instance depends on how the generator is written. Just as software leverages source-to-source compilers, hardware might leverage RTL-to-RTL transformations. Alternatively, we might automatically generate code in a high-level language (e.g., SystemC [9], BlueSpec [2]), which then maps to RTL. Another option might be a parameter generator for soft system components (e.g.,Tensilica [23, 8]) to generate the required RTL. Any of these methods will work in our integrated hardware/software framework as long as they provide RTL, performance and energy simulators, and validation collateral.

Integrated Inference. Software and hardware generators provide performance profilers and simulators, but relying solely on them for optimization is prohibitively expensive. An integrated hardware/software space could easily contain hundreds of thousands of designs. Simulation quickly becomes intractable. Inference addresses these problems by constructing predictive models. Inference sparsely samples designs from the joint software and hardware space, measuring the performance and power of these samples using automatically generated profilers or simulators. With these measurements, models are constructed to predict design metrics as a function of the hardware/software design parameters. While such strategies have been successfully applied to processor architecture [6, 13, 15], customization requires broadening the scope of inference to include application design parameters.

Inferential models, constructed from the joint hardware/software space, enable joint optimization. they can be used for exhaustive search in small spaces or with global optimizers (e.g., hill-climbing, genetic search) in larger spaces. By traversing the joint hardware/software space, the optimizer identifies particular application designs customized for particular architecture designs and vice versa. Thus, software/hardware generators automate the crank-tuning for software and hardware design, respectively, while inferential models from detailed simulation makes integrated optimization computationally tractable.

3 Proof of Concept

We demonstrate a proof of the integrated generators concept for a particular application and architecture. A software generator produces code variants for sparse matrix-vector multiply (SpMV) while a hardware generator produces variants for an embedded processor supported by a parameterized memory hierarchy. Integrated statistical inference constructs regression models to capture performance and energy relationships between the application and architecture. While we leverage previously proposed components for software/hardware generators to prove the concept, we are the first to combine the application of hardware/software generators with inferential models to create an end-to-end customization framework.

3.1 Sparse Matrix-Vector Multiply

Sparse matrix-vector multiply (SpMV) computes $y = y + Ax$ when A is a sparse matrix (i.e., most elements in A are zero). We refer to x and y as the source and destination vectors, respectively. The central problem in efficient performance tuning for sparse computational kernels, such as SpMV, is the considerable irregularity and variation in the best of choice of sparse matrix data structure and code transformations across machines and matrices. Such variation produces a non-monotonic performance topology in which naive optimization heuristics likely get caught in local optima.

Despite matrix sparsity, dense sub-structure may exist. Blocking is a technique to improve reuse and locality over a conventional implementation of SpMV by reorganizing the matrix data structure to exploit sequences of naturally occurring dense blocks. Such blocking reduces loop overhead, reduces indexing overhead, reduces access irregularity, and increases temporal locality to the source vector. As illustrated in Figure 1, a sparse matrix may be reorganized into sub-blocks of size $r \times c$; only those blocks which contain at least one non-zero are stored. SpMV computation proceeds block-by-block. Within each block, the code reuses the corresponding c elements of the source vector and streams through the r elements of the destination vector.

As illustrated in Figure 2 for a 2×2 blocking, we consider a compressed storage format for the blocked sparse matrix (BCSR). Within the array of matrix elements, elements within the same block are stored consecutively and blocks within the same block row are stored consecutively, in row-major order. Moreover, blocked storage requires fewer row and column indices as

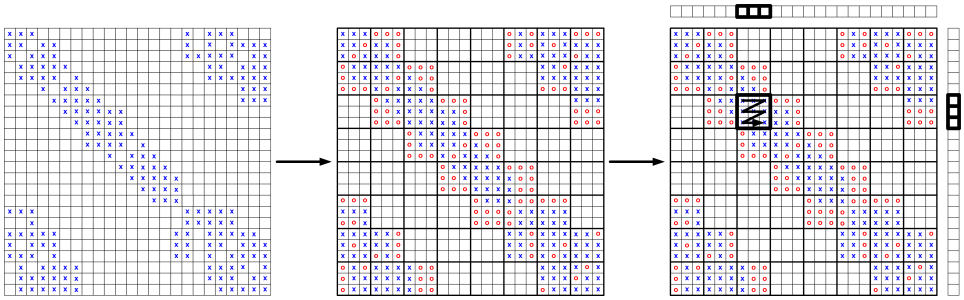

Figure 1: Register Blocking: A sparse matrix with non-zero elements (blue-x) is reorganized into 3×3 blocks. Matrix sparsity requires filling incomplete 3×3 blocks with explicit zero values (red-o), inducing dense sub-structure but incurring storage and computational overheads for zero values. Blocking improves access locality to source and destination vectors.

these indices point to the row and column location for each block of elements instead of pointing the location for each individual element. Thus, blocking reduces storage overheads associated with matrix sparsity.

However, blocking also incurs overheads as imposing a uniform matrix block size may require the storage of explicit zeros. We define the fill ratio as the number of stored values (original non-zeros plus explicit zeros) divided by the original number of non-zeros. Filled zeros require storage overhead in the array of matrix of values and also require explicit computation (i.e., unnecessary Flops). The benefit/cost analysis of a block size depends on the locality and index storage benefits relative to filled Flops and value storage benefits. These effects, in turn, are highly dependent on matrix structure. Moreover, these effects interact with cache structure and memory bandwidth to determine SpMV performance and energy efficiency.

Thus, we define a BCSR-based kernel for SpMV, which enables the parameterization of matrix block sizes (r and c). SpMV code for each block size is generated automatically by OSKI, an optimized library generator for sparse linear algebra [26]. In addition to generating code, OSKI provides profiling collateral by computing matrix fill ratios under different block sizes and provides verification collateral by ensuring each blocked variant of SpMV computes the correct result. We evaluate generated code for the matrices of Table 1 drawn from a variety of application domains.

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 & a_{14} & a_{15} \\ 0 & 0 & a_{22} & 0 & a_{24} & a_{25} \\ 0 & 0 & 0 & a_{33} & a_{34} & a_{35} \end{pmatrix}$$

$$\texttt{b\_row\_start} = (\ 0 \ \ 2 \ \ 4 \ )$$

$$\texttt{b\_col\_idx} = (\ 0 \ \ 4 \ \ 2 \ \ 4 \ )$$

$$\texttt{b\_value} = (\ a_{00} \ \ a_{01} \ \ a_{10} \ \ a_{11} \ \ 0 \ \ 0 \ \ a_{14} \ \ a_{15} \ \ a_{22} \ \ 0 \ \ 0 \ \ a_{33} \ \ a_{24} \ \ a_{25} \ \ a_{34} \ \ a_{35} \ )$$

Figure 2: BCSR with 2×2 blocks. 2×2 blocks are stored contiguously in `b_value`. The first column index of entry (1,1) in each 2×2 block is stored in `b_col_idx`. Pointers to block row starting positions in `b_col_idx` are stored in `b_row_start`.

	Matrix	Dimension	Non-Zeros	Sparsity	Application Domain
1	3dtube	45330	1629474	7.93E-04	3-D pressure tube simulation
2	bayer02	13935	63679	3.28E-04	chemical process simulation
3	bcsstk35	30237	740200	8.10E-04	stiff matrix automobile frame analysis
4	bmw7st	141347	3740507	1.87E-04	automobile body analysis
5	crystk02	13965	491274	2.52E-03	crystal free vibration, finite element method
6	memplus	17758	126150	4.00E-04	circuit simulation
7	nasasrb	54870	1366097	4.54E-04	shuttle rocket booster simulation
8	olafu	16146	515651	1.98E-03	numerical method accuracy improvement
9	pwtk	217918	5926171	1.25E-04	pressurized wind tunnel simulation
10	raefsky3	21200	1488768	3.31E-03	fluid structure interaction
11	venkat01	62424	1717792	4.41E-04	flow simulation

Table 1: Matrices: Dimension quantifies N in square matrix. Sparsity is non-zeros divided by N^2.

3.2 Tensilica and Smart Memories

Embedded processors for high-performance computing have attracted recent interest for their energy efficiency [28]. Such processors are well-suited for scientific applications and numerical methods where Flop to data bandwidth ratios are low. Lower-power, embedded processors balance the system, reducing the traditional gap between high processor Flop rates and restricted memory bandwidth. This work considers the Tensilica Xtensa, which implements a 32-bit RISC ISA with 24-bit instructions and windowed general-purpose register file with 16 registers per window.

Custom functional units may be added to the Xtensa core using instruction set extensions defined by the user with the Tensilica Instruction Extension (TIE) language [27]. The TIE compiler generates a customized processor, simulator, and verification collateral for the modified processor. This work uses several pre-defined Tensilica microarchitectural structures, including a 32-bit integer multiplier and divider, 32-bit floating-point unit, 64-bit floating-point accelerator. Moreover, the core supports VLIW instruction formats that allow up to 3 instructions per cycle using the Tensilica Flexible Length Instruction eXtension (FLIX) framework [14]. The format supports up to two slots for floating-point operations and one slot for loads/stores.

Although Xtensa parameterizes the datapath, it implements constrained cache and memory subsystems. Thus, we use an integrated simulator that combines the Xtensa datapath with the more flexible Smart Memories simulator [23]. Within Smart Memories, we evaluate a system with separate L1 data and instruction caches supported by a 3.2GB/s interface to main memory. Combined, the two infrastructures provide a comprehensive architectural parameterization.

We derive energy models for this architectural space through a combination of sources for core, cache, and memory. Power for a 400MHz/45nm Xtensa core is derived from Tensilica Xplorer Integrated Design Environment, extrapolated from a physical implementation of a 600MHz/90nm Xtensa core, and scaled based on activity reported by the integrated Tensilica-Smart Memories simulator. Cache power is reported from CACTI models [19] based on miss rates reported in simulation. Lastly, we use Micron data sheets to estimate the cost of accessing off-chip memory based on L1 cache miss rates reported in simulation [17].

SpMV characteristics guide our architectural parameterization and its emphasis on the cache and memory subsystem. Despite using blocked matrices, SpMV is memory bound as matrix elements stream through the system.

	Parameter	Range	Baseline
	SpMV		
x_1	brow, block row	1 :: 1+ :: 8	1
x_2	bcol, block column	1 :: 1+ :: 8	1
x_3	fR, fill ratio	function of brow,bcol,matrix	1.0
	Cache Architecture		
y_4	lsize, line size	16B :: 2x :: 128B	32B
y_5	dsize, data size	4KB :: 2x :: 256KB	32KB
y_6	dways, data ways	1 :: 2x :: ::8	2
y_7	drepl, data repl	LRU, NMRU, RND	LRU
y_8	isize, inst size	2KB :: 2x :: 128KB	16K
y_9	iways, inst ways	1 :: 2x :: ::8	2
y_{10}	irepl, inst repl	LRU, NMRU, RND	LRU

Table 2: Joint Parameter Space. SpMV block sizes and cache architecture parameters.

SpMV performance is determined primarily by data locality for elements in the source and destination vector. While matrix blocking exposes locality, the degree to which locality translates into performance depends on L1 cache design. Because most memory traffic is attributed to SpMV streaming the matrix from memory with no reuse on matrix elements, an L2 cache only marginally benefits this application.

Thus, we consider a system comprised of a single Xtensa processor with in-order execution and VLIW support. Within this system, we focus on the parameterization of the L1 cache, which impacts the interface to memory (Table 2). We consider line size, cache size, associativity, and replacement policies. Tensilica generates processor RTL, simulators, and verification collateral. Smart Memories generates RTL, simulators and verification collateral for a reconfigurable cache hierarchy (i.e., post-fabrication programmability). However, in this work, we parameterize and explore the space pre-fabrication, expecting only a subset of Smart Memories functionality to be fabricated in any final design.

3.3 Statistical Regression Modeling

Techniques in statistical inference and regression modeling reveal performance and power trends from sparsely simulated samples, enabling tractable optimization. In particular, we apply the approach proposed by Lee and Brooks for spline-based regression models [15]. In contrast to prior work,

which modeled only the microarchitectural space, we construct integrated models with independent parameters from both the application and the system architecture. With this integration, optimizers can explore the interactions across the hardware/software interface.

Given observed samples from a space of interest, models infer broader trends for unobserved points in the space. In the context of hardware/software design, we refer to z as the design metric of interest (e.g., performance, energy). $z = F(\vec{x}, \vec{y})$ predicts this metric as a function of application parameters $\vec{x} = (x_1, \ldots, x_p)$ and architecture parameters $\vec{y} = (y_{p+1}, \ldots, y_q)$.[1]

Evaluating F with profilers and simulators is a common approach, but their computational costs often hinder the optimization process. These difficulties motivate surrogates $\hat{F}$ that predict the response $z = \hat{F}(\vec{x}, \vec{y}) + \varepsilon$ with some approximation error ε. We construct the surrogate using regression models.

Within a regression framework, the models gets the flexibility to model non-linearities using splines. Splines transform the parameters to construct a model $z = \hat{F}(S_x(\vec{x}), S_y(\vec{y})) + \varepsilon$. The transformation $S_x(\vec{x})$ apply splines to each x_i in $\vec{x}$ by dividing the domain of x_i into intervals joined at intersections called knots. Equation (1) illustrates a cubic spline on x_i with three knots at a, b, and c. Note $(u)_+ = u$ if $u > 0$ and $(u)_+ = 0$ otherwise.

$$\begin{aligned} S(x_i) &= \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \beta_3 x_i^3 \\ &\quad + \beta_4 (x_i - a)_+^3 + \beta_5 (x_i - b)_+^3 + \beta_6 (x_i - c)_+^3 \end{aligned} \tag{1}$$

We account for interactions between parameters with product terms in the model.[2] For example, the performance impact of a block row size x_1 depends on the choice of block column size x_2. This interaction would be captured by adding another parameter into the model $x_{1,2} = x_1 x_2$. Moreover, the locality benefits of a particular block size depend on how often a block is split across cache lines, suggesting a third-order interaction $x_{1,2,3} = x_1 x_2 y_4$ should also be specified. We perform significance testing to explore whether a candidate interaction actually contributes to fit.

[1] In this work, x_1, x_2, x_3 are the matrix block sizes and $y_4, \ldots, y_{10}$ are the cache parameters of Table 2.

[2] Interactions are illustrated in a model $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$ by computing partial derivatives; $\delta y / \delta x_1$ is a function of x_2 and vice versa.

4 SpMV and Integrated Inference

We demonstrate integrated inference for SpMV, a sparse and irregular kernel characterized by non-monotonic performance and energy topologies, which has traditionally made predictive modeling very difficult. However, by using statistically significant interactions of application and architecture parameters, we can construct models that accurately estimate absolute and relative metrics, as well as accurately capture metric non-monotonicity.

We construct a separate model for every matrix. For each matrix, we perform inference to model performance, measured in true floating point operations per second (Flop/s $= 2 * nnz/T$) This measure of Flop/s considers the original number of Flops required for SpMV, which is the number of non-zero matrix elements (excluding explicitly filled zeros) times two (each element requires a multiply and add). Flop count is then divided by execution time (including explicitly filled zeros). Thus, our metric counts the amount of useful work divided by total execution time, which includes blocking overheads. Otherwise, Flops for explicitly filled zeros would inflate performance. We also perform inference to model average power, measured in mW and computed by dividing consumed energy by execution time.

4.1 Exploratory Data Analysis

We sample, sparsely and uniformly at random, 400 designs from the joint application and architecture space of 1.8M points defined in Table 2. With these samples, we perform exploratory data analysis to identify relationship between design metrics and parameters.

Figure 3 performs an association analysis, plotting parameter values against observed performance (MFfop/s) for an illustrative matrix, raefsky3. An analogous analysis may be applied to power. For each parameter value, the figure reports the average Mflop/s rate across all samples with that parameter value. For example, 54 of 400 samples were measured with a block row size (`brow`) of 8 and the average performance of these samples is 74 Mflop/s. Note that, due to averaging, the range of performance values on the x-axis do not illustrate the full range of sampled performance; the worst and best sample produced 14 and 166 Mflop/s, respectively.

The association analysis illustrates broad relationships between parameters and performance. Matrix blocking has a direct impact on performance but the performance trend is discontinuous. While the best block row size is 8, performance does not increase monotonically with block row size; 6

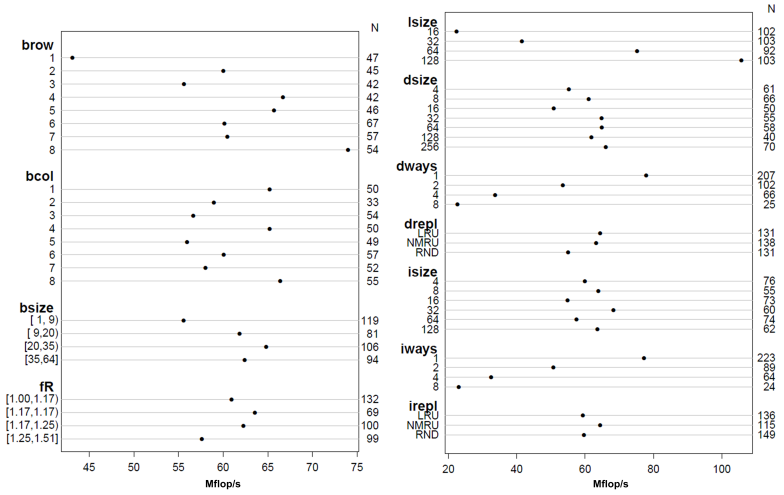

Figure 3: Association analysis for SpMV (L) and cache parameters (R). Parameter values plotted against sampled performance. Each performance point is an average over all samples with a particular parameter value.

or 7 block rows are only as effective as 2 block rows. Similarly, 1, 4, and 8 block columns deliver comparable performance, reflecting a dense matrix sub-structure that occur in multiples of 4. Performance increases with block size as larger blocks achieve greater locality with small penalties arising from filling explicit zeros. However, we observe diminishing marginal returns as larger block sizes (`brow`×`bcol`) may require a greater number of explicitly filled zeros, which require extra Flops and additional indexing overhead.

Cache structure also significantly impacts performance. We observe strong monotonic relationships between line size and performance. A bigger line size amortizes off-chip latency over a larger number of bytes in the cache line, which effectively increases streaming bandwidth for the matrix. A larger data cache size improves performance to a lesser degree as only the source and destination vectors exhibit re-use and locality.

Re-use distance and locality directly impact data cache associativity and the amount of useful data in the cache. Ideally, matrix blocks would not be cached since they are never re-used after making their contribution to the destination vector.However, in a highly associative cache, matrix blocks occupy cache lines longer as they must travel down the LRU stack before becoming a candidate for replacement. While source vector elements are re-used, the re-use distance across matrix blocks, not block elements, is high.

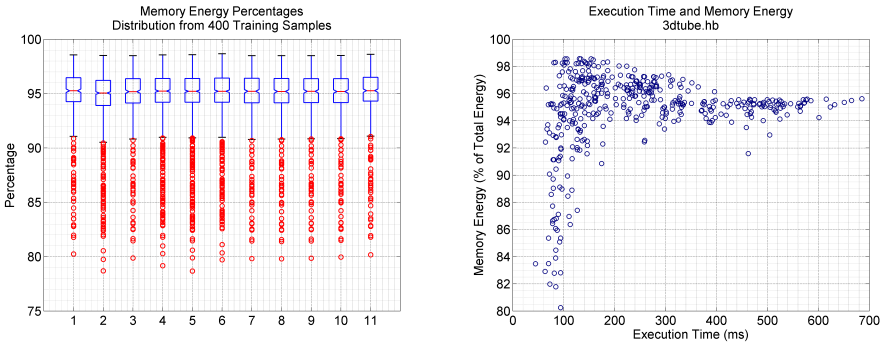

Figure 4: Memory energy (L) and its relationship with execution time (R).

Source vector elements accessed for one matrix block will, at the earliest, be accessed again for the next row's block located at the same column. However, a block may not exist at that location given matrix sparsity. Thus, the re-use distance may be proportional to matrix dimension divided by block size, subject to the distribution of non-zeros. This distance is likely larger than the distance a fully associative cache could support and direct-mapped caches are favored.

Data from training samples indicates total energy is dominated by memory accesses. Figure 4 considers memory energy as a percentage of total energy across the 400 samples. At the median, memory energy accounts for more than 95 percent of total energy. However, we observe a range of outliers with memory accounting for as little as 80 percent of total energy. These outlying designs reduce memory energy by improving performance, improving locality, and reducing the number of memory accesses through matrix blocking and cache structure. Thus, Figure 4R plots the designs in memory energy and execution time coordinates, illustrating the connection between energy and high-performance designs.

4.2 Model Specification

The model originally specifies a large number of terms and parameter interactions. Significance testing then prunes these terms by determining the likelihood a regression coefficient in a model should actually be zero. If a coefficient is highly unlikely to be zero, the parameter associated with that coefficient is significant. For example, consider a model $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \varepsilon$. Testing the significance of x_1 requires testing the null

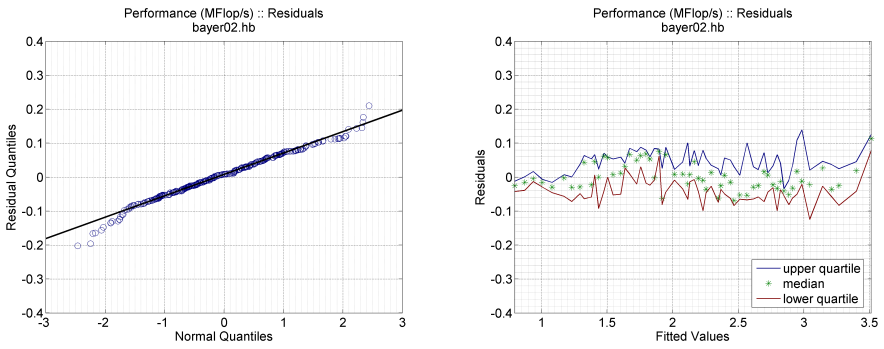

Figure 5: Normality (L) and variance (R) checks. Normality is checked by plotting residuals ε against samples from a Normal distribution. Constant variance is checked by plotting residuals ε against fitted values.

hypothesis $H_0 : \beta_1 = \beta_3 = 0$ with two degrees of freedom.

The F-statistic assumes an unbiased model. This means residuals $\varepsilon \sim N(0, \sigma^2)$ follow a normal distribution with constant variance.[3] Our models satisfy this criteria as shown by Figure 5, which plots residuals against samples from a Normal distribution. Plotting residuals against fitted performance (in this case, log(MFlop/s)) indicate that residuals vary around zero independently of fitted performance. Collectively, this analysis indicates the derivation process provides unbiased models.

Given unbiased models, we perform F-tests to determine interaction significance. Table 3 first quantifies the significance of grouped application and architecture parameters. The table then considers specific interactions between matrix block sizes and data cache parameters. Of these interactions, block and line size interaction is most significant with a p-value of 1.5E-13. In particular, the 5 degrees of freedom used to capture block and line size interaction are as significant as the 54 degrees of freedom used to capture all SpMV parameters. Also important are the 5 degrees of freedom used to capture interactions between block size and data cache associativity. Thus, significance testing indicates block and line size are the main point of interaction across the application/architecture interface.

We use the results of significance testing to construct the following model, described in R syntax [10]. This model fits log-performance. A power model is constructed by replacing the `Mflops` response with one for power.

[3]Residuals are the difference between true and fitted values from training data.

	Predictor	k	F-test	P-value
Grouped App/Arch Parameters				
1	SpMV	-54	3.99	1.24E-14
2	Cache	-46	45.34	2.20E-16
Grouped Cache Parameters				
3	Data cache	-37	19.18	2.20E-16
4	Inst cache	-37	1.10	3.21E-01
Data Cache Interactions				
5	bsize x lsize	-5	15.56	1.50E-13
6	bsize x drepl	-8	0.76	6.38E-01
7	bsize x dways	-5	2.20	5.46E-02
8	bsize x dsize	-6	1.55	1.61E-01

Table 3: Significance testing. k refers to the number of model terms used by a set of parameters. P-value denotes the probability the parameters are insignificant.

```
model.spmv = (log(Mflops) ~ (## first-order
                                rcs(brow,5) + rcs(bcol,5) + rcs(bsize,5) + rcs(fR,5)
                                + rcs(lsize,3) + rcs(dsize,4) + rcs(dways,3) + drepl
                                + rcs(isize,4) + rcs(iways,3) + irepl
                             ## second-order: block effects
                                + rcs(bsize,5) %ia% rcs(fR,5)
                                + rcs(bsize,5) %ia% rcs(lsize,3)
                                + rcs(bsize,5) %ia% rcs(dways,3)
                                + rcs(bsize,5) %ia% rcs(iways,3)
                             ## second-order: cache effects
                                + rcs(dsize,4) %ia% rcs(dways,3)
                                + rcs(lsize,3) %ia% rcs(dsize,4)
                                + rcs(isize,4) %ia% rcs(iways,3)
                                + rcs(lsize,3) %ia% rcs(isize,4)));
```

The association analysis provided insight into significant parameters. Splines can give these parameters, such as block sizes, line sizes, and cache ways, greater flexibility by using a greater number of pieces when fitting the piecewise polynomial. Restricted cubic spline transformations are denoted by `rcs(param,knots)` and interactions are denoted by `%ia%`. Of particular note are interactions between the block size and the significant cache parameters: line size, as well as data and instruction cache associativity.

4.3 Model Assessment

We assess performance and power models for absolute accuracy, relative accuracy, and their ability to capture non-monotonicity in the response topol-

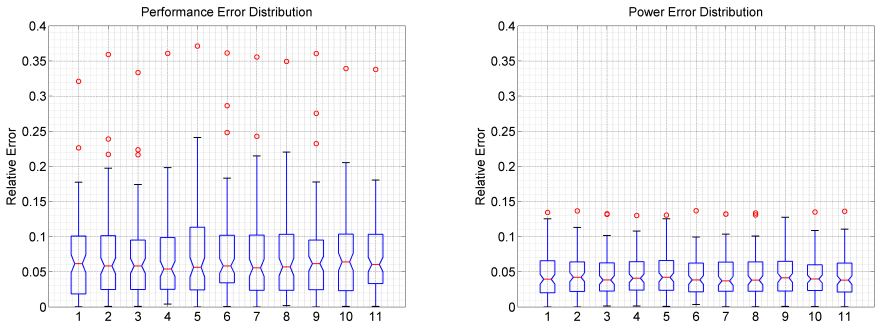

Figure 6: Absolute accuracy for performance (L) and power (R) models. Boxplots characterize the median error (red line), 1st and 3rd quartile (bottom and top of the box), and outliers (red circles). Matrix numbers refer to Table 1.

ogy. Absolute accuracy quantifies how closely models predict true metric values. Relative accuracy correlates predicted and true values to ensure larger values are predicted larger and smaller values are predicted smaller, which is important for optimization heuristics. Lastly, we compare predicted and true performance topologies, showing the models capture non-monotonicity and irregularity.

Figure 6 illustrates the error distribution for 100 validation samples collected separately from the 400 training samples. Performance (Mflop/s) is accurately predicted with median errors between 5 and 6 percent across 11 matrices. For most matrices, 75 percent of predictions have error rates less than 10 percent. Outliers rarely exceed 20 percent. Power (mW) is also accurately predicted with median errors between 4 and 5 percent. 75 percent of predictions have error rates less than 7 percent and outliers never exceed 15 percent. Such accuracy is sufficient for early-stage design optimization and these models could be supplemented with additional simulation if greater accuracy is desired.

Figure 7 illustrate the correlation between predicted and true values, plotting predicted values against observed true values for a representative matrix, 3dtube. Other matrices exhibit similar trends. Strong correlations manifest as strong linear trends, giving confidence that models will accurately predict the relative performance of designs throughout the space. Both performance and power models exhibit strong correlations. Performance and power predictions track their true values with correlation of

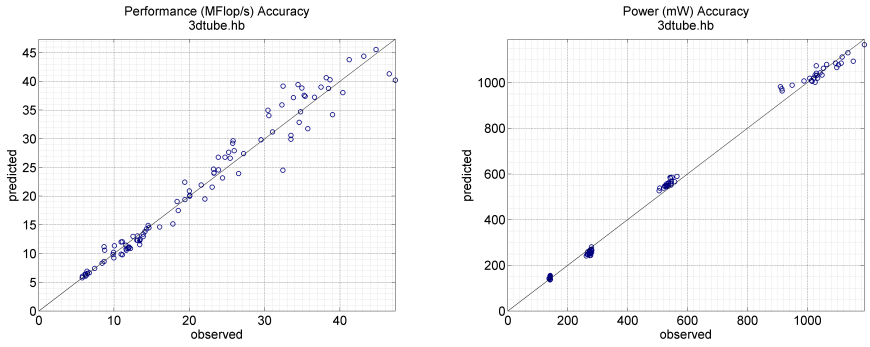

Figure 7: Relative accuracy for performance (L) and power (R) models: A linear trend indicates a strong correlation and high relative accuracy.

coefficients greater than 0.98.

Figure 8 illustrates the ability to infer non-monotonic response topologies. Figure 8L uses simulated data to illustrate the classic example of sparsity and irregularity translating into performance non-monotonicity. The grid's colormap illustrates the performance (Mflop/s) of various block configurations for a baseline cache architecture (Table 2). Each grid cell indicates the speedup relative to a non-blocked (1×1) implementation.

Figure 8R illustrates the model predicted performance topology. We observe significant similarity, with the predicted topology exhibiting high-performance at the same block sizes: 3×3, 3×6, 6×3, 6×6. Each of these blocks are predicted to exhibit similar performance and more detailed simulation might be used to choose a final block size. The models also capture performance discontinuities around optimal block sizes. For example, in both figures, 6×6 is optimal but most adjacent block sizes are worse than a non-blocked 1×1 implementation.

5 SpMV and Integrated Efficiency

Using models constructed by statistical inference across SpMV and architecture parameters, we demonstrate energy efficient performance maximization across the hardware/software interface. While simultaneous optimization of hardware/software parameters is beneficial and tractable, serial optimization incurs only modest penalties.

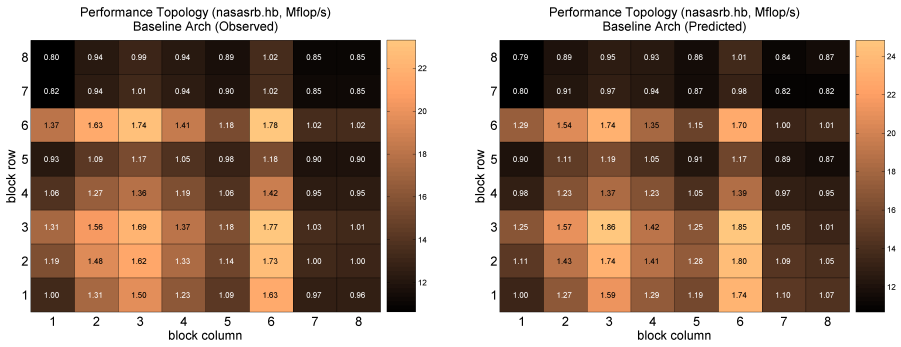

Figure 8: Measured (L) and predicted (R) performance topologies. Colormap illustrates Mflop/s and numbers within each cell indicate speedup over non-blocked 1×1 code. Data shown for a representative matrix nasasrb.

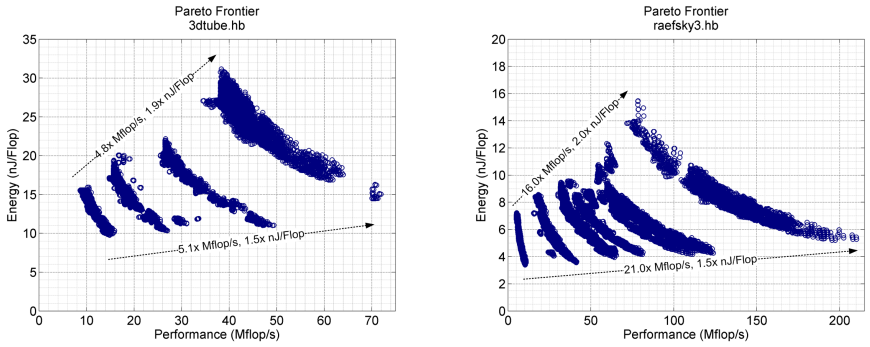

Figure 9: Energy efficiency for representative matrices 3dtube (L) and raefsky3 (R). Designs from the joint application and architecture space in performance and energy coordinates. Efficient frontier is lower right quadrant.

5.1 Energy Efficiency

Figure 9 illustrates the trade-offs between performance and energy for various points throughout the joint SpMV and architecture space. Figure 9L is representative of SpMV for matrices where the optimal block size is small (e.g., matrix 1, 3dtube, 3×3 blocks) while Figure 9R is representative of those where the optimal block size is bigger (e.g., matrix 10, raefsky3, 8×8). The effectiveness of SpMV application optimizations impacts the energy efficiency of performance gains.

In particular, Figure 9L illustrates performance gains for 3dtube where these gains come from the architecture space, primarily in the form of larger cache lines. Thus, we see four distinct strata of designs, one for each cache line size. Once a cache line size is chosen, however, the other parameters interact to improve performance and reduce energy. The degree to which these parameters interact determine the difference between the least efficient frontier and the most efficient Pareto frontier along the bottom of the strata. Simply increasing line size to 128B is insufficient, delivering 4.8x in performance at the cost of 1.9x in energy per operation. This strategy improves performance at a rate 2.5x faster than the rate of energy increases. In contrast, increasing line size and optimizing other cache parameters with optimal matrix block sizes is more efficient, delivering 5.1x in performance at a cost of 1.5x in energy per operation. Thus, coordination improves performance at a rate 3.1x faster than the rate of energy increases.

To illustrate the sensitivity of these trends to the effectiveness of application tuning, Figure 9R considers matrix 10, raefsky3, where blocking has a much larger impact. Raefsky3 is a sparse matrix with a dense sub-structure that enables 8×8 blocking with a fill ratio of 1.0x (i.e., no explicitly filled zeros required). The effectiveness of large matrix blocks further improves efficiency gains through coordination. We also observe greater stratification of the design space where data cache size and associativity create their own strata. Considering the first three strata from low to high performance, initially line size falls (128B to 16B), data cache size increases (16KB to 256KB), and data cache associativity falls (8 to 1). The subsequent four strata increase the cache line size up to 128B again. Overall, the trend along the least efficient frontier produces a 16.0x performance gain for 2.0x energy cost. This strategy improves performance at a rate 8.0x faster than the rate of energy increases. The most efficient frontier, which optimizes both matrix block size and cache structure, delivers 21.0x in performance for a 1.5x increase in energy cost. Performance increases 14.0x faster than energy.

	Matrix	r	c	Fill Ratio
1	3dtube	3	3	1.02
2	bayer02	1	1	1.00
3	bcsstk35	3	6	1.12
4	bmw7st	3	2	1.32
5	crystk02	3	3	1.00
6	memplus	1	2	1.35
7	nasasrb	3	6	1.13
8	olafu	3	6	1.12
9	pwtk	3	3	1.22
10	raefsky3	8	8	1.00
11	venkat01	4	4	1.00

Table 4: Performance maximizing matrix block sizes.

These results suggest applications with more effective parameterization can translate energy costs into performance more efficiently. A larger cache line size alone is less efficient if the application is not structured to exploit the extra spatial locality. In our representative examples, raefsky3 and its 8×8 blocks achieved a performance to energy cost ratio 4.5x greater than that of 3dtube and its 3×3 blocks.

5.2 Simultaneous Optimization

The Pareto analysis indicates the need to optimize both application and architecture. While they show efficient performance-energy trade-offs by moving from one efficient point to another on the frontier, this section considers the optimization of a baseline architecture with an unblocked matrix. Thus, we can identify the relative application and architecture contributions to efficiency. We first consider an ideal scenario where all parameters are optimized jointly by exhaustively evaluating inferential models for every point in the space to identify the performance maximizing design. We then consider the energy costs of performance maximization.

Figure 10L illustrates the performance (Mflop/s) for a range of optimization scenarios. Note again, that we optimize true Flops and do not count explicitly filled zeros that might arise from matrix blocking. The baseline considers the base cache architecture with an unblocked matrix, which achieves an average of 18 Mflop/s. Application optimization identifies the best block size for each matrix (Table 4). Blocking achieves 29 Mflop/s, on average, a 1.6x gain. Matrix 10 (raefsky3) and 11 (venkat01) benefit more from blocking as dense sub-structures allow for larger block

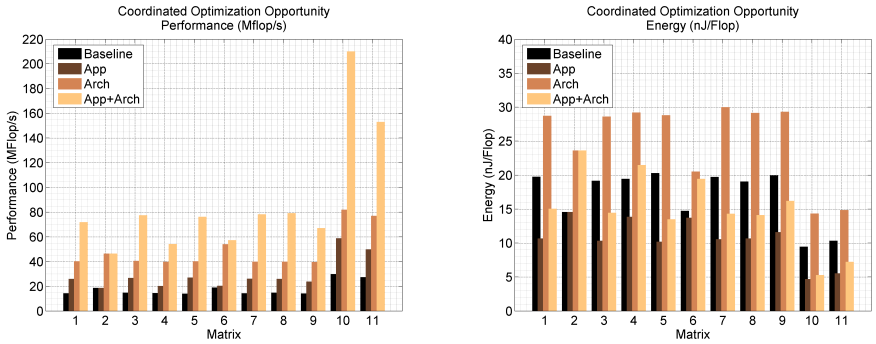

Figure 10: Performance (L) and efficiency (R) optimization. Matrix numbers refer to Table 1.

sizes with modest fill. These matrices can implement large 8×8 and 4×4 blocks requiring no explicitly filled zeros and producing speedups of 2.0x and 1.8x, respectively.

Architecture optimization relies on larger line sizes to amortize off-chip memory latencies over more data and lower cache associativities to reduce the amount of cached streaming data. Architecture alone achieves up 49 Mflop/s, on average, a 2.7x performance gain. However, application and architecture together achieve 88 Mflop/s, a 5.0x gain that is greater than the 4.3x expected from independent effects (i.e., 1.6x from application and 2.7x from architecture).

Figure 10R illustrates the energy costs of these performance maximizing designs. Energy is measured in nJ/Flop and computed by dividing power and performance predictions. The baseline design consumes an average of 17.0 nJ/Flop. Matrix blocking reduces energy costs to 10.6 nJ/Flop by improving locality and reducing the number of expensive, off-chip memory accesses for the source vector. Thus, in an application-only strategy, greater performance and lower energy costs are correlated. Better locality improves performance by 1.6x through fewer memory stalls and reduces energy by 0.6x through fewer memory accesses, which dominate total energy.

An architecture only strategy is less efficient, increasing energy costs to 25.2 nJ/Flop. Such a strategy improves performance by increasing cache line size to amortize memory access latencies across more bytes, but also increases the energy cost of filling a cache line, which increases in proportion to the line size. Secondary effects, such as larger data caches and less associativity, have a smaller effect on both performance and energy. Instruction and data cache structure comprise less than 3 percent of total energy. Thus,

architecture alone improves performance by 2.7x at a cost of 1.5x in energy per operation.

Through coordinated optimization, however, one can enhance performance while controlling energy costs per operation. Figure 10 indicates energy per Flop decreases by 10 percent (0.9x) even as the Flop rate increases by 5.0x. While larger cache line sizes increase energy per access, they require roughly constant energy per byte since memory energy is dominated by data transmission over the bus. At the same time, matrix blocking increases, on average, the number of Flops performed per byte transferred from memory. The net effect is performance (Mflop/s) increasing at a rate faster than energy per operation (nJ/Flop).

5.3 Serial Optimization

Simultaneous optimization achieves synergies across the hardware/software interface and serialized optimization might leave performance and energy efficiency unexploited. Figure 11 quantifies these effects, illustrating the penalties from a serial approach.

- **Arch>App:** For each matrix, optimize architecture then application. First, identify architecture $A_{1\times1}$ that maximizes performance for baseline matrix block size 1×1. Then, identify block size $r\times c$ that maximizes performance given architecture $A_{1\times1}$.
- **App>Arch:** For each matrix, optimize application then architecture. First identify matrix block size $r\times c$ that maximizes performance for baseline architecture A_0. Then, identify architecture $A_{r\times c}$ that performance given architecture $r\times c$.

Arch>App describes current hardware design practices in which architects design an processor to improve average performance over static implementations for hundreds of applications. Application designers then take their application and tune for that static hardware implementation. App>Arch is a more application-centric strategy that might lead to significant hardware design heterogeneity [16].

Figure 11L suggests performances penalties are modest with Arch>App and App>Arch incurring an average penalty of 5.7 and 4.2 percent, respectively, for matrices that implement matrix blocking. Note 1×1 is the optimal block size for matrices 2 and 6, which leads to zero penalties. Figure 11R indicates larger energy inefficiencies of 12.3 and 4.4 percent, respectively. In

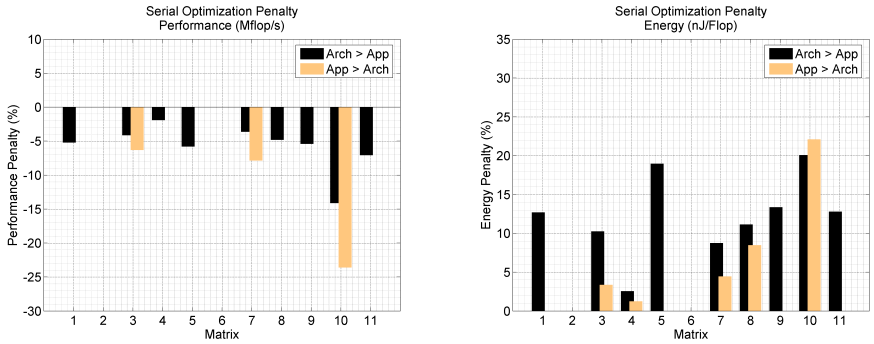

Figure 11: Serial optimization and its implications for performance (L) and efficiency (R). Arch-App first optimizes architecture for an unblocked matrix and then optimizes application for that architecture. App-Arch first optimizes the matrix block size and then optimizes the architecture for that blocked matrix.

both cases, penalties are more likely if the architecture is specified before an application is tuned.

In Arch>App, the architecture over-provisions resources to compensate for poor locality in an unblocked matrix. When the matrix is finally blocked, the energy costs of an over-provisioned architecture remain. In App>Arch, optimizing the application first maximizes re-use and locality. Subsequent architectural optimizations, such as wider cache lines, increase performance by amortizing off-chip latency over more bytes, but also increase power as more bytes are transferred and more energy is consumed for every cache miss. The net effect on energy per Flop is negligible.

Matrix 10, raefsky3, experiences the greatest serialization penalties for both performance and energy, as well as both Arch>App and App>Arch. This matrix also has the largest block size (8×8=64). Performance and energy penalties range from 14 to 24 percent. Matrices 3, 7, and 8 also have larger block sizes (3×6=18) and experience both Arch>App and App>Arch penalties. These results might suggest a relationship between serialization penalties and the extent to which matrix blocking is leveraged to deliver performance and energy efficiency.

6 Related Work

This work is very similar in spirit to prior work in coordinated design for VLIW architectures, where compiler code transformations interact with architecture design. The work by Fisher et al. on custom-fit processors is representative of this era of VLIW compiler and architecture research [7]. In such spaces, back-end compiler optimizations play a critical role in extracting and scheduling parallelism. In more recent, separate efforts, Cooper et al. and Dubach et al., explore back-end compiler optimizations and phases using search heuristics [4, 6].

In contrast, our approach to software tuning focuses on source-to-source transformations, which explore parameterizations of data structures and algorithms. Compared to back-end compiler optimizations, source-level transformations, when applicable, tend to have a larger effects on performance and efficiency. The source code generators used in this work (OSKI) employ compiler-sympathetic code constructs, such as unrolling matrix block loops. The SPIRAL project implements both software and hardware generators for digital signal processing algorithms [20]. Our approach to statistical regression modeling might also be applied to combining these generators.

Past tuning for sparse linear algebra relies on a combination of models and empirical measurements. These empirical measurements are collected from code generators, beginning with PHiPAC for dense matrix multiply [3], ATLAS for dense linear algebra in BLAS [29], and OSKI for sparse linear algebra [26]. The optimal library implementation is identified through profiling and heuristic search on the target platform [24]. Vuduc et al. construct statistical classifiers for dense matrix multiply [25]. In contrast, we construct predictive models for the more irregular, sparse matrix multiply. Moreover, prior work optimizes kernels for a fixed hardware platform whereas we optimize kernels and hardware simultaneously.

Mohiyuddin et al. propose a hardware-software co-tuning methodology that expands conventional software-only tuning for dense matrix-matrix multiply, sparse matrix-vector multiply, and stencil computation [18]. This prior work demonstrates the advantages of co-tuning over a hardware-only or software-only approach. Mohiyuddin et al. observe the challenges of simulation hundreds of kernel implementations on tens of hardware configurations. Our effort addresses these co-tuning costs with statistical inference and predictive modeling.

Lee and Brooks construct statistical regression models to predict performance and power as a function of architectural parameters [15]. Ipek et al. construct neural nets for the same purpose [13]. Separate models are

constructed for each application. Dubach et al. construct similar models for a set of applications and infer the performance of previously unseen applications as a linear combination of the original set [6]. All of these prior efforts model and optimize the architecture for a fixed software implementation whereas we consider software and architecture simultaneously.

7 Conclusion

Process technology and multi-core parallelism are no longer providing the energy efficiencies required by modern computing. Specialization delivers efficiency, but incurs significant engineering costs. We show the potential to mitigate some of these costs by creating generator frameworks that can optimize across the hardware/software interface. In our prototype system, we leverage prior work on domain-specific application tuners and flexible architecture generators. By connecting the OSKI code generator with the Tensilica/Smart Memories hardware through an integrated inference model, we improve the average performance of our sparse linear algebra kernel by 5.0x while decreasing the energy by 10 percent (0.9x).

These efficiencies arise from application tuning, which simultaneously improves performance and lowers energy, and architecture design, which trades-off performance and energy. These different trade-off dynamics across abstraction layers motivate a more comprehensive view of energy efficiency and computing that spans the hardware/software interface. Although we span abstraction layers, we do not seek to break them. Instead, we favor clean and minimal interfaces across abstraction layers to enable coordinated design.

In future, we will examine generalizations of our framework to a broader range of applications and architectures. These efforts will require continued research in parameterized code generation, parameterized hardware design, and optimization strategies. Moreover, we must better understand the possibilities of integrating disparate generators to construct a full system. As application and architecture designers build a variety of generators, we need frameworks that maintain the integrity of abstraction layers while building new bridges across them.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant #0937060 to the Computing Research Association for the CIFellows Project. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the view of the National Science Foundation or the Computing Research Association.

References

[1] J. Ansel, C. Chen, Y. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: a language for compiler and algorithmic choice. In *PLDI: Conference on Programming Language Design and Implementation*, June 2009.

[2] Arvind, R. Nikhil, D. Rosenband, and N. Dave. High-level synthesis: An essential ingredient for designing compex ASICs. In *ICCAD: International Conference on Computer Aided Design*, November 2004.

[3] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A Portable High-Performance, ANSI C coding methodology. In *ICS: International Conference on Supercomputing*, 1997.

[4] K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1), 2002.

[5] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, October 1974.

[6] C. Dubach, T. Jones, and M. O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *MICRO: International Symposium on Microarchitecture*, December 2007.

[7] J. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: Letting applications define architectures. In *MICRO: International Symposium on Microarchitecture*, December 1996.

[8] R. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, Mar/Apr 2000.

[9] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer, 2002.

[10] F. Harrell. *Regression Modelin Strategies*. Springer, 2001.

[11] M. Hill and M. Marty. Amdahl's law in the multicore era. *IEEE Computer*, July 2008.

[12] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein. Scaling, power, and the future of cmos. In *IEDM: International Electron Devices Meeting*, December 2005.

[13] E. Ipek, S. McKee, B. de Supinski, M. Schulz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS: International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[14] D. Jani, G. Ezer, and J. Kim. Long words and wide ports: Reinventing the configurable processor. In *Hot Chips*, August 2004.

[15] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS: International Conference on Architectural Support for Pogramming Languages and Operating Systems*, October 2006.

[16] B. Lee and D. Brooks. Illustrative design space studies with microarchitectural regression models. In *HPCA: International Symposium on High-Performance Computer Architecture*, February 2007.

[17] Micron. Technical note TN-47-04: Calculating memory system power for DDR2. In *www.micron.com*, June 2006.

[18] M. Mohiyuddin, M. Murphy, L. Oliker, J. Shalf, J. Wawrzynek, and S. Williams. A design methodology for domain-optimized power-efficient supercomputing. In *SC: Supercomputing*, November 2009.

[19] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO: International Symposium on Microarchitecture*, December 2006.

[20] M. Pueschel et al. SPIRAL: Code generation for DSP transforms. *IEEE Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[21] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu. Programing by sketching for bitstreaming programs. In *PLDI: Conference on Programming Language Design and Implementation*, June 2005.

[22] A. Solomatnikov, A. Firoozshahian, W. Qadeer, O. Shacham, K. Kelley, Z. Asgar, M. Wachs, R. Hameed, and M. Horowitz. Chip multi-processor generator. In *DAC: Design Automation Conference*, June 2009.

[23] A. Solomatnikov, A. Firoozshahian, O. Shacham, Z. Asgar, M. Wachs, W. Qadeer, S. Richardson, and M. Horowitz. Using a configurable processor generator for computer architecture prototyping. In *MICRO: International Symposium on Microarchitecture*, December 2009.

[24] R. Vuduc. Automatic performance tuning of sparse matrix kernels. *Ph.D. Thesis, University of California at Berkeley*, 2004.

[25] R. Vuduc, J. Demmel, and J. Bilmes. Statistical models for empirical search-based performance tuning. *Int'l Journal of High Performance Computing Applications*, 2004.

[26] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *SciDAC, Journal of Physics: Conference Series*, June 2005.

[27] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *DAC: Conference on Design Automation*, 2001.

[28] M. Wehner, L. Oliker, and J. Shalf. Towards ultra-high resolution models of climate and weather. *International Journal of High Performance Computing Applications*, 2008.

[29] R. Whaley, A.Petitet, and J. Dongarra. Automated empirical optimization and the ATLAS project. *Parallel Computing*, 2001.