Original Article

International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

ExaSAT: An exascale co-design tool for performance modeling

The International Journal of High Performance Computing Applications
2015, Vol. 29(2) 209–232
© The Author(s) 2015
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342014568690
hpc.sagepub.com
SAGE

Didem Unat, Cy Chan, Weiqun Zhang, Samuel Williams, John Bachan, John Bell and John Shalf

Abstract

One of the emerging challenges to designing HPC systems is understanding and projecting the requirements of exascale applications. In order to determine the performance consequences of different hardware designs, analytic models are essential because they can provide fast feedback to the co-design centers and chip designers without costly simulations. However, current attempts to analytically model program performance typically rely on the user manually specifying a performance model. We introduce the ExaSAT framework that automates the extraction of parameterized performance models directly from source code using compiler analysis. The parameterized analytic model enables quantitative evaluation of a broad range of hardware design trade-offs and software optimizations on a variety of different performance metrics, with a primary focus on data movement as a metric. We demonstrate the ExaSAT framework's ability to perform deep code analysis of a proxy application from the Department of Energy Combustion Co-design Center to illustrate its value to the exascale co-design process. ExaSAT analysis provides insights into the hardware and software trade-offs and lays the groundwork for exploring a more targeted set of design points using cycle-accurate architectural simulators.

Keywords

Performance modeling, exascale co-design, exascale systems, performance analysis, combustion codes, abstract machine model, compiler analysis, cache modeling, stencil applications, design trade-offs

1 Introduction

The designers of exascale systems are faced with challenges introduced by system cost and power consumption (Shalf et al., 2010). In order to improve delivered performance for large-scale applications within practical cost and power budgets, it is essential to move towards a hardware–software *co-design* process where the hardware design space is explored in tandem with software optimizations. The US Department of Energy co-design centers Cesar (`http://cesar.mcs.anl.gov/`), Exact (`http://exactcodesign.org`) and ExMatEx (`http://exmatex.lanl.gov/`) are performing multi-disciplinary research to iteratively design various aspects of applications including core algorithms, programming models, compilers, and runtimes to ensure that they will meet the requirements of future scientific simulations. Effective co-design requires a performance framework to *rapidly* evaluate the proposed hardware by vendors and software changes and provide end-to-end analysis of an application.

In order to evaluate hardware–software design trade-offs, we introduce a compiler-based performance modeling framework, ExaSAT (**Exa**scale **S**tatic **A**nalysis **T**ool), that enables rapid exploration of hardware design space and helps bridge the communication gap between the application developers and hardware designers. Because many exascale architectural specifications are currently undefined, our performance model is parameterized to help explore different design choices. Additionally, our framework explores a parameterized software optimization space (e.g. cache blocking, fusion, etc.) together with the hardware design space so that we do not base conclusions about hardware requirements on unoptimized codes.

The initial design of the ExaSAT framework focused on combustion codes that use algorithms on structured grids. Combustion currently provides 85% of the

Computational Research Division, Lawrence Berkeley National Laboratory, USA

Corresponding author:
Didem Unat, Computational Research Division, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720, USA.
Email: dunat@lbl.gov

nation's energy needs and is a key driver for exascale computing (https://flowcharts.llnl.gov/archive.html#energy_archive). Economic and environmental concerns are driving the development of new combustion systems targeting toward clean and efficient use of alternative fuels. Developing these systems requires simulations with sufficient chemical fidelity to differentiate the behavior of candidate fuels in realistic engine conditions. Exascale computing offers the promise of enabling the underlying science to design fuel-efficient, clean-burning vehicles, planes, and power plants for electricity generation (http://exactcodesign.org/main/wp-content/uploads/ExaCT-Deep-Dive-Intro.pdf). For example, exascale computing will enable the development of new homogenous charge compression ignition (HCCI) engine designs that lead to lower emissions, cleaner combustion and a 25–50% increase in efficiency. It is predicted that HCCI will require 20 days runtime at billion way concurrency, a 3 PB memory to hold the simulation state, and will generate 1.0 EB of data for analysis. Thus, studying the performance requirements of combustion applications on potential exascale designs is extremely valuable.

Our framework provides a *missing capability* in the co-design toolset where fast evaluation is needed at the expense of accuracy. Simulations are slow, leading to very narrow, yet highly detailed analysis of a small kernel or a subcomponent of a system. For example, it would take a hardware simulator, such as RAMP (Krasnov et al., 2007; Wawrzynek et al., 2007), a few hours to generate a single configuration of a multi-core processor, though the application on the configured architecture would then run in real-time. It is easier to configure a cycle-accurate software simulator, such as GEM5 (Binkert et al., 2011), but it would take several hours to run an application to get meaningful results. In comparison, ExaSAT can evaluate hundreds of hardware–software configurations per minute on a desktop machine. Thus, ExaSAT complements hardware and software simulators in the co-design process by serving as a design space-pruning tool. In addition, by restricting the framework to structured grid problems, we improve the accuracy of the performance model. For example, while the compiler front-end gathers the read–write properties of streaming arrays, the cache model takes into account the reuse in stencil arrays when estimating the memory traffic. Since the access pattern of such applications operating on dense arrays can be statically inferred, we can quickly derive fast analytic performance models. Our approach does not support analysis for irregular or graph-based codes where the access pattern is only available at runtime. On the other hand, understanding the performance of structured grid problems provides insights into the requirements of an important class of applications using stencil-based partial differential equation (PDE) solvers.

This paper makes the following contributions:

- We develop the ExaSAT framework to statically analyze an application and automatically gather key characteristics about the computation, communication, data access patterns and data locality that are important in characterizing the performance of combustion codes.
- We design both an XML representation of the application workload characteristics and an XML representation of the exascale machine configurations. The XML serves as a medium and an interface for our framework to work with other tools, such as Pin tools or architectural simulators.
- We implement a performance model that can combine both hardware and software parameters to bound performance, rapidly explore design trade-offs, and extrapolate these requirements to potential hardware realizations in the exascale timeframe (2020).
- We perform deep code analysis of SMC, a proxy implementation of a production combustion code, and use our results to address key co-design questions acquired from our industry partners. Finally, we quantify the SMC performance on exascale proxy architectures using ExaSAT.

The rest of the paper is organized as follows. Section 2 provides background on related work and explains how ExaSAT differs from existing performance modeling tools. Section 3 introduces the ExaSAT framework including the compiler-based front-end, XML specification for the code description and abstract machine model, and performance modeling component. Validation of the framework is provided at the end of the section. Section 4 provides an overview on the characteristics of combustion applications and gives details about a proxy application used to conduct performance analysis for this paper. We present performance analysis and results in Section 5. Section 6 makes projections on an exascale machine, evaluates the implications of our findings, and provides feedback to hardware and software designers for exascale systems. The section includes discussion on limitations and future work. We conclude the paper in Section 7.

2 Related work

The overarching goal of the co-design centers is to understand the interplay between hardware and software design trade-offs. Given the uncertainty in exascale architecture, co-design centers need an application characterization tool to iteratively perform the hardware–software optimization processes envisioned for the co-design of HPC systems. GEM5 (Binkert et al., 2011), CACTI (Thoziyoor et al., 2008) and SST (Rodrigues et al., 2011) are software simulators that parameterize machine specifications but they are slow, leading to narrow analysis of small kernels or isolated components of the system such as the interconnect. FPGA-based cycle-accurate, circuit-level emulators such as RAMP (Krasnov et al., 2007; Wawrzynek et al., 2007) and the CoDEx emulator (Shalf et al., 2011) can capture very detailed behavior of the architecture, but are not as easily configurable as software simulators. For example, if the number of cores in the emulator is changed from 64 to 128, every single module will need to be manually adjusted for the new cache sizes, address spaces, and network sizes. Furthermore, very fine-grained circuit-level design introduces the danger of missing general performance trends because of the large amount of extraneous data generated. Benchmarking provides an immediate response, but limits the analysis to current hardware architectures and the results can be biased towards the particular implementation or compiler options used because we cannot separate implementation-specific results from performance opportunities.

Given the cost of setting up both simulations and emulations, analytic models play a complementary role in design space exploration to identify the subset that is of interest for further study with simulation and emulation. Higher level analytic models such as the Roofline model (Williams et al., 2009) provide speed-of-light (cannot-exceed) performance expectations, but offer a very coarse-grained description of performance in terms of flop rates and DRAM bandwidth. Convolution-based approaches such as PMAC tools (Snavely et al., 2002; Carrington et al., 2003) provide coarse-grained performance analysis through correlation, and generate models by convolving application characteristics (the *signature*) through instrumentation with a vector describing the target machine attributes. Similar to ExaSAT, Pbound (Narayanan et al., 2010) mixes static and runtime data to estimate upper performance bounds. However, by focusing on the structured grid applications ExaSAT can better characterize data movement requirements and thus provide tighter performance bounds. In addition, ExaSAT combines software optimizations with the performance model. For more rapid construction of analytic models, pseudo-languages have been proposed. For example Aspen (Spafford and Vetter, 2012) is a domain-specific language that enables a user to describe the parallelism, arithmetic operation counts, and data movement to build a model. Specifying the model in Aspen still requires a lot of work, and the quality of the model depends on the ability of the user to accurately capture the application signature.

Our ExaSAT framework has adopted a compiler-based approach to automate the process of generating the performance model. Compiler-based approaches have the dual advantages of being less labor-intensive (thus more easily applied to large codes) and providing a more accurate description of codes to the analytic model. Static analysis cannot capture the dynamic behavior of the application; however, metrics gathered by dynamic traces or binary instrumentation are very sensitive to compiler flags and machine configuration, which can obscure conclusions during the analysis. Moreover, existing machines do not reflect the characteristics of exascale machines and early prototypes of exascale hardware are not available for evaluation. Instead, in ExaSAT we parameterize the hardware configurations to support the static compiler analysis and increase the flexibility of the framework to support exascale machine models. The model is not completely agnostic of inputs either. Rather it is parameterized by a number of runtime parameters such as problem size. These parameters are extracted from the input deck for the various applications so that the model sees the resultant performance impact.

Another aspect that differentiates our approach from others is that our framework uses data movement metrics to quantify performance. Most existing performance analysis and instrumentation focus on flop counts, cache hit rates, and other processor-based metrics. We focus on data movement as a key metric because it has become one of the most challenging hardware constraints for the design of future systems. Since flops have become cheaper, the energy of data movement dominates the energy cost (Shalf et al., 2010; Unat et al., 2014). Thus, our analysis of both on-node and off-node data movement not only provides valuable feedback to hardware designers, but also to exascale programming model, compiler, and runtime designers.

Finally, in addition to a parameterized machine model, our modeling approach includes a parameterized model for software optimizations. Previous work (Mohiyuddin et al., 2009; Chan et al., 2013) showed that estimating hardware requirements on unoptimized software led to incorrect conclusions. Similarly, tuning software without taking into account hardware choices did not result in an optimal solution. These findings motivated us to incorporate a parameterized set of software optimizations into our framework. Our approach holds a substantial advantage over studies that measure code bandwidth and flop utilization without considering software transformations (Balaprakash et al., 2013). As more detail emerges on hardware design proposals, the upper bounds provided by the analytic models produced by ExaSAT should be examined together with the lower bounds supplied by binary instrumentation on current machines to provide a complete picture of theoretical vs. achievable performance.

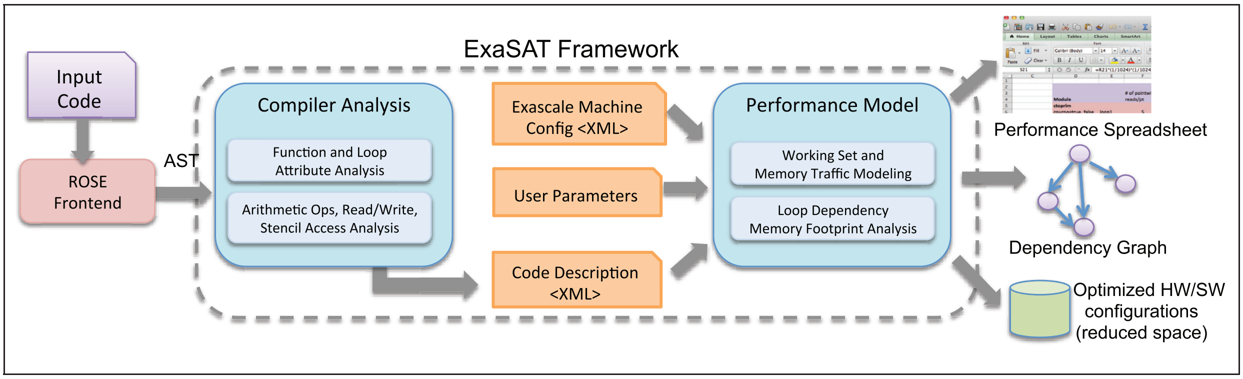
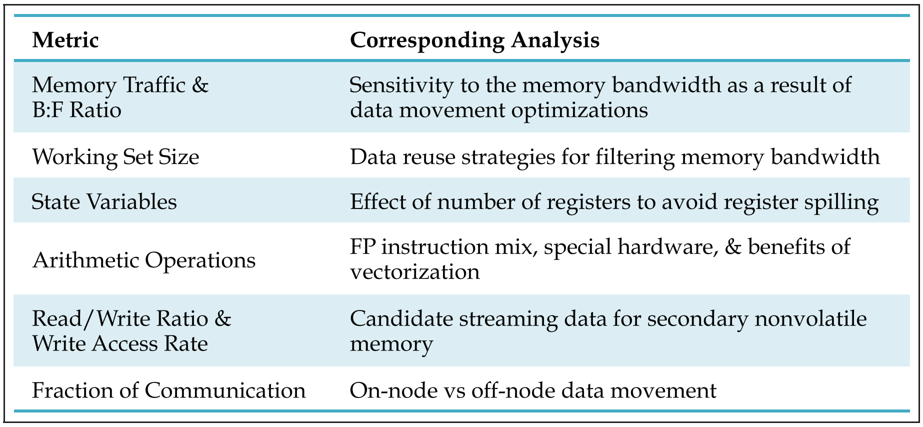

ExaSAT Framework
Input Code
ROSE Frontend
AST
Compiler Analysis
Function and Loop Attribute Analysis
Arithmetic Ops, Read/Write, Stencil Access Analysis
Exascale Machine Config <XML>
User Parameters
Code Description <XML>
Performance Model
Working Set and Memory Traffic Modeling
Loop Dependency Memory Footprint Analysis
Performance Spreadsheet
Dependency Graph
Optimized HW/SW configurations (reduced space)

Figure 1. Workflow in the ExaSAT framework.



Metric	Corresponding Analysis
Memory Traffic & B:F Ratio	Sensitivity to the memory bandwidth as a result of data movement optimizations
Working Set Size	Data reuse strategies for filtering memory bandwidth
State Variables	Effect of number of registers to avoid register spilling
Arithmetic Operations	FP instruction mix, special hardware, & benefits of vectorization
Read/Write Ratio & Write Access Rate	Candidate streaming data for secondary nonvolatile memory
Fraction of Communication	On-node vs off-node data movement

Figure 2. Subset of performance metrics captured.

3 The ExaSAT framework

As illustrated in Figure 1, the ExaSAT framework is composed of two main components – the front-end compiler analysis and back-end performance model. The front-end component collects procedural and loop-level information to create a profile of the code, which is stored in an XML file. The XML code description is then fed into the back-end analysis, which produces dependency graphs, generates performance models, and produces statistical summaries of the code's characteristics. The performance model is parameterized with (1) machine specifications such as cache size, (2) user parameters such as problem size, and (3) software optimizations such as loop fusion and blocking. The optimized hardware configurations provide the reduced design space for the architecture simulators and the optimized software configurations provide feedback to application developers and programming system tools.

As a result of interactions with application experts and industry partners participating in the Department of Energy Fast Forward program (`https://asc.llnl.gov/fastforward/`), we assembled a set of performance metrics that reflect the characteristics of the exascale applications and made these metrics the center of ExaSAT. Figure 2 shows the list of metrics that we used for evaluating the various hardware components and software optimizations. The first metric that quantifies the benefits of data movement optimizations is the byte-to-flop ratio (B:F), which expresses the balance between the application's required flops and memory traffic. We use this ratio as an indicator of energy and performance improvement for optimization strategies. Since the degree of reuse enabled by the on-chip memory configuration can significantly impact memory traffic, we also measure working set sizes.

Another metric related to data movement concerns state variables and registers. We analyze state variables to estimate the impact of the architectural register count on the number of spills, which cause additional loads and stores and pipeline bubbles. Although the majority of our analysis focuses on data movement, ExaSAT provides estimates of arithmetic costs as well. We analyze the instruction mix, the use of expensive transcendental functions such as exponentials, and the impact of vectorization.

ExaSAT enables us to investigate alternative technologies such as non-volatile memory (NVRAM) and integrated network controllers (NIC). NVRAM is considered to be a cost-effective alternative that can serve as a high-capacity, secondary memory (Lee et al., 2009), however the writes to NVRAM are costly both in terms of dynamic energy consumption and performance. In order to assess what data to put into NVRAM, we use the community standard, read/write ratios (Li et al., 2012), and a new metric known as the *write access rate*, which is the fraction of write references to a particular variable. Lastly, we use the fraction of communication time to assess the impact of off-node communication on application performance. This metric helps us evaluate whether there is a strong justification for custom NICs, which integrate the network controller on chip to increase injection bandwidth. Next, we explain how we extract these metrics with compiler analysis and how we employ them in the performance model.

3.1 Compiler analysis

The compiler analysis for ExaSAT was built on top of the ROSE compiler framework (Quinlan et al., 2002), which is an open-source compiler infrastructure developed at Lawrence Livermore National Laboratory. ROSE parses the C, C++ , and Fortran source to convert it into an abstract syntax tree (AST) that we can manipulate and analyze. Our compiler analysis currently accepts Fortran inputs; however, it can be extended to support C/C++ inputs.

3.1.1 Procedure and loop attributes. The analysis of the AST begins by querying the procedure definitions (subroutines and functions) in a module. For each procedure, we collect a list of variable symbols used in the procedure body and classify them into two categories: L: locally declared variable symbols and, U: variable symbols referenced within a procedure. The set difference ($U \setminus L$) of these two gives us the non-local variables, which can be global, defined in another module, or passed as an argument to the procedure. Fortran presents a special case because of its pass-by-reference semantics for subroutine arguments; procedure arguments that are declared with an *intent* type modifier are not technically local, and must be excluded from the L list.

After completing the live variable analysis[1] and locality analysis for the procedure, we collect detailed loop-level information. The loop analysis handles perfectly and imperfectly nested loops and is carried out inclusively on the entire loop body without excluding child loops.

When we generate the XML output, we make the loop-level information exclusive (not including attributes in the subloops). This is important to accurately estimate performance because both the arithmetic and memory operations are multiplied by the iteration space in the performance model.

For each loop, we gather loop attributes such as iteration bounds and strides, which are later used by the performance model to reason about the iteration space. Loop bounds typically depend on application parameters that are determined at runtime; therefore, we track symbolic rather than actual values and later perform symbolic replacement based on the user's parameters. Maintaining a symbolic representation of the iteration space also enables the performance model back-end to analyze the effect of software transformations such as loop blocking and fusion.

In order to estimate the total arithmetic workload, we count the floating point arithmetic operations (addition, subtraction, multiplication and division) in the loop body. In addition, we count math intrinsic functions (e.g. exp() or log()) because they can be significantly more costly to execute. The compiler analysis searches for function reference expressions in the loop body and uses a lookup table to identify such functions.

3.1.2 Data access analysis. Array access analysis is one of the crucial parts of the compiler analysis because the read/write properties of arrays are utilized by the performance model to compute on-chip data movement and memory footprint. ROSE provides an interface to get lists of the read references (R list) and write references (W list) for a given statement. This interface partially serves our needs by enabling us to classify variables as read-only, write-only or both. In scientific codes, some array dimensions may not represent spatial dimensions, but rather different physical properties or quantities such as density, temperature, pressure, or energy. We differentiate such dimensions by representing each array as an array–component pair. For example, the two references to the array Q in $Q(i,j,k,imx)$ and $Q(i,j,k,imy)$ refer to two different array–component pairs: (Q,imx) and (Q,imy). The location of such dimensions is tunable. However, we currently require all the arrays in the application to have the physical property represented in the same index location. The user can identify which indices represent spatial dimensions and which are non-spatial parameters. The differentiation of arrays at the component level is necessary because the reuse pattern, and thus the working set size, may be different for each component. We group references by array–component pairs and return separate lists for the read-only variables ($R \setminus W$), write-only variables ($W \setminus R$), and the arrays that are both read and written ($R \cap W$).

In order to model data reuse in the cache, we need more information with respect to the array access patterns. We support the read/write property analysis by examining all the references to an array–component pair in a basic block. The array references are broken into individual subscript expressions to extract their relative offsets to the loop indices. This helps us determine the distance between two references to the same array. Another important property is whether the first reference to an array is a load or a store. If the first reference is a load followed by a store, the load requires the data to be brought into cache from memory before it is written. On the other hand, if a load is preceded by a store, then the load may be carried out from the cache without incurring any additional memory traffic. Our tool conducts the first reference analysis within a loop to more accurately model cache reuse and support the analysis of advanced memory instructions such as non-temporal stores.

If the program expands into multiple files, we require the user to generate a separate XML per file. The XML code description contains the function call information such as module:function name and argument–parameter mapping. The performance model will take the XML of the file of interest with its dependent XML files as an input. When estimating the performance, if a function call is encountered, the compute cost of that function will be added to the compute time of the callee. Because of the nesting nature of function calls, this complicates the performance analysis particularly for the read/write properties of arrays. We are still investigating how to improve the analysis and currently conservatively assume that arrays are modified if we cannot automatically determine if the function has side-effects. We made an exception for the side-effect analysis of Fortran math intrinsics and assume the arguments for such functions are read-only. In addition to arrays, we conduct a similar analysis for the scalar variables referenced in each loop to help estimate register usage, though the read/write property analysis for scalar variables is much simpler.

3.2 XML Description

The ExaSAT compiler analysis outputs the results in an XML intermediate representation (XML-IR) to interface with the back-end performance modeling component of the framework. This enables the utilization of the performance model directly from the high-level XML-IR, bypassing the compiler analysis step. In this way, program variants or hypothetical code formulations can be evaluated without having to write the actual code. Similarly, the XML output of the compiler analysis can be fed to another tool such as an architecture simulator, bypassing the performance model step. We currently provide the XML description of the communication patterns in the codes to the SST simulator (Rodrigues et al., 2011) to simulate different interconnection topologies. The XML-IR element hierarchy is shown in Figure 3 and a more detailed design document can be found at http://crd.lbl.gov/projects/combustion-codesign-2/.

3.2.1 Machine configuration.

The machine configuration used as an input to the performance model can be specified in a separate XML. ExaSAT focuses on the aggregate performance of the computational throughput and memory bandwidth between the CPU and the DRAM. It considers the bandwidth filtering capability of last level cache, which is determined by the total amount of exclusive on-chip memory per thread or group of co-operating threads. An example machine configuration XML, shown in Listing 1, represents an exascale extrapolation of a many-core architecture (Ang et al., 2014). The example shows a 1000-core machine with 10 TF aggregate computational throughput, 1 TB/s aggregate memory bandwidth and 64 kB cache per core. The XML also allows us to specify other parameters, such as the number of registers, DRAM size, network latency, and network bandwidth. Section 3.3 explains the effects of these properties in greater detail.

```
<machine>
  <prop key="Cores" val="1000" />
  <prop key="Gflop/s/core" val="10" />
  <prop key="GB/s/core" val="1" />
  <prop key="Cache/core (kB)" val="64" />
  <prop key="Division Cost" val="39" />
  <prop key="Transcendental Cost" val="125" />
  <prop key="NIC BW (GB/s)" val="100" />
  ...
</machine>
```

Listing 1. An example XML machine description (partial)

Additionally, some software parameters that affect performance, such as the use of cache-bypassed writes and non-temporal memory accesses, may be configured in our performance model through XML input or at runtime.

3.3 Performance model

Our performance model takes the characteristics of the computational workload specified as an XML and generates performance metrics and execution estimates. For simplicity, we adopt a hardware model abstraction consisting of a collection of parallel hardware cores alongside a parameterized memory on the chip. The CPU is connected to the main memory by a bandwidth-limited off-chip network. Our CPU model does not capture the behavior of individual cores or the on-chip network, but rather takes the aggregate computational throughput as an input parameter. Similarly, the memory model takes the aggregate DRAM bandwidth connecting the CPU to memory (i.e. the stream bandwidth) as an input. Since modeling the effects of on-chip access latency would require a detailed on-chip network design analysis, we focus on the bandwidth-filtering capability of the on-chip memory, i.e. the reduction in memory traffic from capturing temporal locality. Thus, we are primarily interested in the size of the (non-inclusive) on-chip memory capacity per thread or group of threads co-operating on a working set. Our model focuses on capturing the costs of the computational workload and data movement, while taking into account the degree of data reuse enabled by the on-chip memory.

Application performance is estimated using the following method: let α be the aggregate computational throughput of the machine and β be the aggregate memory bandwidth. Let C represent the program's floating-point arithmetic workload and D be the necessary off-chip data movement between the CPU and DRAM. Our model estimates the program execution time as $T = \max(T_\text{c}, T_\text{d})$, where $T_\text{c} = \frac{C}{\alpha}$ is the CPU time and $T_\text{d} = \frac{D}{\beta}$ is the DRAM time. This performance metric assumes the full throughput and bandwidth are achievable, which may not always be the case for a complex application code. The purpose of our framework is not to make exact performance predictions, but instead provides a performance upper-bound in the spirit of the Roofline model (Williams et al., 2009), and is useful for making relative comparisons between different hardware–software configurations. Lastly, we modeled the off-node communication time by assuming an ideal interconnection network. Our model estimates the communication time as $\frac{m}{b} + l$, where m represents the aggregate message size, b is the network injection bandwidth, and l is the network latency. Thus, for large messages, the network latency is negligible. The model also computes the fraction of communication time over the total execution time, which depends on the T_c (on a memory bandwidth-limited kernel) and T_d (on a compute-bound kernel).

3.3.1 Floating-point computation. In order to estimate C, the floating-point (FP) arithmetic workload, we examine the FP operation distribution present in the code. Current FP logic is typically optimized towards FP additions and multiplications, which exhibit their peak throughput on workloads that only consist of a balance of those two operations. However, there are other types of FP operations present in scientific codes that can only sustain a fraction of the peak. For example, on the Intel Sandy Bridge architecture, the throughput of scalar FP division is 39 times slower than SIMD FP adds or multiplies, while scalar exponentiation is 125 times slower (Vladimirov, 2012). ExaSAT weighs operations such as divides and transcendentals according to their costs specified by the user in the machine configuration to determine a weighted computational workload. Further, the model is parameterized to allow exploring optimizations such as vectorized operations.

3.3.2 State variables, registers, and spills. The number of accesses to both state variables (scalars and non-streamed arrays) and streamed arrays can be used to determine how many registers need to be reserved to hold these values during each of the loops in the program. Since state variables are accessed during every iteration of a loop, an optimal allocation for these variables would place the variables with the most number of accesses into registers, while spilling the rest into the next tier of memory (e.g. L1 cache or local memory). Assuming an architecture with an L1 cache, our performance model can compute the traffic that results from spilled state variables based on the user-specified register parameters. In addition, it can compute mandatory traffic that results from streamed variable access to estimate total L1 traffic. This information can be used to analyze the trade-off that results between the number of available registers and the L1 bandwidth.

3.3.3 Working sets and memory traffic. The performance model analyzes every array accessed in each loop of the input XML code description. Each array may have a different access pattern, so the tool computes working set and bandwidth usage for each array independently given the array's access pattern. An array that is written will typically only require access to the current grid element (no neighbors), while arrays that are read may require multiple grid elements. Our memory and cache model is targeted to the reuse pattern that occurs in stencil computations because stencils constitute the most prevalent operator in our target application codes. The cache is assumed to be an ideal, fully-associative least recently used (LRU) cache, which is optimistic in the sense that if the working set fits into cache, full reuse of that working set is assumed. Real caches with random replacement policies are likely to underperform due to conflict misses and imperfect replacement. However, our model provides a performance ceiling and a starting point for more detailed analysis using dynamic instrumentation and simulators.

Figure 4 shows the potential reuse cases captured by our model for the canonical seven-point stencil. If the cache is large enough to hold the cell working set, then there will be reuse between cell iterations. Similarly, the figure shows the working set sizes needed for reuse between pencil iterations (all points in x for a given y and z) and plane iterations (all points in x and y for a given z). For each stencil access pattern encountered in

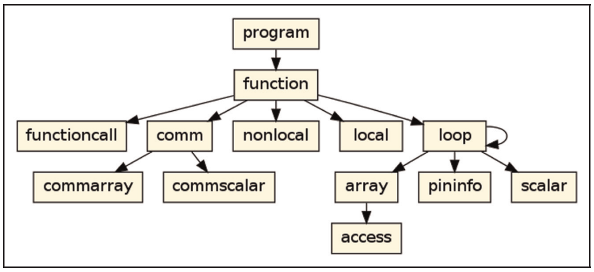


Figure 3. XML-IR element node hierarchy.

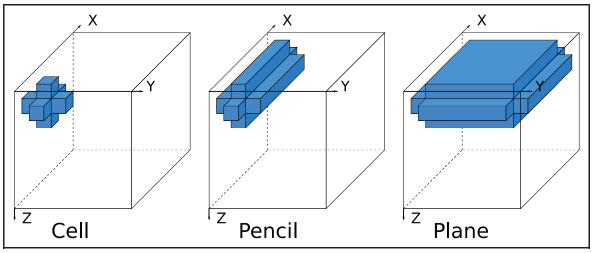


Figure 4. Working sets required for different levels of reuse for a seven-point three-dimensional stencil. The grid is swept in a triply nested loop with the *x* dimension first, then *y* and then *z*.

the code, our model computes the working set sizes required for each of these reuse cases.

If there are gaps in the stencil access pattern, partial reuse may occur near the calculated boundaries between reuse cases. Our model can compute the working set sizes that bound the transitions from no reuse to partial reuse, to full reuse between pencil and plane sweeps. For an LRU cache, no reuse will occur if the cache is smaller than the number of elements accessed in the pattern, while full reuse requires a working set equal to the span of the pattern plus the maximum gap size. For example, a stencil pattern that accesses planes -2, -1, 0, $+1$, $+2$, has a working set of five planes because there's no gap, but a pattern of -2, $+2$ requires a working set of eight planes (five for span, three for gap) for reuse even though it only touches two planes per sweep. It may seem counter-intuitive that accessing fewer planes can increase the working set size, but gaps in the pattern require the cache to hold data for a longer period of time without evicting it. For a software-managed local store, the memory can be managed more efficiently, requiring only the span of the access pattern to fit into the store. Since we are interested in establishing a performance upper bound, the model optimistically assumes full reuse is possible for certain situations where only partial reuse would occur. Future work will take these effects into consideration to increase the tightness of the bound.

The machine configuration specifies the cache line size, which determines the minimum granularity of access in the unit-stride dimension used for working set and bandwidth calculations. Our model rounds the number of contiguous elements within the accessed region up to the next multiple of the cache line size (assuming optimistic alignment), to compute the resulting working set and memory traffic estimates. Also, the configuration allows the user to specify whether cache bypass is utilized for array writes, reducing memory traffic and cache pollution. Non-temporal array reads can also be enabled in the configuration to further reduce cache pollution from arrays with no reuse.

Once the working set and memory traffic estimates are computed, they are compared to the cache size specified in the hardware configuration to determine what reuse scenario will occur for each loop, thus determining the required memory traffic for the whole program. Note that this type of analysis can be conducted at every level of the cache hierarchy. For example, if we specified the cache size available at L1, then the computed memory traffic would be that required between L1 and L2. Using our methodology, we could conduct a multi-level analysis that computes the bandwidth requirements and performance at every level of cache.

3.3.4 Block execution schemes. Cache blocking (Rivera and Tseng, 2000) reduces cache capacity misses by tiling the loop iteration space, thus shrinking the working set to the point where it fits in cache. ExaSAT incorporates two different block execution schemes to analyze the performance impacts of cache blocking. In the traditional blocking scheme, each loop runs over the entire domain before proceeding to the next loop. In an alternative scheme (Woodward et al., 2010) all of the loops are run on a block before moving to the next block, as illustrated in Figure 5. Each large rectangle represents the iteration space at different points of progress (indicated by shading), and each subrectangle represents a block of the iteration space that fits into local memory. While traditional blocking allows reuse of data within loop nests, the alternative scheme schedules loops such that reuse of data *across* loop nests is also possible. The potential disadvantages of the alternative scheme are larger working set sizes and redundant computation needed to satisfy any necessary spatial dependencies between blocks. In the alternative scheme, if the blocks are sized appropriately, all temporary arrays can remain in cache or local store throughout the computation until the final output is produced. If there is sufficient on-chip memory, the only DRAM traffic required would be for reading and writing each function's inputs and outputs.

ExaSAT automatically generates parameterized performance models for both schemes, facilitating the exploration of optimal strategies for different machine configurations in the co-design process. Using liveness analysis, our performance model can estimate the total memory footprint needed at each computation step,

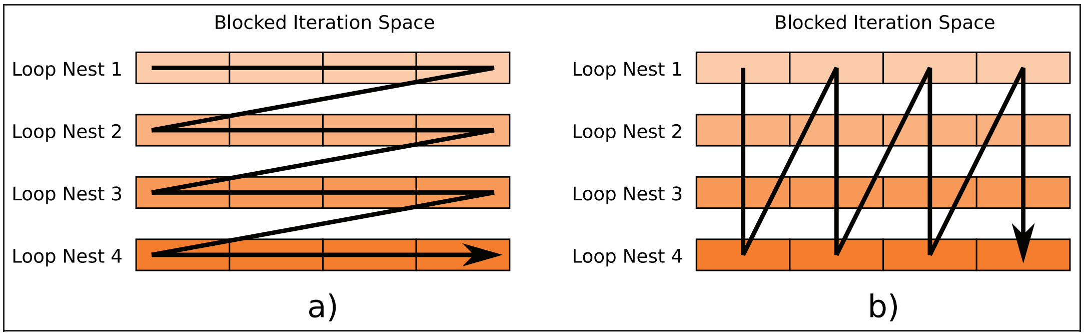


Figure 5. Comparison between a) traditional blocked execution order, and b) the alternative block execution order.

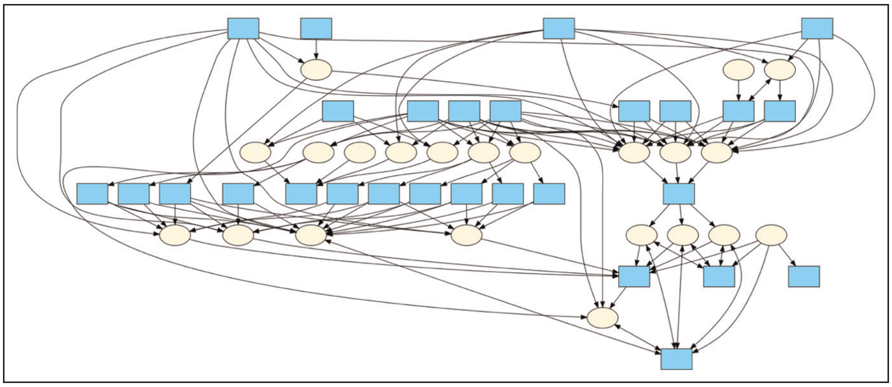


Figure 6. Dependency subgraph for the SMC dynamics code.

giving the on-chip memory size required for each block execution scheme and an estimate of the total memory traffic.

3.4 ExaSAT Outputs

3.4.1 Dependency graph description. The ExaSAT framework outputs a dependency graph indicating the dependencies between loops in a procedure. Flow, anti, and output dependencies are considered across all arrays read and written in each loop. Figure 6 shows an example dependency graph generated by ExaSAT where boxes represent data arrays, ovals represent loops, and arrows indicate which arrays are read and written by each loop. The dependency graph illustrates the code's inherent concurrency and allows us to reason about how the computation can be rearranged for enhanced locality, task co-scheduling, and parallel load distribution. We explore the impact of loop fusion for enhanced locality on our motivating application in Section 5.1.4. Future work will study the use of intelligent runtime analysis for task co-scheduling and load balancing.

3.4.2 Spreadsheet description. ExaSAT outputs a performance spreadsheet for the user to further examine the performance of the application. The spreadsheet contains a table of user-modifiable parameters, which allows the user to change the initial XML software and machine configurations. The rest of the spreadsheet automatically updates itself via formulas to reflect the changes made in the parameter table.

The main section of the spreadsheet is a summary table listing properties for each loop in each procedure in the code. Table 1 shows a part of the summary table generated by our tool, including flop counts, state variable count, working set size, memory traffic, and execution time. Aggregate statistics are also included in the table to summarize whole program performance.

The spreadsheet contains an array access and occupancy table which shows the liveness of arrays through the progression of the program. This analysis is used for memory capacity calculations and NVRAM feasibility studies. Table 1 also shows an example occupancy table generated by our tool. The rows in the table correspond to arrays, while the columns correspond to loops in the program, allowing the table to be read left-to-right to correspond to a possible program execution. The number of copies indicates the number of components of the arrays. Each cell in the table contains one of the following values: read (R), written (W), read-then-written (RW), written-then-read (WR), live (L), or non-resident (). Summary columns are given to show the number of reads and writes to each array as well as the total number of live arrays, which is used to compute the memory footprint.

Other sections in the spreadsheet include tables that summarize the internode communications that must occur during the program execution and state variable accesses to help model the cache traffic resulting from spilled registers. A summary table and histogram are generated for each loop showing the number of state and streaming variables located in registers versus cache and the corresponding number of register hits and misses.

3.5 Model validation

We validated our results against data collected through dynamic instrumentation and benchmarking.

Validation against Pin: First, we used the publicly available Pin tool (Luk et al., 2005) to validate instruction counts and memory traffic. Pin analyzes an application at the instruction level and uses dynamic compilation to instrument executables while they are running. By attaching callbacks around every instruction reading or writing to memory we can extract a stream of load and store addresses from the program as it runs. This stream is then piped into an LRU cache simulator that we implement on top of Pin, which aggregates the relevant statistics such as cache hit, miss, and line writebacks for a given cache size. Floating-point instructions are also monitored to retrieve flop counts.

The FP instruction counts predicted by ExaSAT match with those measured by the Pin tool. Loads and stores of the variable under study also match those reported by the Pin tool. Figure 7 compares the memory traffic modeled by ExaSAT with the memory traffic captured by the Pin tool for the CNS code for various cache sizes. CNS[2] is a combustion proxy that integrates the compressible Navier–Stokes equations assuming constant transport. It is a simplified (single species) version of the SMC code (Emmett et al., 2014), which will be discussed in more detail in Section 4. The analytical performance model in ExaSAT correctly captures the amount of data reuse and the resulting trend of memory traffic as cache size is varied, though the memory traffic modeled by ExaSAT is slightly lower than the Pin tool's because ExaSAT is providing a lower bound. Initially, the number of L1 cache hits measured by Pin was abnormally higher than what ExaSAT estimates considering only array access traffic, which led us to investigate the proportion of L1 cache traffic due to spilled state variables. When there are not enough registers to hold all the state variables in a loop, accesses to these variables will be spilled to the next level memory. This introduces more cache traffic, which will give the impression that there is a higher hit rate. When we separated array accesses from the state variable accesses to the cache in the Pin tool, the loads and stores estimated by ExaSAT matched with those measured by Pin. Register spills are also discussed in Section 5.1.3.

Block size validation: Second, we measure the effect of blocking with three simple stencil benchmarks, namely gradient, divergence, and Laplacian and compare their performance against the estimates by ExaSAT. We manually blocked three simple stencil benchmarks, and collected the execution time with 24 threads on a single node on NERSC Hopper Cray XE6. No software prefetcher or cache bypass is enabled. The results in Figure 8 show that the measured execution times and optimal blocking size correlate well with ExaSAT's. Where the block size is small, the model predicts much better performance than the measured because in the measured code, hardware prefetchers cannot hide the load latencies for small blocks. There are also situations where the model exceeds the measured execution time. The model has sharp transitions at the points where the working set grows larger than the available cache. In reality, the cache replacement policy leads to a smoother transition than ExaSAT.

Optimization opportunities: Third, we collected running times for the CNS code (single species) and SMC code (multiple species) and compared them against the ExaSAT estimated bounds. The purpose of comparing ExaSAT with the benchmark is not to measure how close the estimated and actual running times are, but to point to the parts of the code where there are opportunities for optimization since ExaSAT highlights parameter sensitivities subject to the user-specified constraints rather than giving a performance prediction.

Figure 9 compares the performance bounds by ExaSAT and the actual running times by loop nests collected on the NERSC Hopper machine.[3] As clearly seen from the results, a big performance gap exists between the two for some of the loops and for these there is the potential to gain the performance back through data movement optimizations. In particular, the *hypterm* function exhibits the largest discrepancies between the measured and estimated bound. ExaSAT bounds the running time for the *hypterm* function to 8.1 ms, which is 3.3 × better than what is measured (26.6 ms). ExaSAT estimates that this bound for *hypterm* can be further reduced to 3.3 ms from 8.1 ms if all three loops are fused.

We aggressively optimized the *hypterm* function by applying vectorization, cache blocking, and loop fusion optimizations and the results are shown in the inset graph in the same figure. The first bar in the inset graph shows the total time spent in three loops in *hypterm* and the second bar shows the measured performance as a result of optimizations and compares it with the new

Table 1 Example loop analysis table (top) and array access and occupancy table (bottom) generated by the ExaSAT tool for a subset of SMC dynamics code.

Procedure	Loop line number	flops/cell				State Var		Working set (kB)	Memory traffic (GB)	FP Computation (weighted Gflops)	B:F	Execution times (ms)
		Add	Mul	Div	Exp	Int	FP					
advance	418	128	174	0	0	17	5	356	0.69	0.63	1.16	0.67
advance	533	2	2	0	0	7	2	0.03125	0.11	0.01	12.00	0.11
advance	720	32	39	0	0	13	8	132	0.19	0.15	1.39	0.19
advance	771	18	27	9	0	17	0	0.4375	1.41	1.00	1.52	1.37
advance	1529	860	959	18	0	30	70	818	1.44	5.33	0.29	1.41
ctoprim	85	3	17	1	0	24	32	0.375	1.54	0.15	11.12	1.50
ctoprim	136	4	4	2	1	14	22	0.1171875	0.23	0.44	0.57	0.23
Total/Max								818	5.61	7.71	0.78	5.48

		Loop line number												Totals		
Variable name	Copies	274	418	515	767	771	791	1139	1160	1508	1529	1877	1921	Reads	Writes	Live
Fdif.iryn	53			W	L	L	L	RW	L	RW	L	RW	R	212	212	265
Fhyp.iryn	53	W	RW	L	L	L	L	L	L	L	L	L	R	106	106	477
Hg.iryn	53						W	R	W	R	W	R		159	159	0
Q.qhn	53					R	R	L	R	L	R			212	0	106
Q.qpres	1		R	L	L	R	R	L	R	L	R			5	0	4
Q.qtemp	1						R	L	R	L	R			3	0	2
Q.qxn	53					R	R	L	R	L	R			212	0	106
U.iryn	53	L	R	L	L	L	L	L	L	L	L	L	R	106	0	530
Unew.iryn	53	L	L	L	L	L	L	L	L	L	L	L	RW	53	53	583
dpe	1				W	RW	R	L	R	L	R			4	2	2
dpy.n	53					WR	R	L	R	L	R			159	53	106
Number of arrays resident		159	160	213	214	373	427	427	427	427	427	265	212			

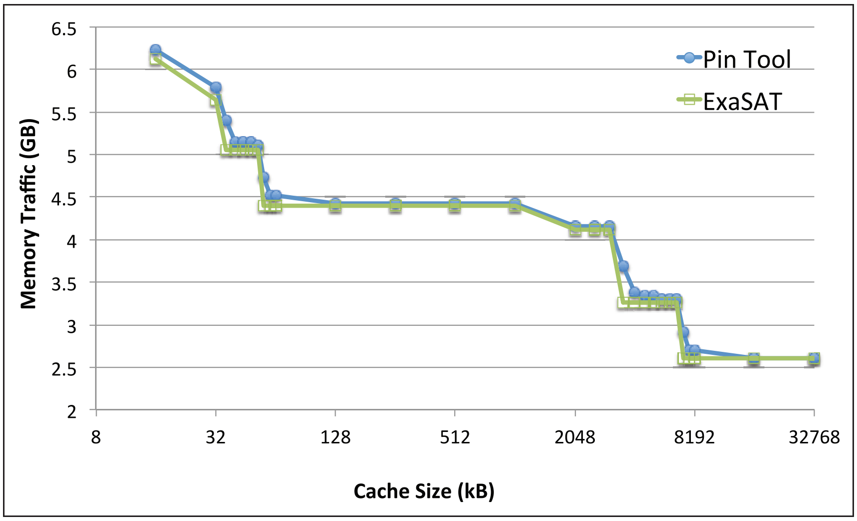


Figure 7. Comparing memory traffic modeled by ExaSAT and simulated by Pin for the CNS code (128^3 problem).

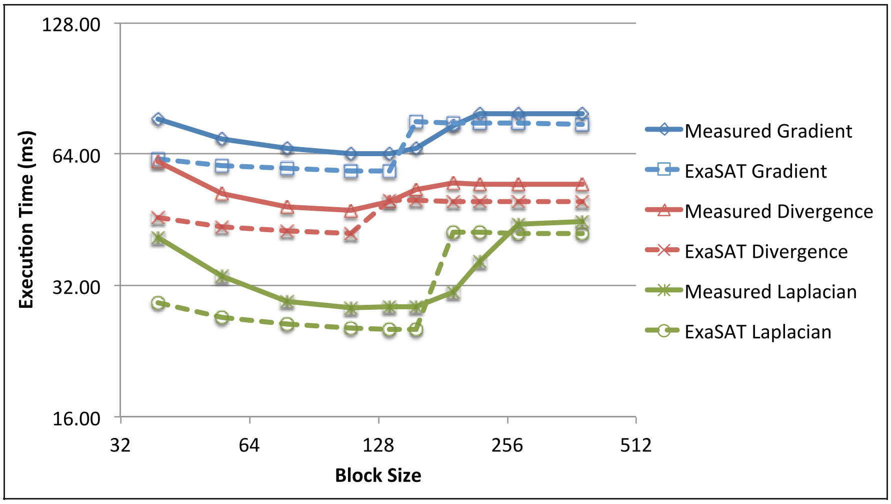


Figure 8. Measured and modeled execution times for blocking sizes for three simple stencil benchmarks.

bound by ExaSAT after the loop fusion. The manual optimizations achieved 6.7× the initial performance, reducing the measured running time from 26.6 ms to 4.0 ms, which is much closer to what ExaSAT predicts (3.3 ms). This illustrates how ExaSAT can be used to identify performance opportunities for the programmer and guide application tuning.

Similarly, ExaSAT suggested that tiling the SMC code would provide a 37% improvement in performance on Hopper and a 41% speedup on San Diego Supercomputer Center's Trestles.[4] We have implemented a tiled version of SMC and observed a 30% improvement on Hopper and 32% on Trestles. We suspect that the poorer measured performance can be attributed to limitations with the hardware prefetchers since the SMC code accesses a large number of arrays in its solvers. Consequently, there is room for improvement and we are still investigating the SMC performance. We did not manually implement the fused version of SMC because of its complexity. We plan to use the CHiLL compiler framework (Chen et al., 2008) to automate loop fusion.

4 Motivating application: SMC

We demonstrate the abilities of the ExaSAT framework by applying the tool to the SMC code, which contains over 10K lines of code, making manual analysis impractical for this code. SMC was developed by the Combustion Co-design Center and is a proxy for the production direct numerical combustion codes such as S3D (Chen et al., 2009). SMC represents structured

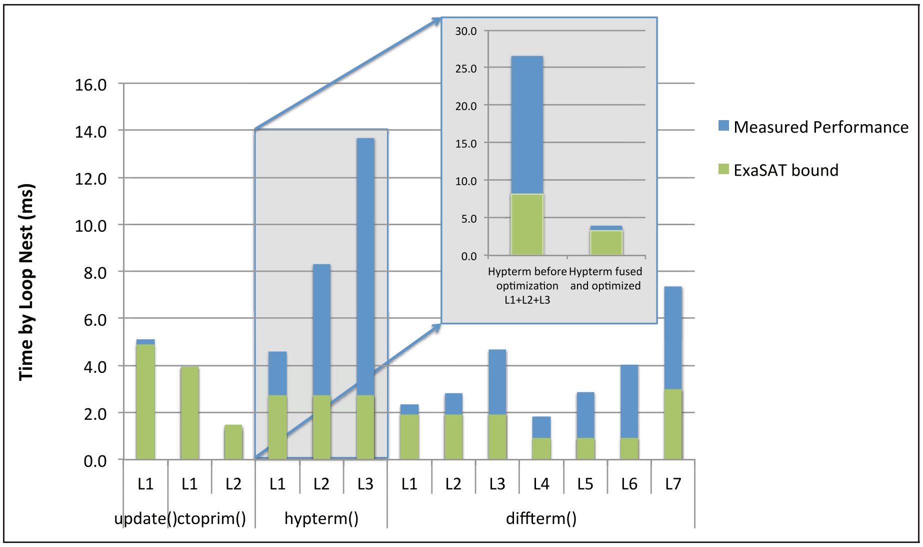


Figure 9. Measured and modeled execution times for each loop in a single Runge–Kutta step in the CNS code. *L#* indicates the loop number.

grid problems, which play an important role in numerical simulations, particularly in stencil-based PDE solvers. Understanding SMC performance provides insights into the requirements of this family of combustion codes on exascale machines.

SMC integrates the multi-component reacting compressible Navier–Stokes equations with detailed models for chemical species diffusion and kinetics. It contains the key elements of both the dynamical core[5] and the chemical kinetics components of S3D; however, SMC is restricted to gas-phase problems and a restricted set of boundary conditions. SMC also uses a simpler temporal integration algorithm that does not include automatic error control. The algorithm is based on the high-accuracy solution of a system of partial differential equations of the form

$$\frac{\partial U}{\partial t} + \nabla \cdot \mathcal{F}(\mathcal{U}) = \nabla \cdot \mathcal{D}(\mathcal{U}) + S$$

The terms $\mathcal{F}$, $\mathcal{D}$, and S correspond to hyperbolic transport, non-linear diffusive processes and chemical source terms, respectively. U is a vector of unknowns, representing density, energy and three components of momentum with an additional density for each chemical species (e.g. octane), for a total of $5 + N_s$ unknowns per point where N_s is the number of species in the problem. The number of chemical species and the number of reactions have a strong effect on the overall computational costs of the algorithm; typical applications will range from as few as 9 species to more than 100. The chemical kinetics model used by SMC is specified at compile time using code that is generated automatically from a tabular description of the reaction mechanism. This mechanism-specific file also includes thermodynamic data needed for the simulation. Transport coefficients are computed using EGLIB (Ern and Giovangigli, 1995).

We focus on two important aspects of SMC: the chemical source term evaluation and the dynamical core. The chemical source term of SMC is a computationally intensive, element-wise computation that uses a large number of transcendental operations. The dynamical core uses high-order stencil computations to approximate spatial derivatives, converting the system into a large system of ordinary differential equations. These ordinary differential equations are then integrated using a third-order, low-storage, TVD Runge–Kutta scheme (Gottleib and Shu, 1998; Qiu and Shu, 2005).

The spatial discretization uses a finite difference approximation on a uniform grid. There are essentially three types of term we need to approximate: first-order derivatives needed to approximate $\nabla \cdot \mathcal{F}$, and terms of the form $(au_x)_y$ and $(au_x)_x$, both of which arise in discretizing $\mathcal{D}$. We first define a first-order derivative operator in the x direction, $D^1,^x$ using an eighth-order finite difference discretization

$$u_{x,i,j,k} \approx D^{1,x} u_{i,j,k} = \sum_{\ell = 1,4} \alpha_\ell (u_{i+\ell,j,k} - u_{i-\ell,j,k})$$

with analogous operators in the y and z directions. These discrete derivative operators are used to evaluate the terms for discretization of $\mathcal{F}$. They are also used to evaluate mixed derivative terms. For example

$$(\eta u_x)_y \approx D^{1,y}(\eta D^{1,x} u)$$

The second derivative terms are discretized using an eighth-order extension of the narrow stencil discretization of Kamakoti and Pantano (2009). In particular,

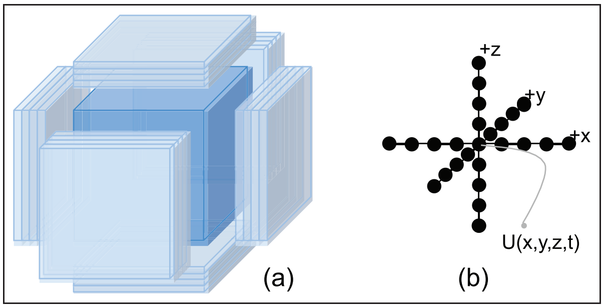


Figure 10. (a) Three-dimensional grid with its ghost cells, and (b) stencil access pattern for SMC.

we approximate variable coefficient second derivative terms in the form

$$\frac{\partial}{\partial x}\left(a\frac{\partial u}{\partial x}\right)_i \approx D^{2,x}(a,u) = \sum_{\ell,m=-4,\dots,4} \beta_{\ell,m} a_{i+\ell,j,k} u_{i+m,j,k}$$

A more detailed discussion of the discretizations in SMC can be found in Emmett et al. (2014).

The parallel grid decomposition for the SMC code requires *ghost cell* exchanges for the vector U. Ghost cells are the data residing in neighboring grid blocks that are required to compute the stencil operations. Figure 10 shows the stencil access pattern and ghost region that needs to be communicated. The depth of the ghost region is four grid cells in each dimension with $5 + N_s$ values per point.

5 Results

5.1 Analysis of SMC with ExaSAT

In order to capture the effect of increasing the number of species, we modeled the SMC code for 9, 21, 53, 71 and 107 species, representing simulations ranging from hydrogen to natural gas to biofuels. Figure 11 provides more description of the species modeled in this paper. Note that only certain values for the number of species are meaningful. Unless stated otherwise, the baseline performance estimates are based on a domain decomposition (box) size of 128^3 per node with 53 species using the machine configurations specified in Listing 1.

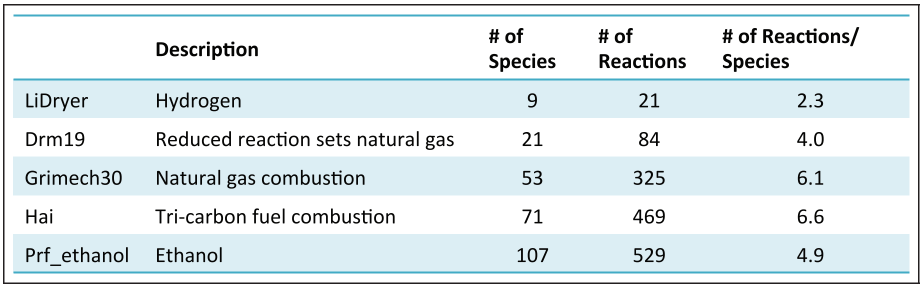
5.1.1 Arithmetic operations. ExaSAT can examine the FP operation mix per loop iteration and provides the flexibility to change the arithmetic operation throughput. Figure 12 shows the operation analysis for both the chemistry and dynamics kernels of the SMC code for a 128^3 problem size with 53 species. The two kernels exhibit substantially different arithmetic operation distributions. The chemistry kernel contains transcendental operations, mainly exponentials (92.5%) and logarithms (7%). Even though a small number of division and transcendental functions appear in both kernels, these operations contribute significantly to the running time since they execute roughly one to two orders of magnitude slower. Figure 12 shows the estimated contribution of each FP operation to the CPU time when we assume vectorized addition and multiplication, and low throughputs[6] for division and transcendental functions (1/39th and 1/125th of peak, respectively). Note that the CPU time (T_c) is computed based on the compute throughput and does not include the DRAM time. Even though transcendental functions in the chemistry kernel are a small fraction of the total flops, they dominate the CPU time (75%). Similarly, the number of divisions in the dynamics kernel seems insignificant but represents one third of the CPU time.

5.1.2 Fast transcendentals and division. Vectorization is one of the main sources of parallelism within a processor that can enable fast execution of FP division and transcendental arithmetic. Besides parallelism benefits, it can also lower energy and control complexity. The downside is that it takes chip surface area and requires programmer assistance. An alternative approach to vectorization is software pipelining, which can hide functional unit latency but also requires more programming effort and more registers.

Table 2, shows benchmarked (not modeled) performance results gathered on the Intel Sandy Bridge E5-2680. *Fast-div* shows the performance improvements for division using the SSE instruction (AVX provides no further performance gain) and *Fast-exp* shows the performance improvement for the exponential function with the AVX Short Vector Math Library. The benchmark results indicate that SSE provides a 1.95 × improvement on division and AVX provides a 2.98 × improvement on exponentials.

ExaSAT allows a user to weight instructions based on their relative throughput to the peak compute rate. The weights can reflect the longer execution times of certain instructions such as division, or they can reflect the potential speedup through improvements to the compilers or hardware. The speedup due to the use of vector intrinsics may not be proportional to the increased weight of the instruction speed because the compiler might fail to generate code that uses vector intrinsics due to complex loop body or divergence effects.

Figure 13 shows the estimated speedup for the SMC code including both the chemistry and dynamics codes as a result of different SIMD lengths. Here, the baseline performance assumes SIMDized addition and multiplication, and low throughputs for division and transcendental functions (first column in Table 2). Figure 13 also shows the estimated speedup for SMC on the Sandy Bridge (indicated as a line) using the benchmarked costs for division and transcendentals shown in Table 2. Our performance model takes the maximum



	Description	# of Species	# of Reactions	# of Reactions/ Species
LiDryer	Hydrogen	9	21	2.3
Drm19	Reduced reaction sets natural gas	21	84	4.0
Grimech30	Natural gas combustion	53	325	6.1
Hai	Tri-carbon fuel combustion	71	469	6.6
Prf_ethanol	Ethanol	107	529	4.9

Figure 11. Number of species, number of reactions and a description of the chemistry component of the SMC codes modeled.

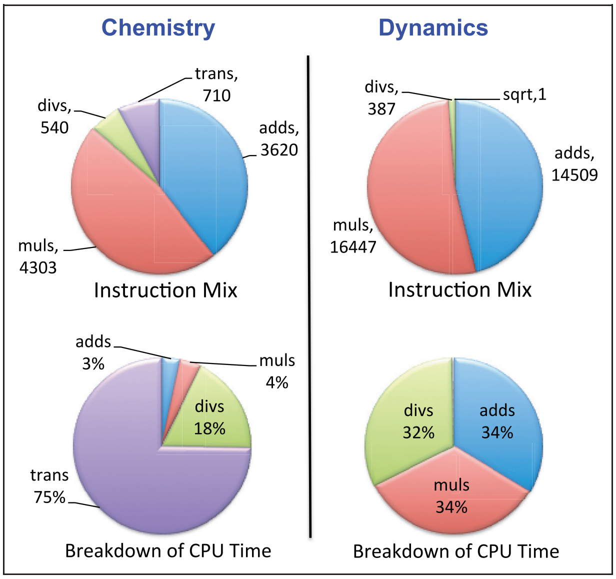


Figure 12. Floating point operation mix and breakdown of CPU time T_c modeled by ExaSAT for chemistry and dynamics kernels.

CPU time and DRAM time for each kernel independently to compute the execution time. Both vectorized division and transcendentals greatly improve the execution time of the chemistry code; however, there is no benefit for the dynamics code since its execution time is limited by DRAM bandwidth. As a result, there is a diminishing return as we increase the SIMD length. For example, for 53 species, the SSE instruction (SIMD2) provides 25%, while SIMD4, SIMD8 and SIMD16 give 43%, 54%, and 60% improvement over the baseline, respectively. The improvement differs between the different number of species because of the number of reactions, thus the number of divisions and transcendentals differ. Both 53 and 71 species have a high number of reactions per species, which means more arithmetic operations and higher benefit from vectorization for the chemistry component.

5.1.3 State variables. The state variable analysis provided by ExaSAT is valuable in the co-design process because it exposes a hardware trade-off between register count and L1 cache traffic (or local memory traffic). In order to measure how many registers the SMC code requires to avoid spills, we collected all the state variables and their access frequencies for each loop using compiler analysis. Based on the number of registers specified by the user, the performance model allocates the state variables to available registers and computes the L1 cache traffic resulting from the register spilling. Figure 14 shows the number of accesses for each FP state variable sorted by number of accesses in the SMC chemistry kernel. For example, in a nine-species simulation, the variable #22 is accessed 15 times. In the best-case scenario, the compiler will allocate the variables with the highest number of accesses to the available registers. Assuming 16 FP named registers (as in SSE or AVX), the vertical dashed line shows the cut-off between variables that would be allocated to registers (left of the line) and those that are spilled to cache (right of the line).

Figure 15 shows the percentage of state variable accesses spilled to the next level memory as the number of available registers is varied. In the 16-register example, about half of the accesses are fulfilled from registers and half go to cache for each of the five chemistry species shown. In the dynamics kernel (not shown in the figure), even though the total number of state variables is much smaller, assigning the top 16 variables to registers only reduces the number of cache accesses by about half since access rates remain fairly high for the top 30 to 40 variables for many loops. Since the chemistry code has a relatively low streaming data requirement compared to the dynamics code, spilled state variables make up greater than 95% of the L1 cache traffic if there are 16 registers. It is possible to filter additional cache traffic by adding registers to the architecture, which would move the cut-off line in Figure 14 to the right. Having 256 registers per thread (as in NVIDIA's Kepler GPU[7]) would filter 88% or more of L1 cache traffic due the state variable for the SMC chemistry code, and 94% or more for the SMC dynamics code. It is important to note that the spills must be balanced against the performance cost of a large register file. The optimal performance point may be reached at an earlier point.

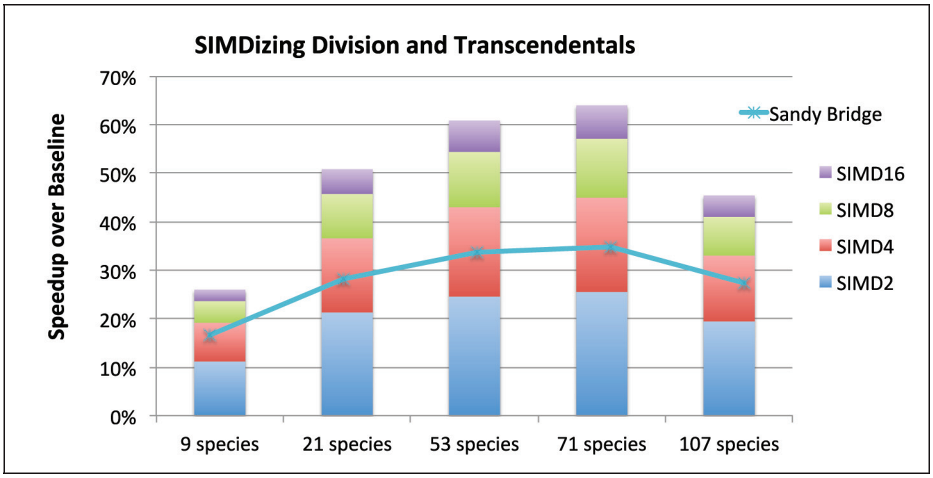


Figure 13. Estimated overall SMC speedup over baseline as a result of SIMDizing division and transcendental functions using different vector lengths. Baseline indicates vectorized addition and multiplication operations running at the peak compute throughput, but no vectorization for division or transcendentals.

Table 2 Relative throughput of divide and exponential compared to vectorized ADD on Intel Sandy Bridge E5-2680 with Turbo Boost.

Relative throughput	Baseline	Fast-div	Fast-exp
Division	1/39	1/20	—
Exponentials	1/125	—	1/42

5.1.4 Memory traffic and working sets. **Cache blocking**: ExaSAT can model the effect of cache blocking on the working set size and memory traffic without manually implementing this optimization. Blocking the iteration space shrinks the size of the working set to enable temporal data reuse. If the reduced working set fits within the available on-chip storage, the memory traffic due to the capacity misses can be greatly reduced. A trade-off of cache blocking is the induced memory traffic for the ghost cells. As the block size is decreased, the redundant traffic to pull the ghost zone increases. Finding the optimal blocking factor for a given cache size is an optimization problem for compilers, auto-tuners and runtime environments. In this context, ExaSAT can guide other programming tools to reduce the search space for blocking factor. We are also interested in illustrating the co-design trade-off of blocking, more specifically the trade-off between cache size and memory bandwidth. For a given cache configuration, ExaSAT can determine a blocking strategy that balances the capacity misses against the additional traffic for the ghost zone.

Figure 16 highlights the change in B:F of the dynamics kernel computed by ExaSAT as a result of blocking for various cache sizes specified by the user. B:F represents the required number of bytes to be transferred off-chip divided by the required flops. The cache size indicates the amount of on-chip memory available per group of threads/cores collaborating on the same working set. Blocking the iteration space reduces the working set size and enables greater reuse. The inflection points in the plot show the points where the working sets no longer fit into the cache. However, using smaller block sizes results in additional memory traffic due to the redundant ghost cell storage and accesses. This effect can be seen even in the unlimited cache case because it is independent of capacity misses. Thus, ExaSAT predicts that blocking with an inappropriate factor could incur more data traffic than necessary. With an optimal blocking factor, a small cache can beat the performance of an unblocked reference implementation on a large cache. Consequently, the compiler or auto-tuner has to find the optimal block size to take full advantage of available cache, while a chip designer has to find a balance between the cache size and memory bandwidth.

Software optimizations: ExaSAT also allowed us to evaluate the performance impact of software optimizations such as loop fusion and the alternative block execution scheme described in Section 3.3.4. Even though loop fusion can reduce memory traffic, it increases the resulting loop's working set, exposing a co-design trade-off between memory bandwidth and cache size. Loop fusion was done by hand, guided by the data dependency graphs generated by the framework, while the cache blocking and alternative block execution schemes were computed automatically from the XML code description. Figure 17 shows the effect of applying various software optimizations on the trade-off space between cache size and the resulting B:F. For small cache sizes, no blocking is used, but there is still some benefit from applying loop fusion to loops that touch the same data. For medium cache sizes, some loops are able to take advantage of reuse within loops in the

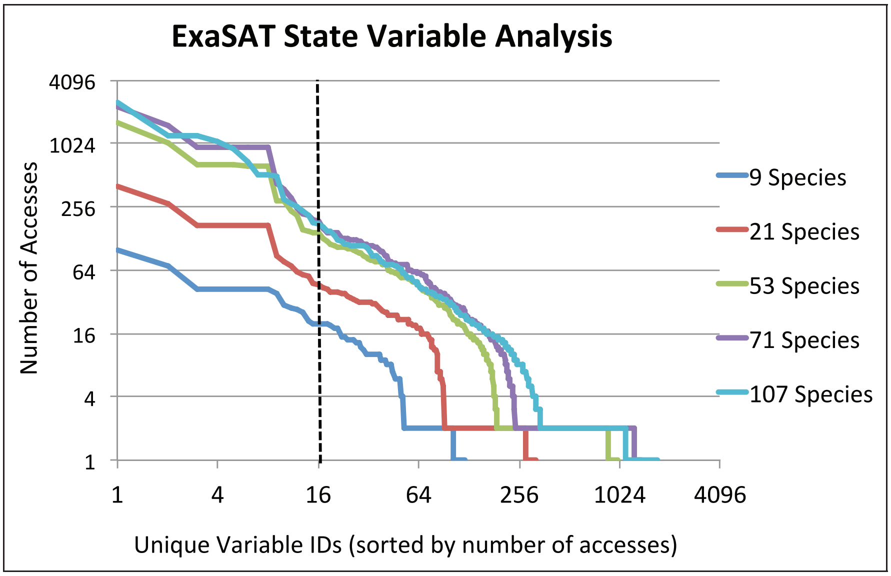


Figure 14. Number of accesses for each FP state variable sorted by their access frequency in the chemistry kernel.

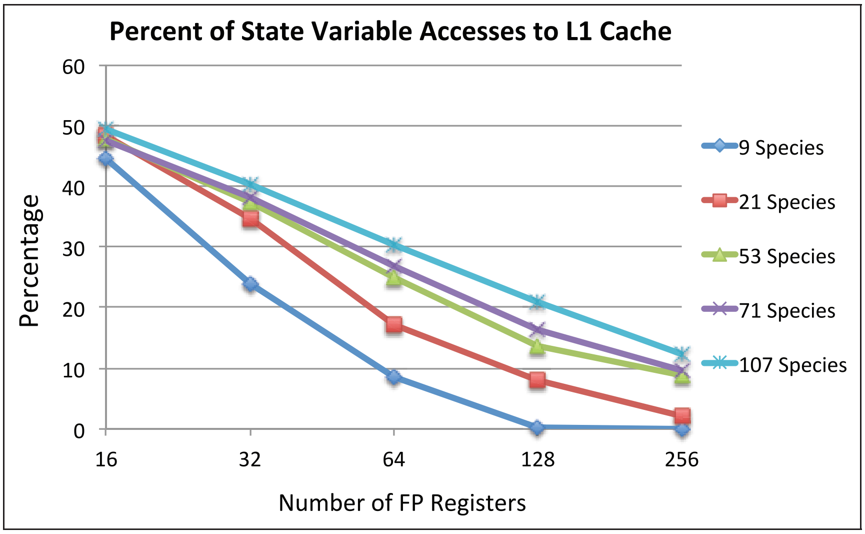


Figure 15. L1 Cache traffic chemistry state variables as the number of the registers is varied. Having more registers can filter cache traffic for state variables.

non-fused case, but there is not enough cache to hold the increased working sets required for fused loop bodies. Once the caches are large enough to contain the increased working sets of the fused loops, fusion becomes beneficial again. For 53 species, the breakpoint is about 2 MB.

The alternative block execution scheme requires the largest working sets because an entire block of data per array must fit in cache (as opposed to a small number of planes per array) to enable reuse *across* loops. However, the benefit from such reuse is a significantly lower B:F (roughly half for the largest cache sizes in the figure). This execution scheme may be most relevant to situations with processing capabilities co-located with large memory banks such as with processor-in-memory and processor-near-memory architectures (Saulsbury et al., 1996). The studied optimizations emphasize the power of software transformations on B:F and their relation to cache size. Not surprisingly as we increase the number of chemical species, the working set size increases (not shown in the figure), requiring a larger cache for fusion to become advantageous. Please see Chan et al. (2013) for a further analysis of software optimizations on combustion co-design.

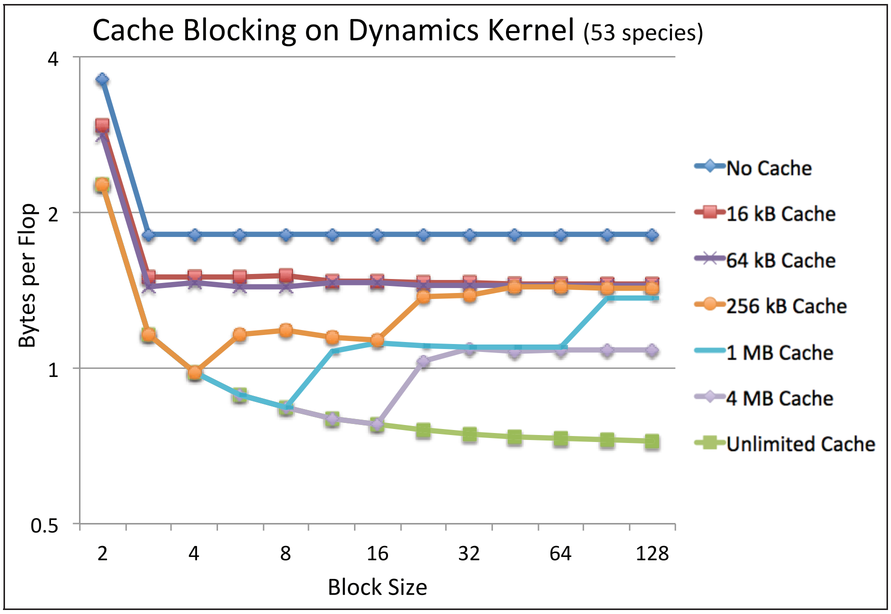


Figure 16. Optimal blocking factor depends on the available cache size.

5.1.5 Memory footprint analysis. ExaSAT can compute the memory required for an application. For SMC, the memory requirement increases linearly with the number of species. A 53-species simulation needs 678 three-dimensional arrays, translating into approximately 13 FP values per species per grid point. Out of 678 arrays, 505 of them have a ghost cell region. Including message buffers, a box size of 128^3 occupies 12.33 GB of memory, which means a 16 GB node can only hold one 128^3 box. Exascale memory capacity is predicted to be primarily constrained by cost (Kogge et al., 2008) which encourages vendors to look for cheaper but denser memory technologies such NVRAM. NVRAM is a cost-effective alternative technology that can serve as a high-capacity, secondary memory. It offers higher density and scalability than DRAM, and uses nearly zero power when in standby mode (Lee et al., 2009). On the other hand, the NVRAM memory cells tend to have a short lifetime. Compared to DRAM, the dynamic write energy is 4 to 40 × worse and the write access latency is an order of magnitude slower (Lee et al., 2009; Qureshi et al., 2009; Caulfield et al., 2010). Nevertheless, without focusing on the details of the NVRAM design, we investigate whether there is sufficient low-write memory traffic for certain variables to justify inclusion of NVRAM in an exascale node, since the specifics of NVRAM design regarding memory endurance, write-voltage and write speed are highly dependent on the technology and are likely to change.

In order to study the NVRAM opportunities in the application, ExaSAT computes the read/write ratio and write access rate of arrays since writes to NVRAM are costly both in terms of performance and energy. In SMC (see Figure 18), there are a number of arrays with low read and low write access rates. We are primarily interested in arrays rather than scalar variables because the idle power consumption is proportional to the memory footprint. If a write access rate of $\leq$ 0.11% is chosen, then a larger fraction of data (75%) qualifies for storage in NVRAM. This would translate into roughly 75% idle power saving. On the other hand, the dynamic energy for these arrays would go up by a factor of 40. Even if a conservative read/write ratio of 5 or higher were chosen, the case for NVRAM would be weak because only 35% of the data would reside in NVRAM. Unfortunately, this is where our analytic model has its limits. To assess whether the dynamic energy consumption overshadows the idle energy savings, power simulators such as NANDFlashSim (Jung et al., 2012) are needed, which is a part of our future work.

5.1.6 Communication analysis. The interprocess communication time as a percentage of total execution time depends on the DRAM time on a memory-bandwidth-limited kernel (or CPU time on a compute bound kernel). Because SMC is memory-bandwidth-limited (see Section 6.1 for details), Figure 19 shows the fraction of communication time for the SMC code as the memory-bandwidth-to-network-bandwidth ratio, δ, is varied. For example, a configuration with 1 TB/s of memory bandwidth and 100 GB/s of NIC bandwidth would correspond to δ = 10, which is an expected value at the exascale. The figure varies δ from 2.5 (a relatively fast network bandwidth) to 40 (a relatively fast memory). According to the analytic results shown in Figure 19, communication time accounts for less than 13% of the

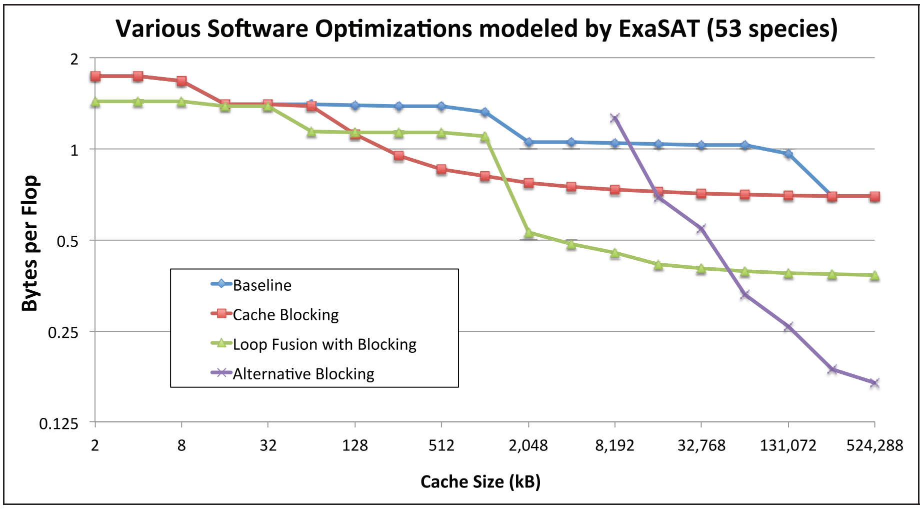


Figure 17. Various software optimizations modeled by ExaSAT for the 53-species SMC dynamics code and the B:F achievable for various on-chip memory sizes.

total time in the worst case and diminishes as we increase the number of species for the SMC code. Because the communication time does not appear to be a performance bottleneck for SMC, there is no strong justification for integrating an NIC on the processor chip to increase the injection bandwidth and reduce latency. Future work will study adaptive mesh refinement codes, where messages tend to be smaller but more frequent. In those cases, we expect there might be more need for on-chip NICs.

The analytic performance analysis ignores the network topology and assumes an idealized network for off-node communication, considering only network latency and injection bandwidth as the performance metrics. However, factors that are hard to capture in an analytic model such as network topology, routing, network contention, and job placement can have a significant impact on performance. We are currently collaborating with Sandia National Laboratory to employ the SST/macro simulator (Rodrigues et al., 2011) to assess the network performance of SMC. ExaSAT serves as a stepping stone for such effort and is used to verify the simulation results. For example, performance on a three-dimensional torus with an optimal job placement agrees with that for the idealized network scenario as used in our analytic model, seeing no network congestion for pure nearest-neighbor communication.

6 Discussion

6.1 Projections on an exascale machine

Figure 20 shows the cumulative effect of the hardware and software improvements modeled by ExaSAT. The estimated effective baseline performance is slightly over 0.5 Tflops using the machine configurations specified in Listing list:machinexml, which is a 10 Tflop node with 1 TB/s memory bandwidth. The SMC code is severely limited by memory bandwidth. Both cache blocking and loop fusion make more efficient use of memory bandwidth, doubling the baseline performance. However, the estimated performance indicates that software optimizations must be supported by hardware improvements at the expense of increased cost and power for the sake of higher performance. If the memory bandwidth is increased from 1 to 4 TB/s, ExaSAT suggests that a 2.5–3 × speedup in the performance is possible. We also modeled the effect of vectorization of division and exponentials for the SMC code. *Fast-div* represents the predicted performance improvements as a result of improved throughput (2 ×) using the SSE instruction and *Fast-exp* represents the improvement for the exponential function by a factor of three with the AVX SVML. While the vectorized division provides a modest performance increase, the chemistry component greatly benefits from the improved exponential function performance. Finally, we changed the network injection bandwidth from 100 to 400 GB/s, which represents a custom NIC that integrates the network controller onto the chip to reduce power and to increase throughput by a factor of 4. Even after the software optimizations and hardware improvements, SMC is still limited by memory bandwidth. In the exascale timeframe, it is unlikely that machines will support bandwidths higher than 4 TB/s, thus more aggressive software optimizations will be needed to reduce data movement and deliver the performance improvements necessary to reach the exascale.

6.2 Implications for hardware design

We evaluated the impact of utilizing vector intrinsics for division and transcendental functions and realized

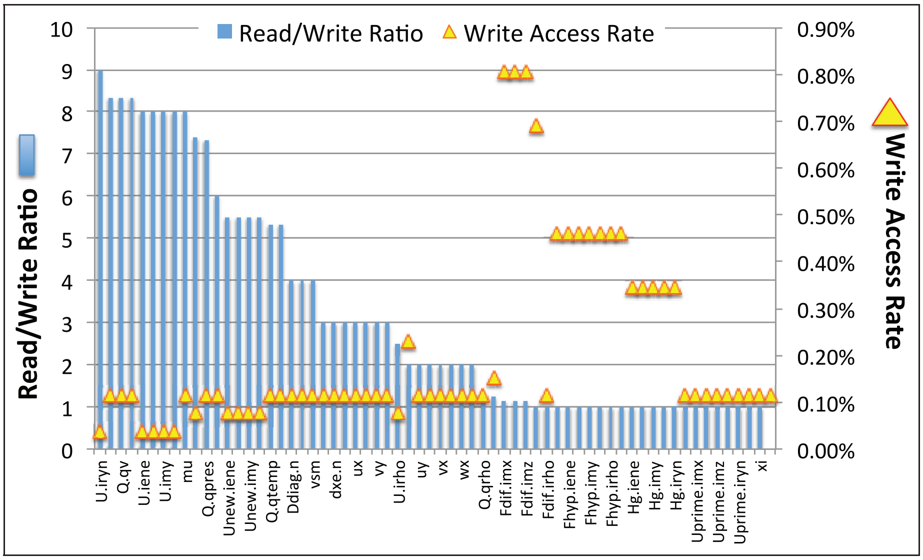


Figure 18. Read/write ratio (left axis) and write access rate (right axis) of arrays in the SMC code.

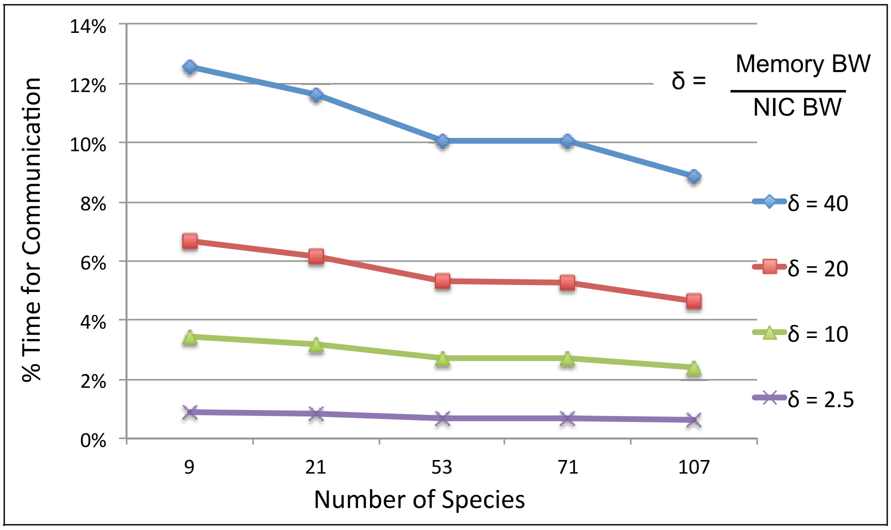


Figure 19. Fraction of communication time for different memory bandwidth/NIC bandwidth ratios. δ = 10 is an expected value for an exascale node. δ = 2.5 represents a relatively fast network bandwidth and δ = 40 represents a relatively fast memory.

that they can greatly improve the CPU-bound chemistry code, provided that compilers or code generators can support vectorization. On the other hand, SIMD lengths more than four do not provide significant performance benefits because the dynamics part of the SMC code is severely limited by the performance of the memory subsystem. For the baseline code with optimal cache blocking, we see very little benefit derived from larger on-chip caches. However, if we adopt a more aggressive approach with loop fusion, we can achieve an order-of-magnitude reduction in memory bandwidth requirements provided there are much larger on-chip memory and register files. For SMC, fusion can reduce traffic by up to 60% versus baseline provided that there is a large enough cache. Having 256 registers per thread would filter 88% or more of the register spills due to the state variables.

In our assessment of data accesses, given that technology allows NVRAM write performance to improve, we see some opportunities to utilize NVRAM to increase memory capacity with low cost. However, the NVRAM technology has to mature before an investment in software support can be justified. In order to determine which data to place to NVRAM, we argue that the write access rate rather than the read/write ratio should be used as a metric because lower write access rates are better suited to NVRAM. There is a modest performance benefit from the reduced latency

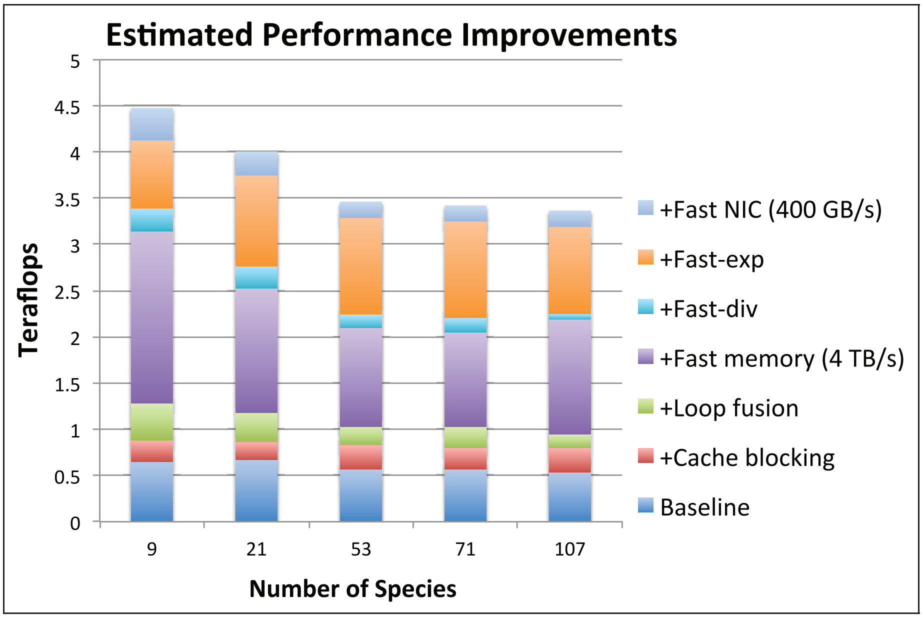


Figure 20. Modeled SMC performance as a result of successive hardware and software optimizations.

and increased bandwidth of on-chip NICs because the network injection bandwidth does not appear to be a performance bottleneck, and the SMC application is insensitive to interconnect latency if the job is placed efficiently.

6.3 Implications for software design

The ExaSAT performance model is a lightweight model and can be integrated into other tools and serve as a cost model. In particular, our analysis of data movement both on-chip and off-chip provides valuable feedback to application, programming model, compiler, and runtime developers. The results emphasize the importance of reduction in memory traffic both for performance and energy reasons. In fact, one of the co-authors of this paper implemented a new blocking optimization in SMC based on our ExaSAT analysis, which yielded an 86× speedup over 1 thread on a 61-core Xeon Phi each running 4 hardware threads (Emmett et al., 2014).

ExaSAT can also provide performance ceilings for compute-bound kernels. Simply having vector units on the chip is not sufficient to increase performance because the compiler also has to generate the appropriate instructions. Current compilers can convert scalar codes into SIMDized programs with some programmer assistance, such as ensuring address alignment and providing compiler directives. Automatic vectorization often fails for complicated loops because other code optimizations may interfere with vectorization or the loop body may be too long to analyze. The highly irregular structure and single-point implementation of the chemistry code currently prevent the compiler from inserting vector intrinsics, especially on GPU-like architectures. The ExaSAT results encouraged the SMC developers to restructure the chemistry component in way to facilitate vectorization by the vendor compiler and this resulted in 2.2× faster chemistry on the Edison machine that includes a 256-bit SIMD (AVX) vector FP. The chemical reactions in the SMC code were previously auto-generated in the order that the species appeared in the input file. Two reactions which have the same number of reactants and the same number of products execute the same instructions with different values. The revised version groups reactions based on the number of reactants and products, which helps vectorization. Similarly, the dynamical core is annotated to provide hints to the compiler for vectorization. The improvement in the performance of the dynamical core is about 1.75× because it is limited mainly by the memory bandwidth, as predicted by ExaSAT.

In addition, we would like to leverage the lessons learned through ExaSAT for the development of a programming model for combustion codes. The new programming model will focus on data structure support for tiling optimizations for data locality and the use of functional semantics to help the runtime reason about data flow and memory use. The goal is to tune the aggressiveness of tiling and fusion optimizations on a given architecture and minimize data movement.

6.4 Future work

An area of future work will be to expand the framework's functionality to cover a broader range of applications besides structured grid problems. For example we are interested in studying dense linear algebra and N -body problems, for some of which a static analysis can be applied. However, the analysis by the compiler and cache model in ExaSAT must be extended to cover their reuse patterns.

One of the current limitations of the framework is how it handles conditionals (Vera and Xue, 2002). Conditionals come in different forms and each form needs to be handled differently. In the codes we analyzed, the branches contain the same number of memory accesses and only the values assigned to the variables are different. Thus, we only need to analyze one of the if-clauses. When conditionals lead to thread divergence (gaps in the iteration space), we would like to be able compute the data movement by weighting each branch. If the branches introduce workload imbalance, the model can be parameterized by a branch-taken probability computed from sample runs or provided by the user.

Although we have made substantial progress in identifying several hardware design trade-offs, there are still a number of co-design questions that remain to be answered. We have formulated plans for comparing analytic model estimates with dynamic analysis and architectural simulators to obtain more accurate results. Some of these plans include more detailed core model, on-chip network, NVRAM power modeling and network job placement strategies. Another exascale co-design challenge that we have already started evaluating is whether the software or hardware should take responsibility for fault tolerance. Hardware-managed resilience mechanisms increase the overall system cost and power consumption. We are extending our analysis of data access patterns to compute data movement requirements of different checkpoint schemes for software-managed resilience. Finally, given that our methodology allows us to address hardware requirements for the SMC combustion code, we would like to extend the ExaSAT framework to examine the requirements for adaptive mesh refinement codes, such as the low-Mach-number combustion code (Day and Bell, 2000).

7 Conclusions

We have developed the ExaSAT framework to rapidly evaluate exascale proxy applications and accelerate the iterative co-design process. ExaSAT complements more detailed architectural simulation tools through rapid generation of abstract analytic models. It is our belief that analytic models are essential to quickly identify the most productive areas for exploring a complicated multi-dimensional design space, including both hardware and software optimizations. ExaSAT parameterizes *both* the machine model and software optimizations to conduct a sensitivity analysis to guide the co-design process. We demonstrated ExaSAT's ability to perform end-to-end analysis on a combustion proxy application (SMC). The SMC results show substantial opportunities to reduce memory bandwidth requirements by increasing chip area for more registers and on-chip memory. Our analysis illustrates to hardware and software designers the need for higher memory bandwidth and more aggressive software optimizations to reduce data movement. This information can be combined with architectural simulations to understand how our design recommendations change with the energy costs of feasible implementations. Future work will expand the scope of analysis to a wider range of applications and improve the coupling of analytic models with architectural simulation environments.

Acknowledgements

The authors would like to thank Weishen Mead and Matthew Cordery for their contribution to the Pin tool validation.

Funding

All authors from the Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science (contract number DE-AC02-05CH11231). This work is part of the Department of Energy Center for Exascale Simulation of Combustion in Turbulence (ExaCT) and the Department of Energy Co-Design for Exascale (CoDEx) projects.

Notes

1. A variable is live if it holds a value that may be used in the future (thus it cannot be deallocated or overwritten).
2. CNS is available for download at the ExaCT co-design center's website (`http://exactcodesign.org`).
3. Using a six-core AMD Opteron 6172 with 6 MB L3 cache.
4. Using a four-core AMD MagnyCours with 4 MB L3 cache.
5. The part of the code that computes fluid dynamics.
6. Based on the benchmarks we conducted and those conducted by Vladimirov (2012).
7. Kepler has 255 32-bit registers.

References

Ang JA, Barrett RF, Benner RE, Burke D, Chan C, Cook J, et al. (2014) Abstract machine models and proxy architectures for exascale computing. In: *1st international workshop on hardware-software co-design for high performance computing (Co-HPC'14)*, New Orleans, USA, 17 November 2014, pp. 25–32. Piscataway: IEEE Press.

Balaprakash P, Buntinas D, Chan A, Guha A, Gupta R, Narayanan SHK, et al. (2013) Exascale workload characterization and architecture implications. In: *21st high* *performance computing symposia (HPC'13)*, San Diego, USA, 7–10 April 2013.

Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, et al. (2011) The GEM5 simulator. *SIGARCH Computer Architecture News* 39(2): 1–7.

Carrington L, Snavely A, Gao X and Wolter N (2003) A performance prediction framework for scientific applications. In: *ICCS workshop on performance modeling and analysis (PMA'03)*, Melbourne, Australia, 2–4 June 2003, pp. 926–935. Berlin Heidelberg: Springer.

Caulfield AM, Coburn J, Mollov T, De A, Akel A, He J, et al. (2010) Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing. In: *2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis (SC'10)*, New Orleans, USA, 13–19 November 2010, pp. 1–11. Piscataway: IEEE Press.

Chan C, Unat D, Lijewski M, Zhang W, Bell J and Shalf J (2013) Software design space exploration for exascale combustion co-design. In: Kunkel JM, Ludwig T and Meuer HW (eds) *Supercomputing*. New York: Springer, vol. 7905, pp. 196–212

Chen C, Chame J and Hall M (2008) CHiLL: A framework for composing high-level loop transformations. Technical report 08-897, University of Southern California, USA.

Chen JH, Choudhary A, de Supinski B, DeVries M, Hawkes ER, Klasky S, et al. (2009) Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery* 2(1): 015001.

Day MS and Bell JB (2000) Numerical simulation of laminar reacting flows with complex chemistry. *Combustion Theory and Modelling* 4(4): 535–556.

Emmett M, Zhang W and Bell JB (2014) High-order algorithms for compressible reacting flow with complex chemistry. *Combustion Theory and Modelling* 18(3): 361–387.

Ern A and Giovangigli V (1995) Fast and accurate multicomponent transport property evaluation. *Journal of Computational Physics* 120(1): 105–116.

Gottleib S and Shu C (1998) Total variation diminishing Runge-Kutta schemes. *Mathematics of Computation* 67(221): 73–85.

Jung M, Wilson EH, Donofrio D, Shalf J and Kandemir MT (2012) NANDFlashSim: Intrinsic latency variation aware NAND flash memory system modeling and simulation at microarchitecture level. In: *28th IEEE symposium on mass storage systems and technologies (MSST'12)*, Pacific Grove, USA, 16–20 April 2012, pp. 1–12. Piscataway: IEEE Press.

Kamakoti R and Pantano C (2009) High-order narrow stencil finite-difference approximations of second-order derivatives involving variable coefficients. *SIAM Journal on Scientific Computing* 31(6): 4222–4243.

Kogge P, Bergman K, Borkar S, Campbell D, Carlson W, Dally W, et al. (2008) ExaScale computing study: Technology challenges in achieving exascale systems. Technical report, DARPA, Arlington, USA.

Krasnov A, Schultz A, Wawrzynek J, Gibeling G and Droz PY (2007) Ramp blue: A message-passing manycore system in FPGAs. In: *17th international conference on field programmable logic and applications (FPL'07)*, Amsterdam, The Netherlands, 27–29 August 2007, pp. 54–61. Piscataway: IEEE Press.

Lee BC, Ipek E, Mutlu O and Burger D. (2009) Architecting phase change memory as a scalable DRAM alternative. *SIGARCH Computer Architecture News* 37(3): 2–13.

Li D, Vetter JS, Marin G, McCurdy C, Cira C, Liu Z, et al. (2012) Identifying opportunities for Byte-addressable non-volatile memory in extreme-scale scientific applications. In: *26th IEEE international parallel and distributed processing symposium (IPDPS'12)*, Shanghai, China, 21–25 May 2012, pp. 945–956. Piscataway: IEEE Press.

Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, et al. (2005) Pin: Building customized program analysis tools with dynamic instrumentation. In: *2005 ACM SIGPLAN conference on programming language design and implementation (PLDI'05)*, Chicago, USA, 11–15 June 2005, pp. 190–200. New York: ACM Press.

Mohiyuddin M, Murphy M, Oliker L, Shalf J, Wawrzynek J and Williams S (2009) A design methodology for domain-optimized power-efficient supercomputing. In: *2009 ACM/IEEE conference on high performance computing networking, storage and analysis (SC'09)*, Portland, USA, 14–20 November 2009, pp. 1–12. New York: ACM Press.

Narayanan SHK, Norris B and Hovland PD (2010) Generating performance bounds from source code. *39th international conference on parallel processing workshops (ICPPW'10)*, 10–13 September 2010, pp. 197–206. New York: ACM Press.

Qiu J and Shu C (2005) Runge–Kutta discontinuous Galerkin method using WENO limiters. *SIAM Journal on Scientific Computing* 26(3): 907–929.

Quinlan DJ, Miller B, Philip B and Schordan M. (2002) Treating a user-defined parallel library as a domain-specific language. In: *16th international parallel and distributed processing symposium (IPDPS'02)*, Fort Lauderdale, USA, 15–19 April 2002, pp. 105–114. Piscataway: IEEE Press.

Qureshi MK, Srinivasan V and Rivers JA. (2009) Scalable high performance main memory system using phase-change memory technology. *SIGARCH Computer Architecture News* 37(3): 24–33.

Rivera G and Tseng CW (2000) Tiling optimizations for 3D scientific computations. In: *2000 ACM/IEEE conference on supercomputing*, Dallas, USA, 4–10 November 2000. Piscataway: IEEE Press.

Rodrigues AF, Hemmert KS, Barrett BW, Kersey C, Oldfield R, Weston M, et al. (2011) The structural simulation toolkit. *SIGMETRICS Performance Evaluation Review* 38(4): 37–42.

Saulsbury A, Pong F and Nowatzyk A (1996) Missing the memory wall: The case for processor/memory integration. In: *23rd annual international symposium on computer architecture*, Philadelphia, USA, 22–24 May 1996, pp. 90–101. New York: ACM Press.

Shalf J, Dosanjh S and Morrison J (2010) Exascale computing technology challenges. In: Laginha JM, Palma M, Daydé M, Marques O and Correia Lopes J (eds) *High Performance Computing for Computational Science – VECPAR* 2010. New York: Springer, vol. 6449, pp. 1–25.

Shalf J, Quinlan D and Janssen C (2011) Rethinking hardware-software codesign for exascale systems. *IEEE Computer* 44(11): 22–30.

Snavely A, Carrington L, Wolter N, Labarta J, Badia R and Purkayastha A (2002) A framework for performance modeling and prediction. In: *2002 ACM/IEEE conference on supercomputing*, Baltimore, USA, 16–22 November 2002, pp. 1–17. Piscataway: IEEE Press.

Spafford KL and Vetter JS (2012) Aspen: a domain specific language for performance modeling. In: *2012 ACM/IEEE international conference on high performance computing, networking, storage and analysis (SC'12)*, Salt Lake City, USA, 10–16 November 2012, pp. 1–11. Piscataway: IEEE Press.

Thoziyoor S, Muralimanohar N, Ahn JH and Jouppi NP (2008) CACTI 5.1. Technical report no. HPL-2008-20, HP Labs.

Unat D, Shalf J, Hoefler T, Schulthess T, Dubey A (eds), et al. (2014) Programming abstractions for data locality. *Workshop on programming abstractions for data locality (PADAL'14)*, Lugano, Switzerland, 28–29 April 2014.

Vera X and Xue J (2002) Let's study whole-program cache behaviour analytically. In: *8th international symposium on high-performance computer architecture (HPCA'02)*, Cambridge, USA, 2–6 February 2002, pp. 175–186. Piscataway: IEEE Press.

Vladimirov A (2012) *Arithmetics on Intel's Sandy Bridge and Westmere CPUs: not all FLOPS are created equal.* Report, Colfax International.

Wawrzynek J, Patterson D, Oskin M, Lu SL, Kozyrakis C, Hoe JC, et al. (2007) RAMP: A research accelerator for multiple processors. *IEEE Micro* 27(2): 46–57.

Williams S, Waterman A and Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* 52(4): 65–76.

Woodward PR, Jayaraj J, Lin PH, Yew PC, Knox MR, Greensky JBSG, et al. (2010) Boosting the performance of computational fluid dynamics codes for interactive supercomputing. *Procedia Computer Science* 1(1): 2055–2064.

Author biographies

Dr Didem Unat has been a full-time faculty member at Koç University in Istanbul since September 2014. Previously she was at the Lawrence Berkeley National Laboratory (LBNL). She was the recipient of the *Luis Alvarez Fellowship* in 2012 from LBNL. Her research interests lie primarily in high-performance computing (HPC), parallel programming models, compiler analysis and performance modeling. She holds a PhD in Computer Science from the University of California, San Diego.

Dr Cy Chan is a research scientist in the Computer Architecture Group at LBNL, working on developing new techniques for software optimization and novel programming models for HPC systems. He holds an AB in Applied Mathematics from Harvard University and an SM and PhD in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology.

Dr Weiqun Zhang is member of the Center for Computational Sciences and Engineering at LBNL. His primary research focus is radiation hydrodynamics. Previously he has worked in a number of other subjects (e.g. relativistic hydrodynamics/magneto hydrodynamics and gamma-ray burst afterglows).

Dr Samuel Williams is a staff scientist in the Performance and Algorithms Research Group at LBNL. His research interests include HPC, auto-tuning, performance modeling, computer architecture, and hardware–software co-design. Dr Williams received his PhD in Computer Science from the University of California at Berkeley in 2008.

John Bachan is a computer systems engineer at LBNL in the Computer Architecture Group. His interests span programming languages (high-level functional and systems level), to software engineering methodologies. At present his research is based in instrumentation-driven simulation of hardware for exascale. This includes cache coherency in memory hierarchies and the performance of asynchronous task-based runtimes.

Dr John B Bell is a mathematician at LBNL. He has made contributions in the areas of finite difference methods, numerical methods for low-Mach-number flows, adaptive mesh refinement, interface tracking and parallel computing. He was elected to the National Academy of Sciences in 2012. He is also a Fellow of SIAM and was the recipient of the *Sidney Fernbach Award* from the IEEE in 2005.

John Shalf is the chief technology officer at the National Energy Research Scientific Computing Center. He is a member of the US Department of Energy Exascale Steering committee, and is a co-author of the landmark *View from Berkeley* paper as well as the DARPA Exascale Software Report. He currently leads projects in exascale technology research such as CoDEx (Co-Design for Exascale), and the LBNL Green Flash project that seeks to develop energy-efficient scientific computing systems using many-core and embedded technologies.