Software Design Space Exploration for Exascale Combustion Co-design*

Cy Chan, Didem Unat, Michael Lijewski, Weiqun Zhang, John Bell, and John Shalf

Lawrence Berkeley National Laboratory

Abstract. The design of hardware for next-generation exascale computing systems will require a deep understanding of how software optimizations impact hardware design trade-offs. In order to characterize how co-tuning hardware *and* software parameters affects the performance of combustion simulation codes, we created ExaSAT, a compiler-driven static analysis and performance modeling framework. Our framework can evaluate hundreds of hardware/software configurations in seconds, providing an essential speed advantage over simulators and dynamic analysis techniques during the co-design process. Our analytic performance model shows that advanced code transformations, such as cache blocking and loop fusion, can have a significant impact on choices for cache and memory architecture. Our modeling helped us identify tuned configurations that achieve a 90% reduction in memory traffic, which could significantly improve performance and reduce energy consumption. These techniques will also be useful for the development of advanced programming models and runtimes, which must reason about these optimizations to deliver better performance and energy efficiency.

1 Introduction

One of the challenges facing the scientific computing community is to ensure applications will perform well on future exascale machines years in advance of their arrival. Meeting the extreme power and performance challenges of HPC system design over the next decade requires a tightly coupled hardware/software co-design process that optimizes both the application *and* the hardware to meet target performance, power, and cost requirements [1]. Tuning software or hardware in isolation is insufficient to reach the optimal balance of these design goals. To this end, we require a capability to rapidly estimate the performance of scientific applications in various potential hardware and software configurations.

We present the *ExaSAT* (Exascale Static Analysis Tool) framework, which enables us to rapidly explore the effects of code optimizations on the performance of a target application in the context of varying hardware parameters.

* This manuscript has been authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. The U.S. Government retains, and the publisher, by accepting the article for publication, acknowledges, that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

J.M. Kunkel, T. Ludwig, and H. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 196–212, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Previous work includes cycle-accurate hardware simulators such as RAMP Gold [2] and discrete event simulators such as SST [3], which produce more accurate performance predictions than are feasible with static analysis but are more computationally expensive. Dynamic binary instrumentation tools such as Pin [4] can also be used to analyze the performance of a code by capturing events during code execution, but are subject to the quirks of the x86 ISA and compiler. In contrast, our framework provides a quantitative measure of application requirements through static code analysis, allowing us to characterize the co-design parameter space much more quickly than would be possible with simulators or dynamic analysis alone. Aspen [5] is a recent notation based language for analytical modeling where the programmer inserts a description of the application's performance behavior into the code. ExaSAT automatically generates a performance model directly from the source code without requiring programmer intervention, allowing us to analyze larger codes more easily.

We applied our framework to two combustion *proxy applications* (CNS and SMC) that were developed by the DOE Exascale Combustion Codesign Center (ExaCT) [6] to provide a representative set of core computational kernels required for combustion simulation. The majority of stencil computations at the heart of these codes are memory bandwidth bound on current architectures [7,8] and are predicted to become even more so on future architectures as computational throughput is expected to increase faster than memory bandwidth [9,10]. Furthermore, data movement is expected to become an increasingly important contributor to power consumption for exascale machines [11,12].

Because memory traffic is so critical, our analysis focuses on the effects of software optimizations that are intended to reduce data movement between the CPU and memory, rather than reducing the number of floating point operations. We examine optimal *cache blocking* (or *tiling*) and *loop fusion* code transformations and their effect on hardware design trade-offs as they relate to application performance for our combustion proxy applications. The software design space is parameterized to expose many of the potential realizations of the application and constituent kernels so that the best implementation can be selected. Applying our framework, we observe up to a 45% and 90% reduction in memory traffic when we apply optimal tiling and aggressive loop fusion, respectively.

Hardware complexity has increased to the point that current compilers are no longer able to automatically produce the code optimizations needed to achieve optimal performance on every target architecture. This paper demonstrates the impact of advanced code transformations that are beyond the capability of current compilers to produce and provides guidance for the development of new programming models and runtimes that will support these transformations. We discuss the following contributions in this work:

- We designed and implemented a fast, flexible static analysis and performance modeling framework and XML-based intermediate representation that can be used to estimate the performance of stencil computations and help explore trade-offs for co-design.

- We utilized our framework to profile the characteristics and estimate the performance of two combustion code variants: CNS and SMC, under a variety of hardware and software configurations.
- We used this model to illustrate the impact of cache blocking and loop fusion optimizations on hardware trade-offs, in particular how they affect cache size and memory bandwidth requirements.
- Our framework provides the deep code analysis and modeling necessary for future programming models and runtimes to reason about choices and make adaptations without a costly combinatorial search.
- Our analysis serves as a key vehicle for communicating with our industry partners for co-designing an exascale machine.

2 CNS and SMC Combustion Codes

To represent the characteristics of a range of combustion applications, we studied two proxy applications developed by the ExaCT project. The CNS code is a simple proxy that integrates the compressible Navier-Stokes equations assuming constant transport properties. It is intended to capture the computational characteristics of the dynamical core of a combustion simulation. The SMC code is a more advanced proxy for the direct numerical simulation combustion code S3D [13], adding detailed models for chemical species diffusion and kinetics. SMC contains the key elements of both the dynamical core and the chemical kinetics components of S3D; however, it uses a simpler temporal integration algorithm that does not include automatic error control. Both codes are based on the high-accuracy solution of a system of PDEs of the form:

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

Here U is a vector of unknowns, representing density, energy and three components of momentum with an additional density for each chemical species, for a total of $5 + N_s$ unknowns where N_s is the number of species in the problem (1 for CNS). The terms $\mathcal{F}$, $\mathcal{D}$, and S correspond to hyperbolic transport, nonlinear diffusive processes and chemical source terms, respectively. The dynamical core uses 8th-order stencil operations to approximate spatial derivatives, converting the system into a large collection of ordinary differential equations that are integrated using a third-order, low-storage, TVD Runge-Kutta scheme [14,15]. The chemical source term is a computationally intensive single-point physics routine that uses a large number of computationally expensive transcendental function evaluations. Further details on our approach will be presented in a forthcoming paper [16].

Figure 1 illustrates the most data-intensive stencil access pattern used in the CNS and SMC codes. This stencil reads values four grid elements deep in both directions of each of three dimensions; however, there are many other stencil patterns in the code that read only a subset of the points shown. Our tool separately analyzes each stencil access for every array in every loop to estimate working sets and data movement.

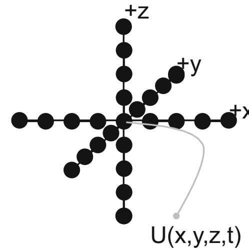

Fig. 1. 3D stencil access pattern in the SMC and CNS codes

Table 1 shows some computational properties of the codes studied. In addition to the ones shown, we also analyzed versions of the SMC code with 21, 71, and 107 chemical species, where the number of species varies with fuel type, from hydrogen to methane to biofuels. The CNS code (which simulates dynamics only) and the SMC dynamics codes have very different computational characteristics compared with the SMC chemistry codes. In our analysis, we utilize the *bytes per flop (B/F) ratio*, which represents the required number of bytes to be transferred between the processor and off-chip memory divided by the required number of floating point operations needed for a particular code. The division and transcendental operations are weighted since they cost more than adds and multiplies (see Section 4.2 for details).

An algorithm with a low B/F ratio will likely be computationally bound, while one with a high B/F ratio will likely be memory bandwidth bound. The CNS and SMC dynamics codes have a relatively high B/F ratio especially in a cache constrained environment (up to 2.22). They exhibit a high degree of data reuse both within and across loops, resulting in a lot of potential to reduce memory traffic using the optimizations discussed in this work. In contrast, the chemistry code is dominated by expensive floating point divisions and trancendental functions with a relatively light memory access requirement, resulting in a much lower B/F ratio. For the 53 species SMC code, the difference is roughly two orders of magnitude (0.01 vs. 1.48, cache-constrained). We expect the dynamics code to be bandwidth bound on most current and future architectures while the chemistry code will remain compute bound.

Although this paper focuses mainly on memory traffic optimizations that improve the performance of the dynamics codes, there are computational optimizations such as vectorization or pipelining that could help improve the throughput of the chemistry codes. Furthermore, the disparity in arithmetic intensity between the dynamics and chemistry codes suggests that co-scheduling could have each code utilize different parts of the processor simultaneously. Support for such optimizations within a programming model and runtime will be explored in future work.

Table 1. CNS and SMC code characteristics (for 1, 9, and 53 chemical species). RK = Runge-Kutta step. †Weighted flops. ‡Cache available to group of threads cooperating on a working set.

		CNS	SMC Dynamics		SMC Chemistry	
Num. 3D spatial loops		14	27		1	
Number of species		1	9	53	9	53
Flops/point per RK	Adds	821	7005	30677	619	7923
	Muls	797	7871	34860	815	9432
	Divs	6	66	374	39	540
	Trans	1	1	1	51	710
Arrays/RK	Resident	40	157	685	20	108
	Reads	92	557	2405	20	108
	Writes	40	253	1045	9	53
Bytes/Flop†	Unlimited $	1.32	0.82	0.74	0.03	0.01
	512 kB $‡	2.22	1.25	1.48	0.03	0.01

3 Software Design Space

3.1 Cache Blocking

The first optimization, cache blocking, focuses on reducing capacity misses (see [17] for more on 3C's cache model) as a core sweeps through the problem's iteration space. Tiling the iteration sweep reduces the size of the working set required to enable temporal reuse of data. If the working set is reduced to within the size of available on-chip memory, capacity misses can be reduced or eliminated, thus decreasing the necessary memory traffic between the CPU and DRAM.

The relationship between working set, cache size, and memory traffic can sometimes cause unexpected performance effects. For example, many programmers may parallelize a triply nested loop by associating an OpenMP `parallel for` pragma with the outermost loop (see Figure 2). This strategy yields the coarsest grain parallelism, which minimizes the overhead of spawning and syncing the resulting threads. However, parallelizing the middle loop instead reduces the working set of each thread by a factor of four (the number of threads). If the reduced working set now fits into the cache, then there may be a significant performance benefit. There is a trade-off in this scenario between cache size, memory bandwidth, and the costs of spawning and syncing threads.

Another trade-off for tiling is redundant *ghost zone* traffic that must be pulled in for each block. The ghost zone consists of neighboring cells outside of the tile that must be read due to the shape of the stencil access pattern (see Figure 1). The left diagram in Figure 3 shows a single, unblocked tile with ghost zones on the outside the tile. As the tile size is decreased (center and right diagrams), the ghost zones overlap with neighboring blocks (indicated with deeper shading).

Previous work has shown the benefits of cache blocking stencil codes [8,7]. In this paper, we are interested more in illustrating the co-design trade-offs that are exposed by software optimizations such as blocking, rather than the

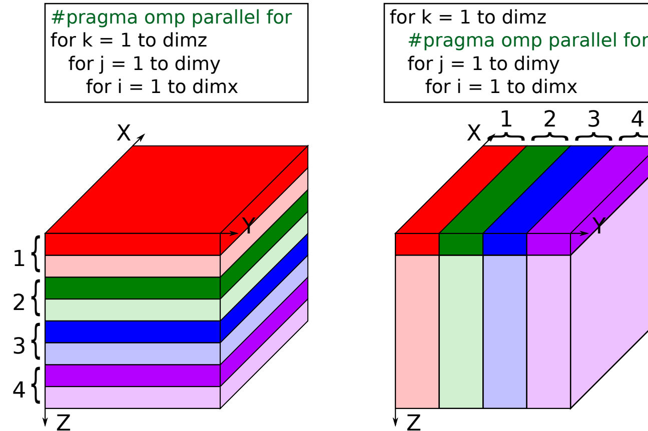

Fig. 2. Working sets that result from using OpenMP `parallel for` with the outermost vs. middle loop with four threads. Each color/number indicates the subgrid updated by a thread. Bold regions indicate the working set tile.

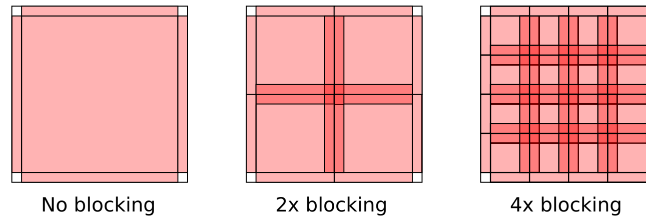

Fig. 3. 2D representation of the cache blocking optimization in the X and Y dimensions. Overlapping ghost zones are indicated by the deeper shading in the diagram.

raw performance benefits enabled. To this end, we developed a cache model to estimate the level of data reuse given different configurations. For any particular cache configuration, a blocking strategy may be chosen that balances the penalty of capacity misses against the overhead of redundant ghost-cell traffic. This optimization exposes a trade-off in hardware between cache size and memory bandwidth explored further in Section 5.

3.2 Loop Fusion

The second optimization, loop fusion, focuses on eliminating the need to stream arrays in and out of memory. While some compilers already implement fusion, they tend to do so to enhance instruction level parallelism and to help hide latency. In contrast, we apply loop fusion for the purpose of decreasing memory traffic by reducing the number of times arrays are transferred to or from memory [18].

Figure 4 shows an example of a loop fusion optimization. In Scenario 1, array A must be streamed from memory twice compared to just once for Scenario 2.

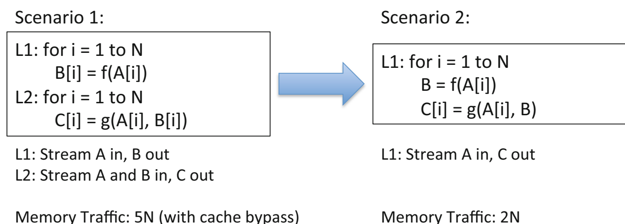

Fig. 4. Example of loop fusion code optimization

Also in Scenario 1, array B must be streamed to memory and back, while in the fused case the array can be replaced with a temporary variable (so long as B is not needed afterwards). Assuming cache bypassed writes, this optimization reduces memory traffic from $5N$ to $2N$, where N is the size of each array.

The trade-off for fusing loops is that the register and cache working sets grow, potentially causing a reduction in performance if the working sets no longer fit within on-chip memory. Loop fusion exposes a trade-off in the hardware involving the balance of memory bandwidth with registers and cache size. Our framework allows us to explore the impact of this transformation on memory traffic in the context of varying on-chip memory capacities. We will explore a couple strategies for applying loop fusion and their effects in Section 5.

4 Methodology

4.1 Framework and Toolchain

We developed a stencil-specific static analysis and performance modeling tool to help estimate the performance of target codes on various potential hardware platforms. Figure 5 illustrates our framework, which consists of roughly two stages of analysis. The first stage, which is built on top of the ROSE compiler [19], takes Fortran code as an input and extracts key characteristics about the computation and data access patterns and stores them in an XML intermediate representation (XML-IR). Data in the XML-IR include (but are not limited to) the following:

- Loop nest structure, bounds, and strides
- Floating point operations
- Scalar accesses (number of reads and writes)
- Array accesses (number of reads and writes for each index)

The second stage (written in Python) combines the XML-IR with user-provided problem parameters (e.g. box size, number of chemical species), machine parameters (e.g. computational throughput, cache size, memory bandwidth), and software optimizations (e.g. loop transformations) to produce estimates of key performance metrics such as working set sizes, DRAM traffic, B/F ratio, and execution time. The resulting performance model can be executed within a script

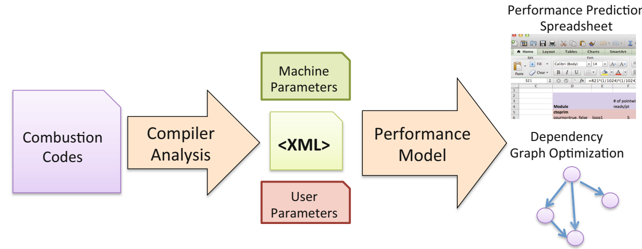

Fig. 5. ExaSAT Tool Chain

to rapidly explore parameter configurations, and it can additionally produce spreadsheets, dependency graphs, and tables with additional details such as array residency and access frequency, memory footprints, and state variable statistics. When applied to the SMC proxy application, our framework can evaluate roughly 900 hardware/software configurations per minute on a laptop. More details will be provided in future work [20].

4.2 Hardware and Performance Model

We utilize a simple hardware model (shown in Figure 6) that abstracts the machine as a collection of parallel hardware cores with some parameterized on-chip memory. Our hardware model exposes the following architectural parameters, which were identified through discussion with industry participants in the DOE Fast Forward program:

- Aggregate computational throughput
- Aggregate memory bandwidth
- Cache or scratchpad size
- Cache line and word sizes
- Cost of special functions (e.g. divisions or transcendentals)

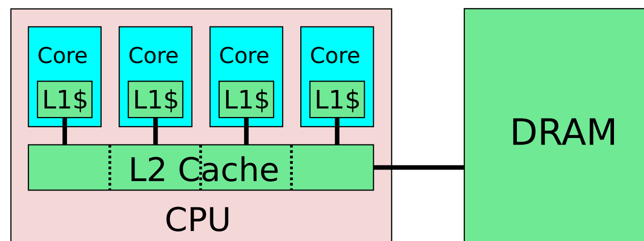

Fig. 6. Hardware model featuring the CPU with two levels of on-chip cache and separate DRAM connected by a bus

The CPU model is agnostic to the number of cores, instead taking the aggregate computational throughput as a model parameter. The memory model

similarly utilizes a parameter specifying the aggregate bandwidth between the CPU and the DRAM (i.e. the STREAM bandwidth) over the memory bus. Figure 6 shows an example cache configuration with private L1 caches and a partitioned, shared L2 cache. While there are many other possible on-chip memory configurations, we are mainly interested in the resulting bandwidth filtering capability, which is primarily determined by the total amount of (non-inclusive) on-chip memory per thread or group of cooperating threads. The performance model focuses on aggregate performance of the machine rather than simulating individual components and their interactions, It captures the costs of the computational workload and data movement and the performance implications of data reuse (or lack thereof).

The performance of an application is estimated in the following way: let α be the aggregate computational throughput of the machine and β be the aggregate memory bandwidth. Also, let C be the application's total computational work and D be the total necessary data movement between the CPU's on-chip memory and DRAM. Then the estimated running time is $T = \max(T_c, T_d)$, where $T_c = \frac{C}{\alpha}$ is the CPU time and $T_d = \frac{D}{\beta}$ is the DRAM time. Our modeling framework is not intended to provide exact performance predictions, but rather sets a performance upper-bound in the spirit of the Roofline model [21].

Since some floating point operations such as divides and transcendentals can take significantly longer to execute than adds and multiplies, our performance model can weight these special operations according to their relative costs. For this paper, we weighted the costs of these operations according to their non-SIMD throughput on the Intel Sandy Bridge architecture [22,23]. The resulting *weighted flop* count determines the estimated CPU time of the computation. Figures 7(a) and 7(b) show the significance of weighting the floating point operations by their cost. In the chemistry module of the SMC proxy application, the CPU time is dominated by transcendentals, even though the transcendental operation count is a small percentage of the total flops.

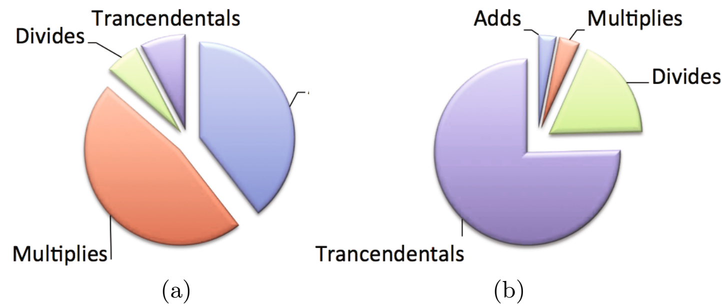

Fig. 7. (a) Floating point instruction mix and (b) CPU Time (T_c) for the chemistry part of the SMC code for 53 chemical species, assuming divides and trancendentals take a relative factor of 39x and 125x longer than the adds and multiplies, respectively

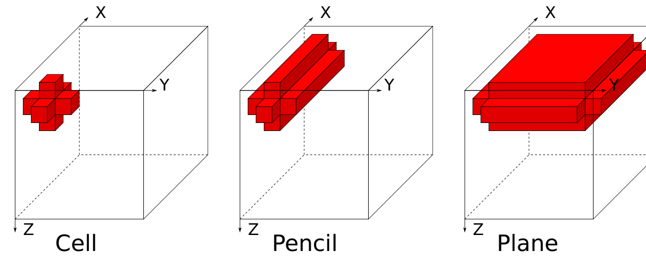

Fig. 8. Working set sizes to enable different levels of reuse for an example 7-point 3D stencil. The block is swept in a triply nested loop with the X dimension swept in the innermost loop and Z in the outermost.

Memory and Cache Model. In order to determine D, the total data movement, our cache model captures the data reuse pattern that occurs with stenciled array accesses. The on-chip memory is modeled as an ideal, fully-associative cache with a least-recently used (LRU) replacement policy. Our model assumes reuse of data will occur if the associated working set is small enough to fit in on-chip memory. Actual cache behavior is likely to under-perform in comparison due to conflict misses and imperfect replacement, though our model should capture the first-order behavior of a finite-cache memory system. Threads on a chip may cooperate to gain the benefit of a larger aggregate memory space in which to store a shared working set, possibly enabling larger block sizes and reduced memory traffic; however, the costs of sharing data between the caches on the chip are not included in our model. The amount of memory traffic required to execute a particular stencil loop is determined by the amount of the on-chip memory available per group of threads collaborating on a working set. Our model uses the specified cache size to 1) determine what temporal reuse of data will occur as threads sweep through the grid and 2) estimate the resulting cache miss traffic.

Figure 8 shows the working set sizes needed to enable reuse between cells, pencils, and planes. If no on-chip memory is available, every array access in the kernel requires data to be transferred from DRAM. If the cell working set (left) fits in cache, then those values will remain in cache for reuse on the next cell iteration. Similarly, if the pencil working set (middle) fits into cache, there will be reuse between pencil sweeps, and so forth. Based on the shape of the stencil access pattern, our model computes 1) the sizes of the working sets and 2) the resulting memory traffic for each of the reuse cases. This information is then combined with hardware and software parameters to determine the estimated memory traffic and DRAM time required for each array in every loop in the code.

If on-chip data movement is a concern, a conservative estimate can be made by limiting the size of modeled on-chip memory to the size of the private L1 cache; however, the resulting memory traffic estimates produced by our model will be the total traffic between the L1 cache and the next level of on-chip memory rather than the traffic between the CPU and DRAM. Our methodology could

potentially be extended in future work to do a multi-level analysis that computes bandwidth filtering and performance modeling at every level of cache.

Model Validation. Figure 9 shows the effect of tiling with three simple 7-point stencil benchmark kernels on a single node of the NERSC "Hopper" Cray XE6 using 384^3 data grids. Table 2 shows properties of the benchmarked machine. We measured the execution time for 24 concurrent threads (1 thread per core) with no software prefetch or cache bypass used. To first-order approximation, the measured execution times correlate well with our model's predictions with respect to optimal execution times and block sizes. The model departs from measurement for smaller blocks as the hardware prefetchers are no longer able to hide load latencies for short stanza accesses. We also observe the effect of the machine's randomized cache replacement policy, which smooths the sharp transition in the model at the point where the working set grows larger than the cache and capacity misses begin to occur. Since static analysis does not resolve system behavior to the same degree of precision as an event simulator in the interest of speed and flexibility, our results are necessarily more comparative than absolute in nature. That said, we believe valuable lessons can be learned from examining the trade-offs exposed by our analysis framework in the co-design parameter space.

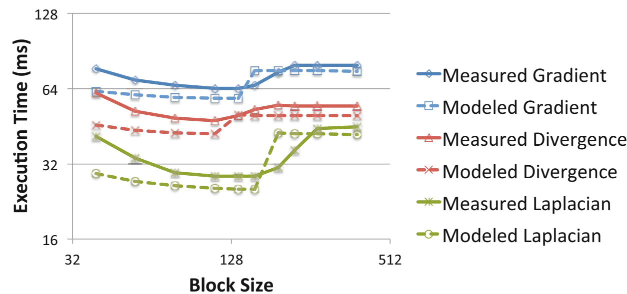

Fig. 9. Measured and modeled execution times for various blocking sizes for three benchmark finite difference kernels

5 Results

Since the optimizations studied do not change the amount of computation (flops) required, the B/F ratio can be used as a proxy metric for memory traffic (and thus execution time) in memory bandwidth bound codes. In this context, the B/F ratio is an indicator of relative code performance independent of a particular machine's specifications, and is useful when making comparisons between code requirements and machine capabilities during the design process.

Table 2. Properties of a NERSC Cray XE6 compute node [24]. All numbers given are *per node* except cache, which are given *per core*. †Data cache.

One NERSC "Hopper" Node	
CPUs	Opteron 6172
Sockets / Cores	2 / 24
Peak Compute	201.6 Gflop/s
Priv. L1/L2 $	64 kB† / 512 kB
Shared L3 $	1 MB / core
Mem. Interface	DDR3-1333
Mem. Channels	8
Peak Mem. BW	72 GiB/s
STREAM BW	∼51 GiB/s
Peak B/F Ratio	0.38

5.1 Cache Blocking

Figure 10 shows the B/F ratio for the CNS proxy application for different block sizes and on-chip cache sizes. In many cases there is insufficient cache to enable the best reuse case outlined in Section 4.2. Blocking the iteration space reduces the sizes of the working sets needed to enable reuse, but incurs the overhead of pulling in additional ghost zones for the smaller blocks. This overhead is illustrated by the unlimited cache case, where the B/F ratio increases as the block size decreases. For a fixed block size, as the amount of cache is reduced, more capacity misses occur, increasing the B/F ratio. For a fixed cache size, we observe the minimum B/F ratio typically occurs at the largest block size whose working sets still fit within cache. The multiple inflection points are due to the code's various loops having different working set sizes and reuse behaviors.

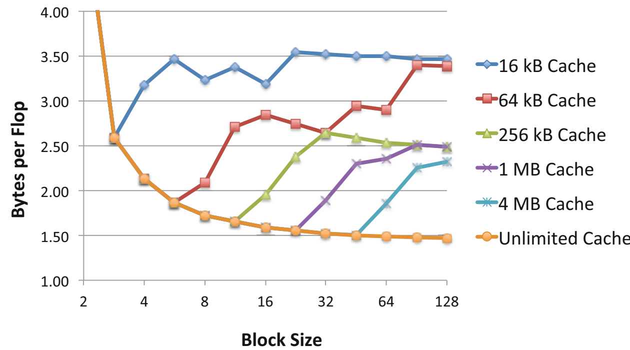

Fig. 10. Byte to flop ratio for various block and cache sizes

Figure 11 shows that as chemical species are added to the simulation, the memory traffic required per Runge-Kutta step increases across all block sizes.

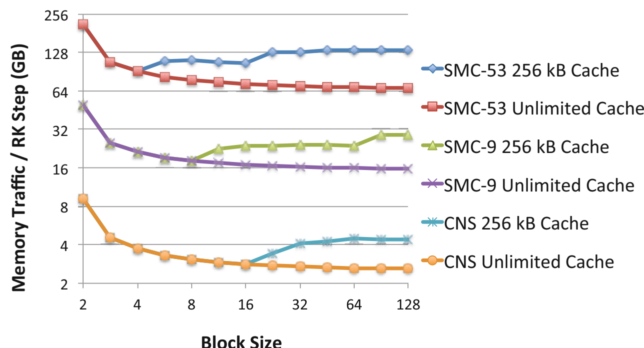

Fig. 11. Memory traffic per Runge-Kutta step as block size, cache size, and number of chemical species are varied

Since the working set size increases with number of species, the optimal block size for a fixed amount of cache decreases. In cases with a large number of chemical species, augmenting the cache resources on the chip would reduce the blocking overhead and ease memory bandwidth requirements.

5.2 Loop Fusion

We examined two variants of the loop fusion optimization: simple and aggressive. We use the term *stencil dependency* to refer to a data dependency between loops where data written by the first loop is read by the second in a stencil (non-pointwise) access pattern. In the simple fusion case, only loops with no stencil dependencies are fused, while in the aggressive fusion case, all loops in the solver are fused. Simple fusion can be applied without major changes to the loop bodies, but aggressive fusion requires the introduction of temporary buffers and a staggered update strategy to replace arrays with stencil dependencies. While our framework is able to model the effects of cache blocking without any manual code modification, the loop fusion transformations studied here were implemented manually using the dependency graphs generated by our tool for guidance. Analysis of the resulting fused code was then handled by our framework.

Figure 12 shows the dependency graphs generated by our framework (simplified for clarity) corresponding to the different fusion cases for the CNS code. The ovals correspond to loops in the solver, while rectangles represent data arrays. The arrows show which arrays are read and written by each loop (dashed arrows represent stencil dependencies).

Figure 13 shows the bandwidth filtering that results for the CNS code using various cache sizes and loop fusion strategies. For each point in the graph, the cache blocking strategy was independently chosen to minimize the resulting memory traffic in our model. The stair-step pattern observed with the simple fusion scenario is a result of the transition between cell, pencil, and plane reuse

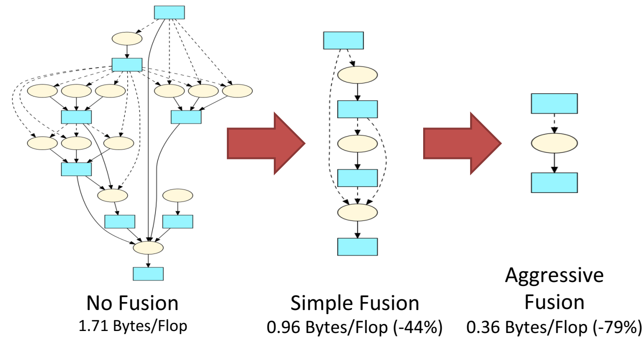

Fig. 12. Dependency graph showing loop fusion optimizations

cases as explained in Section 4.2. As expected, increasing the cache size introduces the opportunity to substantially reduce memory bandwidth requirements.

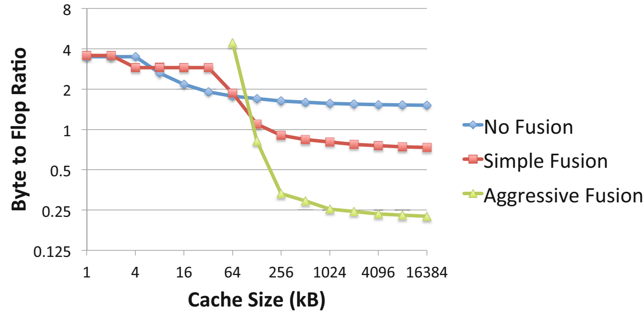

Fig. 13. Bandwidth filtering diagram for the CNS code for different loop fusion strategies

For small caches, there is typically only modest benefit from using loop fusion because the larger fused working sets require smaller block sizes to fit into cache. However, once the cache is large enough to fit the fused working sets, the benefits can be dramatic. For the largest cache, a 6.7x reduction in B/F ratio can be attained for the CNS code using the aggressive fusion strategy, but even with only 128 kB of cache the traffic is reduced by 2.1x compared to the unfused strategy.

5.3 Analysis

Given the size of the on-chip memory in our hardware configuration, we can choose the best overall optimization strategy to minimize memory traffic. As a result of applying the best combination of blocking and fusion strategies, the realizable bandwidth filtering curve is the minimum across the curves shown in Figure 13.

In some cases, making the trade-off of dedicating extra die area for cache can lead to a substantial benefit in power and performance from reduced memory

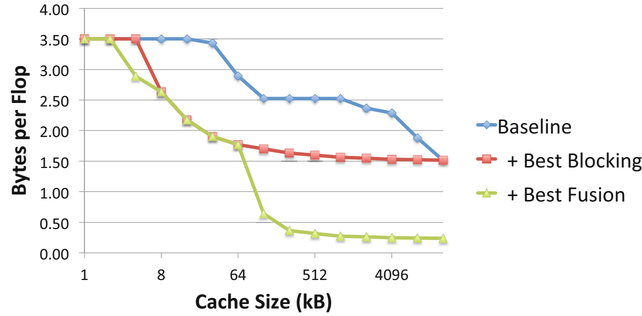

Fig. 14. Impact of software optimizations on the trade-off between cache size and memory bandwidth for the CNS code

traffic and lower bandwidth requirements. The effect on this trade-off due to software optimization is illustrated in Figure 14 for the CNS code. Tiling provides up to a 45% improvement versus the baseline unoptimized code at cache capacities larger than 8 kB. Loop fusion has the potential to filter bandwidth by as much as 90% compared to baseline, but it requires a cache larger than 64 kB. Similarly, Figure 15 shows the impact of the code optimizations on the 53 species SMC dynamics code. The stair-step pattern resulting from the transition between reuse cases is more prominent here due to the larger working sets. The SMC code has a lower baseline B/F ratio compared with CNS due to the arithmetic complexity of the code, but the minimum cache size for blocking improvements is higher than CNS due to the increased amount of data handled. Furthermore, several loops in the SMC code can be fused without large working set penalties, improving performance even with small cache sizes. Tiling provides up to a 39% improvement versus the baseline unoptimized code, while fusion can reduce traffic by up to 60% versus baseline.

The lowered bandwidth requirements due to these optimizations could have a significant beneficial impact on energy efficiency of future systems. However,

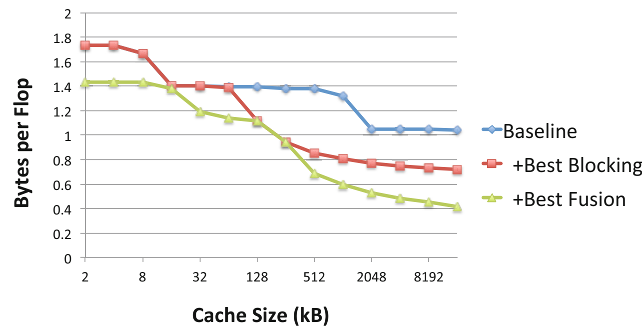

Fig. 15. Impact of software optimizations on the trade-off between cache size and memory bandwidth for the 53 species SMC dynamics code

the larger cache sizes required could be difficult to implement without moving towards a scratchpad memory implementation, such as the one used in the STI Cell processor. Our results show that it is essential to consider both software optimizations and hardware design parameters simultaneously. These observations would not have been apparent from benchmarking on fixed hardware alone.

6 Conclusions and Future Work

We developed a compiler-based framework that is able to automatically construct performance models directly from source code and applied it to explore trade-offs in the hardware design space using the CNS and SMC combustion proxy applications. Using this approach, we demonstrate tuned hardware/software configurations that achieve up to 45% and 90% reductions in compulsory memory traffic with the application of optimal data tiling and aggressive loop fusion, respectively. We believe this kind of deep code analysis and performance modeling demonstrates the importance for future advanced runtimes to make dynamic adaptations in the context of changing computing environments without a costly combinatorial search. Our analysis serves as a key vehicle for communicating with our vendor partners for co-designing an exascale machine.

We wish to generalize our approach and make it practical to apply these techniques to larger, more complex codes. Because the optimizations studied here require significant code transformations, current compilers are unable to perform them automatically. We are using the lessons learned here to guide the development of programming models and frameworks that will enable the automation of our code transformation and performance analysis techniques. For example, we are exploring the use of functional semantics and annotations to help reason about data flows and on-chip memory footprints. In summary, our work demonstrates the utility of a co-design approach, which explores the design space of software optimizations with parameterized hardware and offers deeper insight into the future of application and machine design.

Acknowledgements. The authors would like to thank George Michelogiannakis and Sam Williams for their insightful comments during the preparation of this paper. All authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231. This work is part of the DOE Center for Exascale Simulation of Combustion in Turbulence (ExaCT) and the DOE Co-Design for Exascale (CoDEx) projects.

References

1. Mohiyuddin, M., et al.: A design methodology for domain-optimized power-efficient supercomputing. In: SC 2009, pp. 12:1–12:12. ACM, New York (2009)
2. Tan, Z., et al.: RAMP Gold: An FPGA-based architecture simulator for multiprocessors. In: 2010 47th ACM/IEEE Design Automation Conference (DAC), DAC 2010, pp. 463–468 (June 2010)

3. Janssen, C.L., et al.: A simulator for large-scale parallel computer architectures. International Journal of Distributed Systems and Technologies 1(2), 57–73 (2010)
4. Luk, C.-K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI 2005, pp. 190–200. ACM, New York (2005)
5. Spafford, K.L., Vetter, J.S.: Aspen: a domain specific language for performance modeling. In: SC 2012, pp. 84:1–84:11. IEEE Computer Society Press, Los Alamitos (2012)
6. ExaCT: Center for exascale simulation of combustion in turbulence. Website (2013), http://exactcodesign.org
7. Datta, K., et al.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: SC 2008, pp. 4:1–4:12. IEEE Press, Piscataway (2008)
8. Rivera, G., Tseng, C.-W.: Tiling optimizations for 3d scientific computations. In: Supercomputing 2000. IEEE Computer Society, Washington, DC (2000)
9. Kogge, P., et al.: Exascale computing study: Technology challenges in achieving exascale systems (2008)
10. Shalf, J., Dosanjh, S., Morrison, J.: Exascale computing technology challenges. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds.) VECPAR 2010. LNCS, vol. 6449, pp. 1–25. Springer, Heidelberg (2011)
11. Miller, D.A.B.: Rationale and challenges for optical interconnects to electronic chips. In: Proc. IEEE, pp. 728–749 (2000)
12. Borkar, S.: Design challenges of technology scaling. IEEE Micro 19(4), 23–29 (1999)
13. Chen, J.H., et al.: Terascale Direct Numerical Simulations of Turbulent Combustion Using S3D. Comput. Sci. Disc. 2(015001) (2009)
14. Gottleib, S., Shu, C.: Total variation diminishing Runge-Kutta schemes. Mathematics of Computation 67(221), 73–85 (1998)
15. Qiu, J., Shu, C.: Runge-Kutta discontinuous Galerkin method using WENO limiters. SIAM J. Sci. Comp. 26(3), 907–929 (2005)
16. Zhang, W., et al.: Multirate higher-order discretization approaches for the multicomponent, reaction compressible Navier-Stokes equations (in preparation)
17. Hill, M.D., Smith, A.J.: Evaluating associativity in cpu caches. IEEE Trans. Comput. 38(12), 1612–1630 (1989)
18. Ding, C., Kennedy, K.: Improving effective bandwidth through compiler enhancement of global cache reuse. J. Parallel Distrib. Comput. 64(1), 108–134 (2004)
19. Quinlan, D.J., Miller, B., Philip, B., Schordan, M.: Treating a user-defined parallel library as a domain-specific language. In: IPDPS 2002, p. 324. IEEE Computer Society (2002)
20. Unat, D., Chan, C., et al.: Exasat: A static analysis and performance modeling tool for exascale co-design (in preparation)
21. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commununications of the ACM 52(4), 65–76 (2009)
22. Williams, S.: Intel Sandy Bridge SVML benchmark results (2012)
23. Vladimirov, A.: Arithmetics on Intel's Sandy Bridge and Westmere CPUs: not all FLOPS are created equal. Colfax International (2012)
24. NERSC: Cray XE6 (Hopper). Website (2013), http://www.nersc.gov/users/computational-systems/hopper