DOSA: Design Optimizer for Scientific Applications *

David A. Bader[1] and Viktor K. Prasanna[2]

[1]College of Computing
Georgia Institute of Technology
Altanta, GA 30332
bader@cc.gatech.edu

[2]Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089
prasanna@ganges.usc.edu

Abstract

In this work, we propose an application composition system (ACS) that allows design-time exploration and automatic run-time optimizations so that we relieve application programmers and compiler writers from the challenging task of optimizing the computation in order to achieve high performance. Our new framework, called "Design Optimizer for Scientific Applications" (DOSA), allows the programmer or compiler writer to explore alternative designs and optimize for speed (or power) at design-time and use its run-time optimizer as an automatic ACS. The ACS constructs an efficient application that dynamically adapts to changes in the underlying execution environment based on the kernel model, architecture, system features, available resources, and performance feedback. The run-time system is a portable interface that enables dynamic application optimization by interfacing with the output of DOSA. It thus provides an application composition system that determines suitable components and performs continuous performance optimizations. We focus on utilizing advanced architectural features and memory-centric optimizations that reduce the I/O complexity, cache pollution, and processor-memory traffic, in order to achieve high performance. The design-time effort uses a computer-aided design space exploration that provides a user-friendly graphical modeling environment, high-level performance estimation and profiling, and the ability to integrate low-level simulators suitable for HPC architectures.

1. Introduction

High-performance computing (HPC) systems are taking a revolutionary step forward with complex architectural designs that require application programmers and compiler writers to perform the challenging task of optimizing the computation in order to achieve high performance. In the past decade, caches and high-speed networks have become ubiquitous features of HPC systems, canonically represented as clusters of multiprocessor workstations. To efficiently run application codes on these systems, the user must carefully lay out data and partition work so as to reduce communication, maintain a load balance, and expose locality for better cache performance. Realizing the gap between processor and memory performance, several HPC vendors, such as IBM and Cray, are incorporating into their next-generation systems innovative architectural features that alleviate this *memory wall*. These new architectural features include hardware accelerators (e.g., reconfigurable logic such as FPGAs, SIMD/vector processing units such as in the IBM Cell Broadband Engine processor, and graphics processing units (GPUs)), adaptable general-purpose processors, run-time performance advisors, capabilities for processing in the memory subsystem, and power optimizations. With these innovations, the multidimensional design space for optimizing applications is huge. Software must be sensitive to data layout, cache parameters, and data reuse, as well as dynamically changing resources, for highest performance.

Until recently, design-time analysis and optimizing compilers were sufficient to achieve high-performance on many HPC systems because they often appeared as static and homogeneous resources with a simple, well-understood model of execution at each processor. Today, techniques for load-balancing and job migration, readily-accessible grid computing, complex reconfigurable architectures, and adaptive processors, necessitate the requirement for run-time optimizations that depend upon the dynamic nature of the computation and resources. At run-time, an application may have different node architectures available to its running processes because it is executing in a distributed grid environment, and each component may require its own spe-

*This work is supported in part by the National Science Foundation, USA, under grant numbers CNS-0614915 and CNS-0613376.

cific optimization to make use of the unique features available to it. HPC systems may have adaptable resources such as processors, and the run-time system gains new responsibility for requesting the appropriate configuration for the workload. Also, in a large, complex computing system, the available resources may change during iterations, and the run-time system must monitor, select, and tune new components to maintain or increase performance. Our goal is to design a dynamic application composition system that provides both high-performance computing and increased productivity. In this work, we discuss the Design Optimizer for Scientific Applications (DOSA), a semi-automatic framework for software optimization. DOSA will allow rapid, high-level performance estimation and detailed low-level simulation.

2. DOSA Framework

2.1. Framework Overview

In this paper we use the following naming conventions. In the framework, applications will be represented in terms of the kernels of which they are comprised. Kernels, in turn, are made up of tasks and are represented as task flow graphs. Tasks correspond to the components in the component library. For example, molecular dynamics (MD) simulation is an application. One kernel of MD simulation is non-bonded force calculation. One task of non-bonded force calculation is the transmission of the molecule-position information to the appropriate processors. The component library would provide various components for carrying out this task, such as one component with which the transmission would be carried out through traditional message passing and another component with which the transmission would be carried out through cache injection. Throughout the paper, we use the term "performance models" to describe the models that are used within and produced by our DOSA framework. When describing theoretical models, such as the I/O complexity model, we use the term "model."

The design framework (DOSA) allows various optimizations to be explored based on the architectural features of the HPC platforms. In the DOSA framework, at design time, the designer models the kernels and architectures of her/his application. This modeling process includes performance estimation, simulation, identification of bottlenecks, and optimization. Through this modeling process, the designer identifies sets of components in the component library that can be utilized for the tasks in the kernels of her/his application. We will implement a high-level estimator that will estimate bandwidth utilization, memory activity, I/O complexity, and performance. Our approach is hierarchical: the framework will be used to perform a coarse exploration to identify potential optimizations, followed by detailed simulations to validate the performance improvements of the optimizations. The detailed simulations can also expose additional optimizations. This design-time exploration is manual. Thus our framework uses a semi-automatic design space exploration tool. The framework outputs a representation of the kernel, the performance models, and a run-time optimizer that can use these for efficient execution of the kernel (see Fig. 1).

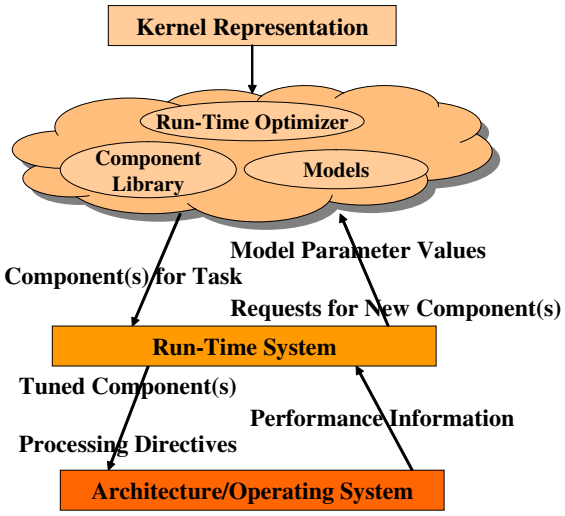

Figure 1. The run-time optimizer and run-time system compose applications dynamically based on system performance and available resources.

The run-time system is a portable, high-level advisor that interfaces between the underlying execution system (operating system and architecture) and the application. During a kernel's execution, the execution system supplies performance information that is used by the run-time system to determine if the run-time optimizer should be called and to update the values of the parameters in the performance models. The run-time optimizer uses the updated performance models to select appropriate component(s) from the set determined during design-time optimization for executing the current task. Together, the run-time system and run-time optimizer make the run-time optimization decisions as a dynamic application composition system.

2.2. Framework Implementation

The Design Optimizer for Scientific Applications (DOSA) framework will provide a user-friendly graphical modeling environment for design-time optimization and the identification of components for run-time optimizations. To this end, the framework will enable high-level perfor-

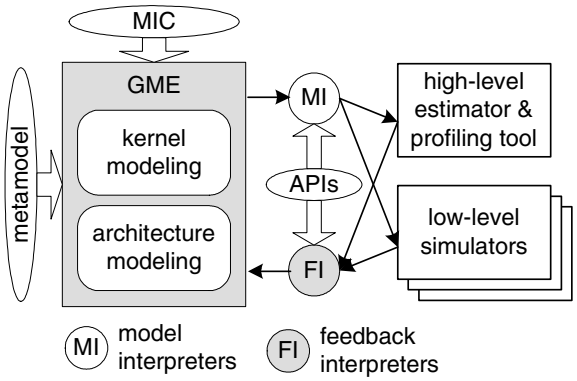

Figure 2. Design of the DOSA framework

mance estimation and profiling, integrate low-level simulators, and facilitate semi-automatic design space exploration. For each kernel, there are three outputs from the design process: a representation of the kernel, the performance models that describe the kernel's performance given under various conditions, and a run-time optimizer that handles the selection of components at run time.

We will use the Model Integrated Computing (MIC) methodology to create the design framework [11, 14]. The MIC approach advocates the extension of the scope and usage of models such that they form the "backbone" of a model-integrated system design process [7, 8]. In this design process, a model captures the information relevant to the system to be developed, represents the designer's understanding of the entire system, and specifies dependencies and constraints among the different components of the system.

The Generic Modeling Environment (GME) is a configurable graphical tool suite supporting MIC [5, 9]. GME allows the designer to create domain-specific models [14]. A metamodel (modeling paradigm) is a formal description of model construction semantics. Once the metamodel is specified by the user, it can be used to configure GME itself to present a modeling environment specific to the problem domain. MIC enables design reuse through the models. For example, several kernels can be optimized for the same advanced architecture, each using the same model of the system. We will use the model database supported by GME to save and reuse models.

Model interpreters are software components that translate the information captured in the models to drive integrated tools that estimate the performance (latency, energy, throughput, etc.) of a system. Model interpreters can also be configured to automatically translate the models into executable specifications. Feedback interpreters are software modules that analyze the output generated by the integrated tools and update the models. These interpreters are essentially automation tools that, once written, can be used for several system design problems. GME supports a set of well-defined C++ APIs that provide bidirectional access to the models [5]. Both model and feedback interpreters are developed using these APIs.

2.3. Design Flow for Design-Time Optimization

The design flow consists of two phases: configuration and exploration. In the configuration phase, the framework is configured based on the target kernel and the architecture. The kernel developer (designer) initially defines a structural model of the target architecture. The structural model captures the details of the architecture. The designer uses appropriate micro-benchmarks and model and feedback interpreters to automatically perform cycle-accurate architecture simulation using integrated simulators (such as Mambo or SimpleScalar) to estimate the values of these parameters and update the model. We also plan to create a library of architecture models because it is likely that a designer would optimize several kernels for the same target architecture. If architecture models are available for the target architecture, then the designer can choose to skip the configuration phase and use the library instead. Along with the architecture, the designer also models the kernel. Kernel modeling involves describing the kernel as a task graph and specifying the components that implement these tasks. We will create a library of common components, such as matrix multiplication, FFT, and graph operators [3, 4]. If the desired component is not in the library, the designer can develop it from within the framework. We will write model interpreters that will generate executable specifications from a task graph. Once a kernel is modeled, the designer uses the integrated simulators to automatically generate a memory access trace for each code segment and associate it with the model.

In the exploration phase, the performance models (architecture and kernel models) defined in the configuration phase are used to perform design space exploration. Initially, the designer uses the high-level performance estimator to rapidly estimate the performance of the design and generate a profile of memory access. Based on the estimate and the profile, the designer identifies appropriate optimizations such as I/O complexity optimizations, data layout optimizations and data remapping optimizations, and modifies the models to include components supporting these types of optimizations. For example, in the case of in-memory processing for data remapping, the kernel model will be modified to indicate that a component for this type of processing is available and the estimator will account for concurrent in-memory processing during performance estimation and also suitably modify the memory access profile to reflect the remapping. The designer also can perform automatic

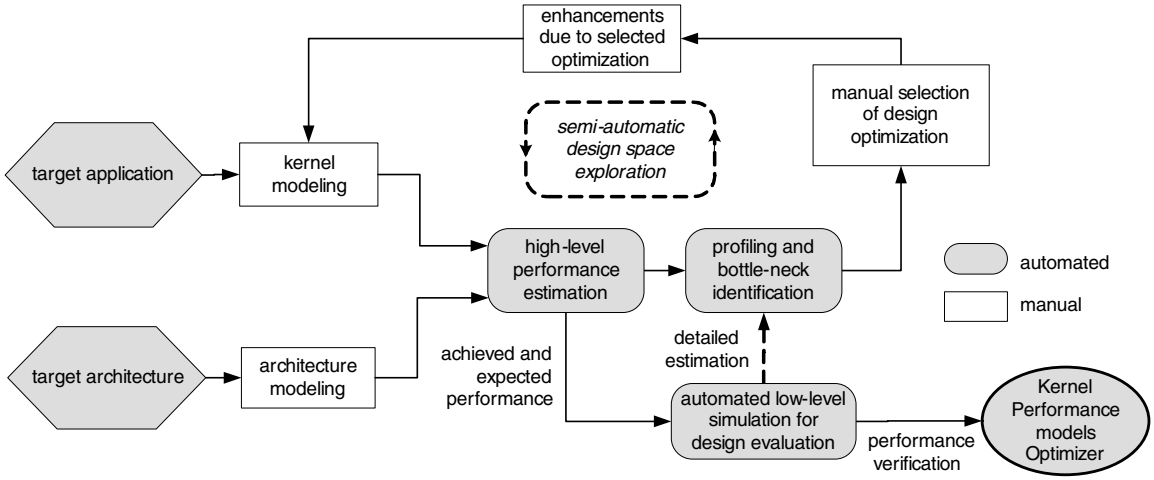

Figure 3. Design flow using the DOSA framework

low-level simulation to verify design decisions. Note that the low-level simulation is optional. By using the high-level models and the estimations, the design time can be reduced significantly. However, low-level simulations can provide additional insights with respect to optimizations to be considered. The designer continues to perform estimation, optimization, and low-level simulation until desired performance is reached or optimal processor-memory traffic is achieved. Similarly, for memory energy optimizations, the designer can identify data placement schemes (blocking, tiling, etc.) and memory/bank activation schedules and use the performance estimator to evaluate reduction in energy dissipation. Simplifying assumptions made to enable rapid estimation may induce errors in the estimates. Therefore, the DOSA framework will support specification of multiple candidate designs with estimated performance close to the desired performance. The framework will then output a representation of the kernel, the performance models, and a run-time optimizer that will use the representation and the performance models for efficient execution of the kernel.

2.4. Run-Time System Implementation

The run-time system automates the component selection process rather than requiring manual intervention from the user and hence, increases the productivity for efficiently using available resources. The design-time effort produces a set of components from the component library for the tasks in each kernel. These components are targeted towards different optimization points and architectures. Together with the run-time optimizer output from the DOSA framework, the run-time system provides a dynamic application composition system that selects from these components using the performance models. The run-time optimizer is hooked into the computation when the kernel executes. The first time the kernel executes, the run-time optimizer makes an initial selection from the optimal designs using the design-time analysis, performance models, execution-system information, and knowledge obtained from prior runs. For each task in the kernel, the run-time optimizer selects the component(s) from the component library, requests any system configurations and adaptations required by the component(s), and sets up its baseline for tracking the component's performance. At this point, the component selection is made, and the run-time optimizer calls the component. When the component returns, the run-time system logs its resource usage. On the next call for this kernel, the run-time system evaluates whether the system performed as the performance models suggest (or within the power budget), and takes corrective action, if needed. For instance, it may tune cache parameters within the component, select a different component (perhaps from other leading components for the task), and/or request a processor reconfiguration or system adaptation. Once the performance is within tolerance, the run-time system may back-off from its managerial role, only checking occasionally that the achieved kernel performance is still within an acceptable range. When an application finishes its execution, the run-time forensics and learned configurations, performance, and behavior, are incorporated as knowledge into the component library so that future runs may benefit from these optimizations.

2.5 Performance Optimizations

We divide the optimizations facilitated by the DOSA framework into two classes: design-time optimizations and run-time optimizations.

Design-Time Optimizations

1. **Optimizations for Latency.**

(a) **I/O complexity optimizations**. We will extend the standard, theoretical I/O complexity model to incorporate multi-level memory hierarchies of the proposed HPC architectures. This will allow us to derive communication complexity bounds and exploit these bounds in software development using the framework.

(b) **Data layout optimizations**. In many HPC applications, sophisticated data layouts in memory can significantly improve performance over naïve data layouts. For instance, techniques such as blocking, tiling, and linked-list to array conversion can be applied. Data locality and, consequently, cache utilization are improved.

2. **Optimizations for Memory Energy.** The problem of optimizing for memory energy dissipation reduces to structuring the computations such that data accesses are localized to individual memory banks and defining an activation schedule to put unused memory banks in a sleep state. We will study optimizations for the kernels, making the assumption that they access a multi-level memory hierarchy with an in-memory processor that is capable of changing the data layout dynamically.

Run-Time Optimizations These optimizations are performed dynamically by the run-time system and the run-time optimizer that is output from the framework, using the performance models and the representation of the kernel.

1. **Data Remapping Optimizations.** Remapping between computation stages can significantly reduce processor-memory traffic and lower the consumed memory bandwidth—a critical resource limitation in many important applications. Data stored in memory can be efficiently remapped using in-memory processing features of HPC systems. The DOSA will choose components that perform the appropriate amount of remapping for the system and its current performance.

2. **Optimizations for System Throughput and Overall Energy Dissipation.** System throughput is a metric by which to evaluate the overall system performance in executing multiple instances of a kernel. We model system throughput as a flow optimization problem with costs associated with computation and communication, as well as memory accesses. In addition, we will represent the energy dissipation as another variable.

3 Representative Applications to be Studied

Three representative applications will be used as examples in investigating various dynamic optimizations and use on our framework demonstration. Our component library will incorporate the design-time optimizations, and the run-time optimizer will compose applications from these critical design points.

A. Molecular dynamics (MD) is a technique for the simulation of the interactions between the molecules in a system over time. It is used in a wide variety of fields, including nanotechnology and biology [10, 15]. In the largest simulations, there are over a billion molecules. The simulations are extremely time-consuming: to simulate a microsecond of time can take a week of compute time. As a result, there has been much work done toward developing scalable versions of MD to be executed on tightly coupled parallel machines and on cluster machines [2, 6, 12]. Even so, the most time-consuming part of the simulation is the non-bonded force calculation [1]. We, therefore, propose two memory-centric optimizations for non-bonded force calculation in MD simulations.

In MD simulations, the simulation space is broken into cells which are distributed among the processors of a parallel machine. Initially, molecules are randomly placed into the cells. The mapping between cells and molecules is kept in a linked list. During non-bonded force calculation, the positions of some molecules on each processor must be transmitted to other processors. We envision an optimization in which the in-memory processor uses cache injection to communicate these positions to the required processors, thereby reducing the amount of communication that must be carried out by each processor and ensuring that the data needed by each receiving processor is already in the cache.

In non-bonded force calculation, the processor must traverse the linked list to calculate the molecule interactions. Since the placement of molecules in simulation space is random, this linked list traversal leads to truly random memory access patterns. This randomness leads to poor cache performance. Therefore, we propose data remapping as an optimization for MD. An in-memory processor acts as a data remapping engine, traversing the linked cell list and reordering it such that during non-bonded force calculation, the data is accessed in a sequential manner.

B. Dynamic programming is a general problem-solving technique used widely in various fields such as control theory, operations research, cryptanalysis and security, and biology.

We select the computational biology problem of optimal local pairwise alignment using nucleotide and amino acid sequences for our compact application of this kernel. Pairwise alignment is used in similarity searching where uncharacterized but sequenced "query" genes are scored against vast databases of characterized sequences. Dynamic programming approaches give optimal alignments for this problem, such as the quadratic-time Smith-Waterman [13] algorithm. In addition to the data layout optimization, an-

other optimization we consider is performing the computation directly within the memory system. In this case, the core processor could issue a work parcel to the in-memory processors describing the recurrence in the dynamic programming algorithm, and the in-memory processors would easily handle the natural flow of computations with respect to the data dependencies.

C. Graph theoretic. We focus on sparse, irregular graph problems that are notoriously hard to solve efficiently in part because of a lack of spatial and temporal locality. Suppose a sparse graph $G\langle V, E\rangle$ is given. V is the set of vertices and E is the set of edges. Vertex $v \in V$ has multiple characteristics $f_1(v)$, ... $f_i(v)$. Edge $(u, v) \in E$ has multiple characteristics $g_1(u, v)$, ..., $g_j(u, v)$. **Finding paths:** Given a pair of vertices x and y, find all paths such that the number of vertices in the path is less than a specified number $min\_nodes\_in\_path$ and the degree of all nodes in the path is less than a specified degree min_degree **Characteristic-based graph matching:** Given a graph representation of an interaction pattern $P\langle V_p, E_p\rangle$ where each vertex $v \in V_p$ and each edge $(u, v) \in E_p$ has certain (possibly multiple) specific characteristics, find a sub-graph G' in G such that a certain similarity metric $s(G', P)$ is maximized. **Characteristic-based graph partitioning:** Find a partition $\langle V_1, V_2, \cdots, V_k\rangle$ of V so that vertices within each subset V_i is reasonably *clustered* and any two sets V_i and V_j are reasonably *separated*.

4. Concluding Remarks

The goal of our research is to develop a framework that supports design-time optimizations of applications in high-performance computing and whose output can then be used to perform further optimizations at run-time. For design-time optimization, we will study HPC architectures and define a hierarchical performance model for the memory subsystem associated with these architectures. To support this modeling, we will develop the DOSA framework. Doing so includes creating a metamodel based on the memory performance model to configure GME [5] and developing a library of components for common tasks in high-performance computing. Using the configured GME, we will define detailed models of several HPC architectures. We will define a semi-automatic design flow using the framework to optimize kernels using HPC architectures. To illustrate our ideas, we plan to use kernels from the domains of molecular dynamics, gene sequence analysis, and graph theory.

References

[1] M. Allen and D. J. Tildeseley. *Computer Simulation of Liquids*. Oxford University Press, New York, 1987.

[2] G. S. Almasi, C. Cascaval, J. G. Castanos, M. Denneau, W. Donath, M. Eleftheriou, M. Giampapa, H. Ho, D. Lieber, J. E. Moreira, D. Newns, M. Snir, and H. S. Warren, Jr. Demonstrating the scalability of a molecular dynamics application on a petaflop computer. Research Report RC 21965(98713), IBM Research Division, Yorktown Heights, NY, February 2001.

[3] D. Bader and G. Cong. Fast shared-memory algorithms for graph theoretic problems. *Journal of Parallel and Distributed Computing*, 66(11):1366–1378, 2006.

[4] D. Bader and G. Cong. Efficient parallel graph algorithms for shared-memory multiprocessors. In S. Rajasekaran and J. Reif, editors, *Handbook of Parallel Computing: Models, Algorithms, and Applications*. CRC Press, 2007.

[5] Generic modeling environment. `http://www.isis.vanderbilt.edu/Projects/gme/`.

[6] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[7] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[8] A. Ledeczi, J. Davis, S. Neema, and A. Agrawal. Modeling methodology for integrated simulation of embedded systems. *ACM Transactions on Modeling and Computer Simulation*, 13(1):82–103, January 2003.

[9] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason IV, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *Proc. of Workshop on Intelligent Signal Processing*, 2001.

[10] Z. Mao, A. Garg, and S. B. Sinnott. Molecular dynamics simulations of the filling and decorating of carbon nanotubules. *Nanotechnology*, 10(3):273–277, 1999.

[11] Model integrated computing. `http://www.isis.vanderbilt.edu/research/mic.html`.

[12] A. Nakano, R. K. Kalia, P. Vashishta, T. J. Campbell, S. Ogata, F. Shimojo, and S. Saini. Scalable atomistic simulation algorithms for materials research. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, New York, November 2001. ACM Press.

[13] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Molecular Biology*, 147:195–197, 1981.

[14] J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer Magazine*, 30(4):110–111, April 1999.

[15] P. Tang and Y. Xu. Large-scale molecular dynamics simulations of general anesthetic effects on the ion channel in the fully hydrated membrane: The implication of molecular mechanisms of general anesthesia. *Proceedings of the National Academy of Sciences of the United States of America*, 99(25):16035–16040, December 2002.