OPTIMIZING LARGE-SCALE SCIENTIFIC APPLICATIONS THROUGH THE INTEGRATION OF HIGHER-LEVEL SEMANTICS AND ANALYTICAL PERFORMANCE MODELING

by

JICHI GUO

M.A., University of Texas at San Antonio, 2012
B.A., University of Science and Technology of China, 2004

A dissertation submitted to the Graduate Faculty of the University of Colorado at Colorado Springs in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Department of Computer Science

2014
This dissertation for the Doctor of Philosophy degree by

Jichi Guo

has been approved for the

Department of Computer Science

by

Qing Yi, Chair

Jiayuan Meng

Pavan Balaji

Jia Rao

Xiaobo Zhou

Kristen Walcott-Justice

Date
OPTIMIZING LARGE-SCALE SCIENTIFIC APPLICATIONS THROUGH THE INTEGRATION OF HIGHER-LEVEL SEMANTICS AND ANALYTICAL PERFORMANCE MODELING

The scale and complexity of scientific computing have reached an unprecedented level, with applications now typically consisting of thousands of different components written in multiple programming languages (C/C++/Fortran) and dynamically interacting with each other in unpredictable fashions. The diversity and intricacy of machine architectures, on the other hand, have made matters worse, as developing high-performance applications entails careful management of a wide variety of parallel resources and a deep hierarchy of shared / distributed caches and memories. The complexity, heterogeneity, and sometimes highly dynamic nature of problems at hand have made it overwhelmingly difficult to properly understand and ensure both the correctness and efficiency of these applications on diverse platforms.

To address the above problem, this thesis develops an optimization approach to enhance the parallelization efficiency of large-scale scientific applications that seamlessly integrates domain-specific knowledge from developers to accurately model the performance bottlenecks of large-scale scientific applications and to effectively elevate the performance portability of these applications. The framework first interacts with developers to summarize the performance characteristics, computational structures, data access patterns, and expected runtime behaviors of applications using a collection of statistical modeling notations and optimization annotations. These notations are then combined with ROSE, an existing source-to-source C/C++/Fortran optimizing compiler, to automatically discover optimization opportunities. Finally, the application is optimized and tuned based on the optimization opportunities.
## TABLE OF CONTENTS

1 Introduction .......................... 1
   1.1 Motivation .......................... 1
   1.2 Thesis Statement ................... 5
   1.3 Integration of higher-level semantics ............... 6
   1.4 Analytical performance modeling ........... 9
   1.5 Optimizing large MPI applications ........ 10

2 Background and related work .......... 15
   2.1 Compiler optimizations For high performance computing .................. 15
      2.1.1 Dependence analysis ............. 16
      2.1.2 Compiler optimizations ......... 18
      2.1.3 Enhancing parallelization efficiency .......... 20
   2.2 Performance modeling .................. 21
   2.3 Empirical tuning of optimization configurations ........... 22
   2.4 Optimizing MPI applications ........... 23

3 Integrating higher-level semantics for optimization .......... 25
   3.1 The annotation language ............... 27
      3.1.1 Language semantics ............. 28
      3.1.2 Writing annotations ............. 29
   3.2 Enhanced inlining algorithm ............ 32
      3.2.1 Annotation-based Inlining .......... 33
      3.2.2 Automatic parallelization ...... 35
      3.2.3 Reverse-inlining ............... 35
      3.2.4 Correctness, efficiency, and generality .......... 36
   3.3 Automatic parallelization using Polaris .......... 38
      3.3.1 Loss of parallelism due to inlining .......... 38
      3.3.2 Missed opportunities ............. 41
   3.4 Experiment results .................. 46
      3.4.1 Enhancing automatic loop parallelization .......... 46
      3.4.2 Performance of optimized code ........... 49
   3.5 Conclusion .................. 50

4 Analytical Performance Modeling Of Scientific Applications 51
   4.1 Application skeleton ............... 54
      4.1.1 Control flow structures ........... 55
      4.1.2 Performance characteristics .......... 55
      4.1.3 Critical data-flow ............... 55
      4.1.4 Runtime hints .................. 56
   4.2 Skeleton generation ............... 57
      4.2.1 Design principles ............... 57
      4.2.2 Generating control flow structures ........ 57
6 Conclusion 101
   6.1 Publication ...................................................... 102

Bibliography 104
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Summary of the PERFECT benchmarks</td>
<td>46</td>
</tr>
<tr>
<td>3.2</td>
<td>Automatically parallelized loops using different inlining strategies</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>The hot spot selection quality, defined as the run time coverage of the first ( N ) projected hot spots compared to that of the first ( N ) profiled hot spots. Q.Modl and X.Modl refer to hot spot selections for BG/Q and Xeon based on our modeling, respectively. Q.Xeon refers to hot spot selection for BG/Q based on Xeon profiling. X.BG/Q refers to hot spot selection for Xeon based on BG/Q profiling.</td>
<td>70</td>
</tr>
<tr>
<td>4.2</td>
<td>Differences between the projected hot spot selection and the measured hot spot selection based on profiling. Zero means the set of ( N ) hot spots equals the top ( N ) hot spots. Q.Modl and X.Modl refer to hot spot selections for BG/Q and Xeon based on our modeling, respectively. Q.Xeon refers to hot spot selection for BG/Q based on Xeon profiling. X.BG/Q refers to hot spot selection for Xeon based on BG/Q profiling.</td>
<td>71</td>
</tr>
<tr>
<td>4.3</td>
<td>Performance characteristics for SORD’s top three hot spots. The number in the table represents the instruction count of row type generated by the column method. ( ld ): mem loads, ( st ): mem stores, ( fp ): floating-point inst, ( xp ): fixed-point inst.</td>
<td>78</td>
</tr>
<tr>
<td>4.4</td>
<td>Loop counts in the hotpath of SORD’s #1 hotspot</td>
<td>78</td>
</tr>
<tr>
<td>5.1</td>
<td>Micro benchmark of overlapped MPI_Alltoall</td>
<td>96</td>
</tr>
<tr>
<td>5.2</td>
<td>Differences between the projected hot spot selection and the measured hot spot selection based on profiling with 80% threshold for class B data on 4 nodes. Zero means the set of ( N ) hot spots equals the top ( N ) hot spots.</td>
<td>97</td>
</tr>
<tr>
<td>5.3</td>
<td>Differences between the projected hot spot selection and the measured hot spot selection based on profiling. Zero means the set of ( N ) hot spots equals the top ( N ) hot spots.</td>
<td>98</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

1.1 Overall workflow of SORD. Hot spots are in `diff_cn()` and `diff_cn()` that perform matrix computation. For each iteration in the while-loop, `swap_hallo()` exchanges the halo regions of the matrices between MPI processes. Data are read from or write to files in `file_io()`.

1.2 `swap_hallo()` in SORD

1.3 `field_io()` in SORD

1.4 Workflow of the overall optimization approach

1.5 Workflow of the annotation-based inlining for enhancing loop parallelization

1.6 Workflow of analytical execution flow modeling

1.7 The structure of NAS FT (1D layout) before (a) and after (b) overlapping computation and communication. Before and After in the figures are computation loops before and after the communication.

1.8 Workflow of the CCO optimization approach

3.1 The annotation language

3.2 Annotations for the opaque subroutine defined in Figure 3.13

3.3 Annotations for the subroutine in Figure 3.17

3.4 Automatic parallelization with annotation-based inlining

3.5 Annotations for the `MATMLT` subroutine in Figure 3.11

3.6 Call site of `MATMLT` in Figure 3.7 after parallelization

3.7 Call site of `MATMLT` in Figure 3.12 after annotation-based inlining

3.8 Call site of `MATMLT` in Figure 3.6 after reverse-inlining

3.9 A subroutine with automatically parallelizable loops at lines 3 and 6

3.10 A call site of the subroutine in Figure 3.9 (loops in PCINIT become no longer automatically parallelizable after inlining)

3.11 A subroutine with automatically parallelizable loops at lines 22, 23, and 26

3.12 A call site which invokes the subroutine in Figure 3.11 (loops in `MATMLT` are no longer automatically parallelizable after inlining)

3.13 A subroutine excluded from inlining by Polaris

3.14 A loop nest invoking the subroutine in Figure 3.13 (the inner $K$ loop at line 4 can be automatically parallelized after annotation-based inlining)

3.15 Definition of subroutine `GETCR` invoked by `FSMP` in Figure 3.13

3.16 Definition of subroutine `SHAPE1` invoked by `FSMP` in Figure 3.13

3.17 A subroutine which contains indirect references in array subscripts

3.18 A parallelizable loop invoking the subroutine in Figure 3.17

3.19 Performance of automatically parallelized code

4.1 An example code skeleton, and its corresponding block skeleton tree and bayesian execution tree

4.2 Building the hot path for Figure 4.1
4.3 Profiled and projected hot spot selections for SORD on BG/Q. Prof.Q refers to the hot spot selections resulted from profiling on BG/Q and it serves as the baseline. Modl.Q(p) and Modl.Q(m) show the projected and measured runtime coverage on BG/Q for the model-projected hot spot selections, respectively. Prof.Q(x) shows the measured runtime coverage on BG/Q for the hot spot selections obtained from Xeon profiling.

4.4 Profiled and projected hot spot selections for SORD on Xeon. Prof.X refers to the hot spot selections resulted from profiling on Xeon and it serves as the baseline. Modl.X(p) and Modl.X(m) show the projected and measured runtime coverage on Xeon for the model-projected hot spot selections, respectively. Prof.X(q) shows the measured runtime coverage on Xeon for the hot spot selections obtained from BG/Q profiling.

4.5 The projected performance breakdown for SORD’s hot spots on BG/Q.

4.6 The projected performance breakdown for SORD’s hot spots on Xeon.

4.7 The profiled performance insights for SORD’s hot spots on BG/Q. We show the issue rate and the computation intensity, calculated as the number of completed instructions divided by the number of L1 misses.

4.8 The SORD hot path for the top 5 hot spots on BG/Q. Note that a hot spot may be invoked from multiple places.

4.9 Profiled and projected hot spot selections for CFD. Prof, Modl refer to the hot spot selections resulted from BG/Q profiling and performance modeling. Modl (p) and Modl (m) show the projected and measured runtime coverage on BG/Q using the model-projected hot spots, respectively.

4.10 Profiled and projected hot spot selections for SRAD. Prof, Modl refer to the hot spot selections resulted from BG/Q profiling and performance modeling. Modl (p) and Modl (m) show the projected and measured runtime coverage on BG/Q using the model-projected hot spots, respectively.

4.11 Profiled and projected hot spot selections for CHARGEI. Prof, Modl refer to the hot spot selections resulted from BG/Q profiling and performance modeling. Modl (p) and Modl (m) show the projected and measured runtime coverage on BG/Q using the model-projected hot spots, respectively.

4.12 Profiled and projected hot spot selections for STASSUIJ. Prof, Modl refer to the hot spot selections resulted from BG/Q profiling and performance modeling. Modl (p) and Modl (m) show the projected and measured runtime coverage on BG/Q using the model-projected hot spots, respectively.

5.1 Sample hot path of NAS FT

5.2 Annotation for MPI_Send

5.3 Single or multiple computation loops with MPI_Test inserted

5.4 The micro benchmark used to test distribution of MPI_Test

5.5 Micro benchmark of overlapped MPI_Alltoall

5.6 Speech up of NPB with input B class on Intel x86

5.7 Speech up of NPB with input B class on Intel x64
Chapter 1
Introduction

1.1 Motivation

Scientific applications in areas such as numerical simulation, mathematical optimization, model fitting, and big data analysis, have reached such a scale, where a single application can include thousands of different components written in varying programming languages, e.g., C/C++/Fortran. Each component may dynamically interact with each other in such a highly complex fashion that is far beyond what a compiler could understand via conventional static program analysis techniques. Further, because many of the components do not have their source code available at compile-time, compilers must treat each file as a separate unit of abstraction, and assume the most conservative manner when reasoning about interactions of functions defined across different files (compilation units). As a result, interprocedural optimization becomes a luxury that is rarely applied, and even when applied, their effectiveness is seriously compromised due to the difficulty of extracting higher-level semantics from the low-level implementation details of the actual source code.

To demonstrate the complexity of understanding a real-world scientific application, Figure 1.1 shows the overall workflow of SORD, a parallel earthquake simulation application written in Fortran while using MPI [53] to support network communications among many of distributed application components. The application itself is relatively small, consisting of five thousands lines of source code separated into 28 separately compiled files and 184 subroutines. In addition to the difficulty of obtaining the source code of the many libraries, including MPI, used by the application, the following obstacles further complicate the task of understanding its logistics across procedural boundaries.

- The application itself invokes many standard libraries, e.g., MPI, but higher-level semantics of the library operations are not known to the compiler. For example, Figure 1.2 shows the swap_halo() function in SORD, which exchanges the halo region of a matrix $f4$. Not
Figure 1.1: Overall workflow of SORD. Hot spots are in `diff_cn()` and `diff_cn()` that perform matrix computation. For each iteration in the `while-loop`, `swap_halo()` exchanges the halo regions of the matrices between MPI processes. Data are read from or write to files in `file_io()`.

knowing the semantics of the many MPI calls, it is impossible for a compiler to understand the logistics of this function sufficiently to apply any aggressive reordering optimizations.

- Many arrays in SORD are defined as having different dimensions when used in different subroutines, especially between communication and computation functions. Figure 1.2 shows the `swap_halo()` function in SORD, where the matrix `f4` to exchange is declared as a 4D array. The same array is then invoked by `MPI_Sendrecv`, where the first parameter is declared as a 1D array. As the result, even if the compiler has the source code of MPI and the whole application, it cannot reconcile the different shape definitions of an array except by linearizing all the array dimensions into a single combined offset to the beginning of the array. Such a strategy, however, generates exceeding complex and often non-linear expressions that are hard (if not impossible) to analyze statically, making it difficult to determine the dependence relations among statements that access these arrays [56].

- Lacking input data, a compiler cannot accurately predict the statistics of the dynamic execution flows of the application and thus cannot adequately model its runtime behavior. For
! vector swap halo
subroutine swap_halo(f4, nh)
use mpi
real, intent(inout) :: f4(:,:,:,:)
integer, intent(in) :: nh(3)
integer :: i, e, prev, next, nm(4), n(4), isend(4), irecv(4), tsend, trecv, comm
nm = (/ size(f4,1), size(f4,2), size(f4,3), size(f4,4) /)
do i = 1, 3
if (np3(i) > 1 .and. nm(i) > 1) then
  comm = comm3d
  call mpi_cart_shift(comm, i-1, 1, prev, next, e)
  n = nm
  n(i) = nh(i)
  isend = 0
  irecv = 0
  isend(i) = nm(i) - 2 * nh(i)
  call mpi_type_create_subarray(4, nm, n, isend, mpi_order_fortran, rtype, tsend, e)
  call mpi_type_create_subarray(4, nm, n, irecv, mpi_order_fortran, rtype, trecv, e)
  call mpi_type_commit(tsend, e)
  call mpi_type_commit(trecv, e)
  call mpi_sendrecv(f4(1,1,1,1), 1, tsend, next, 0, f4(1,1,1,1), 1, trecv, prev, 0, comm,
                  mpi_status_ignore, e)
  call mpi_type_free(tsend, e)
  call mpi_type_free(trecv, e)
  isend(i) = nh(i)
  irecv(i) = nm(i) - nh(i)
  call mpi_type_create_subarray(4, nm, n, isend, mpi_order_fortran, rtype, tsend, e)
  call mpi_type_create_subarray(4, nm, n, irecv, mpi_order_fortran, rtype, trecv, e)
  call mpi_type_commit(tsend, e)
  call mpi_type_commit(trecv, e)
  call mpi_sendrecv(f4(1,1,1,1), 1, tsend, prev, 1, f4(1,1,1,1), 1, trecv, next, 1, comm,
                  mpi_status_ignore, e)
  call mpi_type_free(tsend, e)
  call mpi_type_free(trecv, e)
end if
end do
end subroutine

Figure 1.2: swap_halo() in SORD

element, Figure 1.3 shows the field_io() function in SORD. The function either loads field data from file f, or writes field data to file f, which is determined by whether the pass variable equals to ” > ” or ” < ”. The memory side effect of the function will be significantly different when the function is invoked with ” > ” or ” < ” at runtime. Using string parameters to control the functionality of procedures like field_io() is common in Fortran scientific applications.

Besides the difficulty of understanding the higher-level semantics of applications, another challenge in optimizing the execution efficiency of large-scale scientific applications is the wide variety of modern architectures that may be used to deploy these applications. Each architecture may have different numbers of multicore/manycore processors and hierarchies of private/shared caches and
! vector swap halo
subroutine field_io(passes, field, f)
...
do ipass = 1, len(passes)
   pass = passes(ipass:ipass)
   if (pass == '<') then
      ! read field from file f
...
   end if
   if (pass == '>') then
      ! write field to file f
...
   end if
end do
...
end subroutine

Figure 1.3: field_io() in SORD

shared/distributed memories, connected by networks of a wide varieties of latencies/bandwidths. For example, the requirements of optimizing SORD on the BlueGene/Q cluster, a supercomputer with 2K nodes, each node with 16 1.6GHz PowerPC A2 processors, 16 KB instruction and data cache per core, 32MB shared L2 cache, and 16GB shared memory, and all nodes interconnected via InfiniBand, is very different from those of optimizing the same application on an alternative HP data center cloud server with three racks of ProLiant BL460C G6 blade server modules, each node with 8 3.2GHz Intel Xeon processors interconnected with 1 Gbps Ethernet. While modern programming languages provide sufficient support of functional portability so that an application written in C/C++/Fortran combined with popular parallelization models such as OpenMP/Pthread-/MPI/OpenCL can be ported to an arbitrary machine and work correctly, the application performance could vary by an order of magnitude. To effectively optimize an application on the wide variety of architectures, a compiler not only needs a reasonable understanding of the higher-level semantics of the key data structures and algorithms used in the applications, it but also needs an accurate model of the dynamic interactions between the application and the different components of the architectures. In particular, the performance bottlenecks of an application are usually
hardware-dependent, and its optimization space needs to be carefully tuned to support the performance portability of the compiled code.

1.2 Thesis Statement

The thesis aims to enhance the parallel efficiency of large-scale scientific applications by providing means for compilers to better integrate the higher-level semantics of separately compiled data and algorithm abstractions and to better understand the performance bottlenecks and optimization needs of these applications via accurate analytical performance modeling.

**Figure 1.4:** Workflow of the overall optimization approach

Fig 1.4 shows the overall optimization workflow, which contains four components: (1) integration of higher-level semantics, which summarizes higher-level semantics of the input program including algorithmic-level workflow and data structure, and outputs annotated code that integrates the semantics into the source code of the input program; (2) analytical performance modeling, which integrates hardware performance models to construct an execution flow model from the input program and hardware specification to summarize the application’s runtime behavior on the target hardware, including the runtime code paths and the corresponding estimated performance; (3) the optimization component, which combines execution flow model and annotated code to enhance the effectiveness of safety and profitability analysis to optimize the input program; and (4) the tuning component, which tunes the performance of the optimized code for
the target runtime environment, and the analytical performance model is used to prune the large optimization space.

Within the workflow, my approach relies on two existing source-to-source compiler frameworks, The ROSE C/C++/Fortran compiler developed at DOE/LLNL [98] and the POET program transformation and tuning framework developed at UTSA/UCCS [120], for the optimization and tuning components. This thesis focuses on developing the two new components, the integration of higher-level semantics for optimization, and analytical performance modeling of large applications, to enhance the effectiveness of compiler optimizations. Combining the two approaches together, a framework is at last developed to enhance the overlapping of communication and computation in large-scale scientific MPI applications.

1.3 Integration of higher-level semantics

For an arbitrary scientific application, we define its higher-level semantics as the algorithm-level data and control flow of the subroutines and the intended operational logistics of data abstractions. As multi-core architectures become ubiquitous in modern computing, optimizing compilers need to automatically extract parallelism for large scale scientific applications composed of many subroutines. By summarizing the control and data flow of the subroutines using a small annotation language to emphasize the higher-level semantics while abstracting away the low-level implementation details of each subroutine, the higher-level summaries of the subroutines are then used to substitute each function call within the application, allowing better understanding of the application logistics across procedure boundaries.

Inlining is a well-known compiler transformation which substitutes procedure invocations with their corresponding implementations to erase artificial procedural boundaries [24]. Our annotation-based inlining approach is similar to conventional inlining except that the callee will be substituted by the code generated from the higher-level annotations instead of the low-level implementations from the source code. The annotation-based inlining approach offers the following advantages over conventional inlining.
• Inlining can be applied even for subroutines defined in external libraries without their source code and for recursive subroutines because developers can provide a high-level summary of the semantics of these subroutines. In contrast, conventional inlining cannot be applied to recursive procedures or subroutines defined in external libraries where the source code is not available.

• The potential code size explosion problem can be avoided entirely as the inlining transformations will be reverted back to the original call statements after optimization. In contrast, when excessively applied, conventional inlining can cause code size explosion and curtail the compiler’s effectiveness in applying optimizations (e.g., automatic loop parallelization and register allocation) due to the increased code complexity resulted from inlining.

• The user-supplied annotations do not need to include irrelevant implementation details of the subroutines of interest. Therefore, after annotation-based inlining, the code within the caller can be easily analyzed using conventional compiler techniques, as the inlined code reflects the control and data flow of the algorithm at a higher semantic level, without being made difficult to understand by discrepancies in low-level implementations across procedural boundaries, e.g., inconsistent array dimensions and unknown library calls.

Figure 1.5 shows the workflow of the enhanced inlining approach. In contrast to conventional inlining, which substitutes a procedure invocation with the complete implementation of the callee, we use annotations, which summarize the computational structure and side effects of the callee, to replace the invocation. The inlined code is then optimized by a conventional compiler which applies sophisticated loop dependence analysis to automatically parallelize loops via OpenMP when safe. The optimized code from the compiler is then piped into a reverse inliner, which reverts the earlier inlined code back to using the original procedure invocations but leaves the optimizations intact. The output from the reverse inliner is essentially the original input code optimized with automatic parallelization, where the annotations, currently manually provided by developers, have been used to enable more aggressive loop optimization in spite of opaque procedure calls.
There are two key insights behind the inlining approach:

1. Program analysis should not require transforming the actual code, but inlining will. The disadvantages of conventional inlining come from its permanent transformation to the code. There are also other improved inlining approaches based on this insight, such as inline tri-als [36].

2. The inlined code might still not be analyzable because of the opaque data structures and dynamic code execution paths. Domain knowledge of the procedures’ runtime behaviors could help improve the precision of data-flow and control-flow analysis.

This thesis investigates the techniques that enhance the effectiveness of inlining to support more aggressive loop parallelization by optimizing compilers. In particular, after identifying limitations of conventional inlining while using the Polaris compiler [19] to parallelize a collection of Fortran77 applications from the PERFECT benchmark suit [18], a new inlining approach is presented to overcome these limitations. And the experiment results show that the new approach can elimi-
nate negative impact of conventional inlining while enhancing the effectiveness of interprocedural parallelization for a majority of the PERFECT benchmarks.

1.4 Analytical performance modeling

A critical difficulty faced by compilers when optimizing large scale scientific applications is to effectively model the application’s behavior on any target hardware. To overcome this difficulty, this thesis aims to understand potential application behavior on conceptual systems through analytical modeling of their interactions and thereby to gain insights about performance bottlenecks, and discover optimization opportunities for both applications and hardware design. In particular, we aim to answer two key questions about application behavior: (1) what are the hot regions (i.e., the most time-consuming regions of the application), and (2) what are their performance bottlenecks. The hot regions of a workload are often determined by its execution flow, defined as the input-dependent run time traversal of the application code. The concept includes two aspects: hot spots and hot paths. A hot spot is a single code block (e.g., a loop or a function) that consumes a large portion of the application run time; a hot path is the subset of the application execution flow that connects all the hot spots. Such information can be used to enhance profitability of the optimization and prune large optimization search spaces.

Profilers such as gprof [51] are often used to find hot regions; however, they are only available on existing, accessible systems. Micro-architecture simulators, on the other hand, often take an enormous amount of time to simulate even a single kernel with a reasonable sized data set. Moreover, simulators treat applications as black boxes and are oblivious to workloads’ control flow structure. Implementing a profiling tool over a simulator, however, is a daunting task and may lengthen the simulation time even further.

To quickly gain first-order insights about application behavior, specifically, hot regions and reasons behind their performance bottlenecks, this thesis presents an approach that analytically models the holistic execution flow of an application and then employs parameterized hardware performance models to project application performance for varying architecture design configura-
tions. As shown in Figure 1.6, the overall workflow of the approach includes three steps. First, a source-to-source application analysis engine, named skeletonizer, analyzes the input code to generate a structural description of the application in the form of the SKOPE workload modeling language [85]. The structural description, named a code skeleton, depicts the application’s control flow, data flow, communication, and computation intensity, together with frequencies of its control-flow branches gathered from profiling the application on a local machine. Second, based on the resulting code skeleton, the performance analysis engine automatically constructs a Bayesian Execution Tree (BET) to statistically model the application execution flow. The BET are then combined with a hardware performance model parameterized with the appropriate architecture configuration to later to identify computation and communication hot spots, and extract hot paths for optimization profitability analysis. Finally, the performance projection for potential hot regions and their performance bottlenecks are projected and reported.

This approach essentially uses statistical execution flow models of applications and coarse-grained performance models of hardware to predict their interactions. Without actual performance measurement, my technique trades off accuracy for more insights and less resource requirement. By mathematically modeling the repetitive control flow caused by looping, the analysis and projection time does not scale with the input data size or the actual execution time of the application. By modeling the application execution flow, this approach is able to gain “contextual” insights such as the control flow leading to individual hot spots, the relations among hot spots, and sizes of input data associated with each invocation of a hot spot. Moreover, it can project whether computation or memory access is the bottleneck.

1.5 Optimizing large MPI applications

As computing platforms migrate towards clusters of heterogeneous multicore/manycore processors connected by varying network capacities, applications often need to manage the distributed memories of the heterogeneous processors via explicit message passing runtimes, e.g., MPI. Since the latency and bandwidth of the communication network is often hard to predict a priori, appli-
cation performance is often critically determined by their abilities to flexibly overlap communications with local computations, thereby minimizing wait time. As a proof of the concept, this thesis applies our optimization framework for integrating higher-level semantics and analytical performance modeling of large-scale applications to enable better overlapping of computation and communication (CCO) for MPI applications. In particular, annotation-based inlining is used to integrate domain knowledge to enable accurate dependence analysis of the applications to ensure the safety of optimization, and analytical performance modeling is used to select the profitable computations and communications to overlap and to prune the tuning space for optimization.

To illustrate the CCO optimization, Figure 1.7a shows the structure of the NAS FT benchmark [11], which applies FFT transformation to a 3D matrix through a loop that interleaves the computation of scaling the input matrix, with a collective communication of MPI_Alltoall to exchange data among the processes, which is then followed by a final transposition of the resulting matrix. The clear separation of computation and communication phases makes the algorithm design easy to implement and maintain. Additionally, the communication buffers can be reused.
The structure of NAS FT (1D layout) before (a) and after (b) overlapping computation and communication. Before and After in the figures are computation loops before and after the communication.

across different loop iterations, saving memory. However, the blocking MPI communication requires all processes to wait while the MPI_Alltoall operation is in progress. Consequently, unless executed on a platform with the fastest network connections, the performance of the application is likely to suffer due to the excessive wait time.

Figure 1.7b illustrates how the structure in Figure 1.7a may be modified to better overlap computation with communication. In particular, the MPI_Alltoall operation is decoupled into two finer-grained operations: a non-blocking MPI_Ialltoal and a blocking MPI_Wait. Then, by reordering the computation and communication across loop iterations, some computation of iterations \( I - 1 \) and \( I + 1 \) are moved in between the decoupled MPI operations of the \( i \)th iteration. Finally, MPI_Test operations are inserted into the computation to ensure the progress of the non-blocking communications\(^1\) such as solving graph problems to find the shortest communication paths and managing buffers and locks for each network request and reply. By reordering the communication and computation as shown in Figure 1.7b, the communication of each iteration can be overlapped with the computations of the previous and next iterations, allowing the applications to perform

\(^1\)Although MPI communications do not need full usage of CPU, they need some CPU time, provided when operations such as MPI_Test and MPI_Wait are invoked, to make progress.
well even on systems where the network latency is long, and the communications can take a long time to complete.

While the structure in Fig 1.7b enables MPI applications to better tolerate slow network connections, the non-blocking communication could take longer time to finish than the original blocking operation, more memory could be required to hold the data during communications, and the MPI_Test operations, when inserted too frequently, can slowdown the computations. In particular, to guarantee the profitability of the CCO (Computation-Communication Overlapping) optimization, any communication slowdown from the use of the non-blocking operations must be fully overlapped with the local computation, and the insertion of MPI_Test operations should only cause marginal slowdown of the local computation so that its effect is insignificant when compared with the reduction of the original communication time.

**Figure 1.8**: Workflow of the CCO optimization approach

Fig 1.8 shows the optimization workflow for automatically enabling overlapping of communication and computation in large MPI applications and thereby enhancing their performance portability. The workflow contains three key components: (1) the *performance modeling* compon-
nent, which analyzes the runtime statistics of an MPI application to extract a Bayesian Execution Tree[85] representation of its execution flow, including the frequencies of various runtime code paths and their performance characteristics such as computation intensities, working set sizes, communication characteristics of MPI operations; (2) the CCO analysis component, which identifies hot computation and communication regions that are likely to benefit from the CCO optimization and summarizes the optimization configuration based on profitability and safety analysis of the optimizations; and (3) the CCO optimization and tuning component, which applies the appropriate program transformations by replacing the blocking MPI operations with non-blocking ones, by reordering the computations and communications involved, and by inserting MPI_Test operations with a frequency determined by empirical tuning of the optimized code.
Chapter 2
Background and related work

The topics studied in this dissertation span across four main subject areas: (1) compiler optimizations for high performance computing, specifically using compiler-driven dependence analysis at varying scopes, from locally within a single basic block to interprocedurally across the whole program, to automatically and safely improve the parallelization efficiency of scientific applications; (2) application performance analysis and modeling, which aims to systematically identify the performance bottlenecks of large-scale applications and determine when and where to apply profitable optimizations [107]; (3) automated tuning of optimization configurations, which seeks to empirically determine the configuration of compiler optimizations, e.g., blocking factor [44], parallelization granularity [28], scheduling of tasks/threads [50], to ensure the performance portability of the optimized applications; and (4) the optimization of MPI applications, which identifies critical techniques to effectively enhance the parallelization efficiency of applications that use the MPI programming model [53] on a variety of modern computing platforms. The following summarizes the basic concepts and the state of the art in each area.

2.1 Compiler optimizations For high performance computing

Compiler optimizations usually refer to a set of program transformations applied to the source code by a compiler offline to maximize the performance of the application. It is commonly done by removing redundant computations and/or reordering computations. Safety and profitability are two critical constraints to determine whether and how to apply an optimization: (1) the optimization should not change the content and order of the output of the application for arbitrary program input; and (2) the performance of the result code should be always or usually better than the original code. For the first safety constraint, an aggressive compiler optimization might sometimes change the output, if the corresponding input is illegal according to domain knowledge that will never happen, or if the change is tolerant under domain context. For example, a loop parallelization
transformation might change the order of the debug output from each iteration, which might be OK to the developer. For the second profitability constraint, a critical concern is the performance portability of the optimize code. Because the performance is hardware-dependent, an application optimized for one platform might become slower on other platform, it is challenging to have an optimized code to outperform on all platforms. For example, the optimal factor for the loop tiling optimization is affected by the sizes of the cache hierarchy, which could be different on modern architectures.

2.1.1 Dependence analysis

Dependence analysis produces the execution order constraints between instructions or statements. If there is a statement $S_1$ depends on $S_2$, statement $S_1$ must be executed before $S_2$. Such dependence could be caused by either the data accesses (memory load/store) in the two statements or control structures (such as branches and loops), which categorizes the dependence into data dependence and control dependence. Dependence analysis is indispensable to compiler optimizations that require reordering statements.

Dependence analysis for scientific applications is a classical problem since the end of 1980s, when the focus is on extracting data dependence from array accesses and the analysis process happens offline at compile-time [12, 82, 62, 95]. To further improve the accuracy of the analysis, the data structure and data access patterns are needed to be known at compile-time, which, however, could be runtime behavior of the application, or opaque in the source code that is not available or challenging to recognize. To address the challenge, recent research focuses on extracting data access patterns at runtime, or directly getting the information from developers. There are runtime speculative approaches to check if two array or pointer references are the same [76, 77]. There is online dynamic dependence analysis for parallelized workflow through instrumentation [71, 72]. There are runtime dependence profilers [75, 124]. And there are interfaces to get user-input dependence via graph reachability [102].

Depending whether the code region to analyze contains function calls, the dependence analysis
could be categorized into intra-procedural or inter-procedural dependence analysis. The scope of
the analysis could hence be (1) local, when the analysis happens at some blocks within a func-
tion, (2) global, when the entire function is the target to analyze, (3) regional, when a couple of
related functions in the call graph are the target to analyze, or (4) whole-program, when the entire
application is analysis target. The accuracy of the analysis could be improved by looking at more
code context through inter-procedural analysis, which could on the other hand increase the analysis
time. To control the balance to effective optimize large numbers of deeply-nested function calls
buried in the large code base, region-based inter-procedural optimization are usually used, such as
whole program optimization regions [109] and region-based compilation [60, 83].

Beyond inter-procedural analysis, Inlining is another widely adopted technique used by compil-
ers to erase procedural boundaries and apply optimizations to larger regions of code [4, 10, 105].
Ayers et. all [10] shown that aggressive inlining and cloning based on profiled information can
dramatically improve the effectiveness of a large number of back-end optimizations. However,
when excessively applied, inlining can cause the well-known code-explosion problem [33] and
could degrade the effectiveness of many compiler optimizations as the input code becomes over-
whelmingly large and complex. Previous research has developed a variety of heuristics, including
temperature heuristics [126], demand-driven online transformation [112], inlining trials [36], and
interprocedural flow analysis [7, 69], to selectively apply inlining so that performance benefits can
be gained without incurring serious problems [6]. As the profitability of the inlining is mainly
input-dependent on the runtime code path and data sizes, other directions include automatic tuning
of different inlining heuristics [25], and combining inlining with hot code outlining [125].

The approach in this thesis that uses annotation to summarize to summarize the semantics of
opaque subroutines is similar to the semantic inlining approach by Wu et al.[118, 117], which al-
 lows their compiler to treat user-defined abstractions as primitive types in Java. The Broadway[58]
and DyC[52] compilers used annotation languages to guide domain-specific optimizations and
dynamic compilation of C code. Annotations and semantic specifications have also been used
extensively to specify dynamic properties of lower-level implementations in program verifica-
The main difference between my annotation-based inlining and existing approaches is that my approach will generalize legacy C/Fortran code to inline so that existing optimization techniques don’t need to understand the semantics of the annotation in order to optimize the annotated code.

### 2.1.2 Compiler optimizations

A large collection of compiler optimizations have been developed to enhance application efficiency by loop optimizations, data-flow optimizations, function optimizations, and code generator optimization. Loop optimizations target on loops that mainly include loop fusion, fission, splitting, interchange, tiling, skewing, unroll and jam, software pipelining and automatic parallelization (constant propagation and partial redundancy elimination). Data-flow optimization by analyzing data flow graph could achieve constant propagation, partial redundancy elimination, dead code elimination, and induction variable elimination for loops. Function optimizations include function inlining, specialization, outlining, tail recursion elimination, and automatic parallelization for recursive functions. Code generator optimizations happen when generate assembly or machine code from optimized intermediate representation of the application, such as register allocation, instruction selection and scheduling, and SSE optimization. Based on the bottleneck hardware resources to optimize, the impacts of the optimizations could be categorized into memory, CPU, and parallelization.

**Optimizing memory locality** In the modern architecture, the latency of a memory access is determined by the level it hits on the memory hierarchy: in the raw memory, in L3/L2/L1 cache, or in a register. Since the number of registers are very limited, most of recently used data are kept in the cache, which exposes optimization opportunity for time and space locality of data accesses. After a cache miss cached by memory access such as `mov eax, dword ptr ds:[0x1000]`, a cache line of the data continuous in memory after `ds : [0x1000]` will be fetched into the cache. If a later memory access location is also at `ds : [0x1000]`, it will hit the cache, which will produce
a cache reuse for time locality, If a latter instruction accessed \( d_s: [0x1004] \) that is near the previous accessed location, it will produce a cache reuse for space locality, which memory accesses are near in space. For example, loop tiling is widely used to improve data locality by reordering loop iterations so that the iterations accessing the same memory locations will be executed closed in time.

**Optimizing CPU performance** In modern CPU, instructions are executed out-of-order in parallel. As a result, the total number of cycles to execute a instruction sequence is affected by both the number of stalls needed to flush the instruction pipeline and dependence constraints to reorder instructions, which create opportunity to exploit instruction level parallelization to reduce average cycles per instruction (CPI). For example, loop unrolling is widely used to improve CPU performance. It could reduce the loop branch instructions that could cause stalls while providing more independent instructions from adjacent iterations to schedule, and hence reduce the average CPI to execute the loop.

**Parallelization** Parallelization by compilers focuses on loops, which is usually the major components of the hot spots of an application. Loop dependence analysis is usually used for safety analysis to identify parallelizable loops. Additionally, as parallelization could involve overheads for synchronization and buffering, profitability analysis is needed to determine whether and how to apply the parallelization so that the parallel speedup can overshadow hide the overheads. There are domain heuristics or empirical formula. There are speculative and runtime approaches. And there are programming interfaces to allow developers to adjust the parallelization strategies.

There are a number of existing commercial optimization compilers for C/Fortran, such as ICC, GCC, and LLVM. All of them support automatic loop parallelization (ICC, GCC AutoPar [9], and LLVM Poly [93]). (ICC [66], GCC AutoPar [9], and LLVM Poly [93]). There are also research compiler frameworks for sequential optimization ( COINS [31], Open64 [87], PIPS [92], ROSE [98], ) and parallelization ( SUIF [116], PLUTO [23], Paradigm [57], FortranD [42], Polaris [20], and others [5, 22]) ).
2.1.3 Enhancing parallelization efficiency

Limited by the energy wall, multicore/manycore processors are widely used in the modern architecture to increase computation power instead of increasing CPU frequency. Parallelization is a form of computation to carry out many calculations simultaneously in order to take advantage of multiple processors. Depending on the way to divide and schedule the computation workflow, parallelization can be categorized into data parallelism and task parallelism. Data parallelization focuses on distributing data to process, while task parallelism focuses on distributing computation workflows.

To apply the parallelization optimization, it is challenging for compilers to guarantee the safety and profitability constraints of the optimization. The safety constraint needs to reorder statements that requires accurate dependence analysis, which is hard to compute for code that uses indirect addressing, pointers, recursion, or indirect function calls. The profitability of the parallelization is affected by the input-dependent numbers of iterations and scheduling policy, which could be runtime-dependent on the application’s input parameters and dynamic behavior. It is especially challenging to parallelize irregular control flow structures (while, or loops with continues or breaks) at compile-time.

Existing approaches focus on integrating domain knowledge about the application’s runtime behavior to enhance parallelization efficiency. Bases on the source of the domain knowledge, existing approaches can be classified into two categorizes.

1. Provide programming interface (OpenMP [21]), domain-specific parallelization libraries or languages (TBB [68], Cilk [67]), or directly interact with developers (SUIF [116]) to get domain knowledge to guide parallelization.

2. Speculative parallelization [63] or profiling-based parallelization [74] that automatically get domain knowledge at runtime.
2.2 Performance modeling

Performance modeling is an approach in order to understand application behavior on specific hardware on specific hardware that first constructs a model from the data of the application’s existing behavior on the hardware and then use the model to reason the future behavior of the application. To optimize a large-application, either the compiler or the developer must attain a thorough understanding of its expected runtime behavior on a variety of modern architectures, e.g., the most time consuming code regions and bottlenecks hardware resources, which could be obtained through performance modeling. There are two key challenges for performance modeling: (1) how to get the information about the application’s dynamic behavior, and (2) how to reason the lower-level information to construct a higher-level model. To address the challenges, existing techniques using various tools including profilers, architecture simulators, and domain-specific performance models.

Profiling is widely used to collect runtime performance of the application. A variety of software tools, e.g., TAU [104], HPCToolkit [1], among others [51, 37, 99, 49], monitor application performance through instrumentation, sampling, and analytical interpretation of runtime statistics to help developers identify potential performance bottlenecks. These tools rely on actual execution of the workload. As a result, they require a fully functional software stack on the target hardware, which may not be available when studying newly adopted or future hardware. Most existing profiling tools focus on individual functions and statements and lack a high level picture of how they are connected at runtime. It is well recognized application performance cannot be fully modeled with precision analytically, due to the many unknowns in application behavior and architectural intricacies. Consequently, automated empirical tuning has been adopted by researchers to guide the optimization of both domain-specific libraries [113, 45] and general-purpose compilers [47, 61, 120].

When the is actual target platform is not available, microarchitecture simulators have been widely used to study prospective hardware [17, 8, 70]. However, developing detailed simulators
for new emerging architectures is an extremely time-consuming task. By processing an application in its binary form, it often takes days or even weeks to simulate a large application, and it is difficult to trace the behavior to the source level to gain additional insights.

Beyond profilers and simulators, application or hardware-specific models have been used in many scenarios to study workload performance and to guide application optimizations [29, 48], where the applications are usually run at a small scale to obtain knowledge about their runtime overhead and performance scaling. Snively et al. developed a general modeling framework [106] that combines hardware signatures and application characteristics to determine the latency and overlapping of computation and data movement. An alternative approach uses black-box regression, where the workload is executed or simulated over systems with different settings, to establish connections between system parameters and runtime performance [14, 81, 80, 107].

My research essentially augments static control flow of programs with input-dependent runtime statistics to effectively model the runtime behaviors of applications. Similar to my work, Narayanan et al. [86] modeled performance characteristics as an expression of input variables but did not analytically model the control flow structure and did not focus on identifying hot regions. Ertvelde and Eckout proposed the idea of using generic algorithms to synthesize shorter and more representative workload [110]. Their machine learning approach has to be trained by running various configurations of the workload on a target hardware.

2.3 Empirical tuning of optimization configurations

Empirical performance tuning is an optimization approach to find the best optimization parameters by running the optimized application the target runtime environment and select the configuration that could produce the best performance. Empirical performance tuning has been used to build many successful scientific libraries, including ATLAS [114], FFTW [46], among others [16, 111, 96], which use specialized kernel generators to parameterize and orchestrate differently optimized code. More recent research on iterative compilation has empirically modified the configurations of general-purpose compiler optimizations based on performance feed-
backs [78, 97, 13, 91, 89]. The key challenge for tuning optimizations for large-scale applications is how to reduce the large search space of whether turning on the optimizations, optimization parameters, and orders of optimizations.

There are existing approaches that integrate domain knowledge from developers to reduce the search space. The work by Hall et al. [59] allows developers or search engines to provide a sequence of loop transformation Recipes to guide transformations performed by an optimizing compiler. The X language [38] uses C/C++ pragma to guide the application of a pre-defined collection of compiler optimizations.

Previous tuning approaches have adopted a wide variety of search algorithms, including both optimization-specific algorithms that are custom made for a tuning framework [114, 16, 111, 43] and generic algorithms that are oblivious of the optimizations being tuned [88, 97, 123, 103], combined with model-driven search where compiler models are used to prune the space before tuning [108, 97, 27, 79]. Seymour, You, and Dongarra [103] studied the relative efficiency of 6 different generic search algorithms in terms of their abilities to find the best performance candidates under varying time limits.

Beyond search algorithms, static performance models have also been used in both domain-specific tuning frameworks [43, 122] and general-purpose iterative compilation [27, 30, 13] to improve the efficiency of tuning. Chen, Chame, and Hall [27] used models within a compiler to prune the search space before using generic search algorithms to tune memory optimizations such as tiling, unroll-and-jam, array copying, and scalar replacement. Recent research has adopted predictive modeling from machine learning to statically build optimization models from a representative training set of programs [2, 39]. The learned models are then used to automatically determine optimization configurations for future applications without any additional tuning.

### 2.4 Optimizing MPI applications

The performance of MPI applications is bound by both computation and communication. Compiler optimization can be used to optimize intra-node computation. Existing MPI optimization ap-
Existing approaches to overlap computation and communication (CCO) focus on automatic optimization safety analysis and improving the accuracy of the profitability analysis. Compiler techniques have been widely-used for automatically or semi-automatically applying CCO transformation. Recent directions include enhancing the accuracy of the dependence analysis for CCO [90], formulating memory side effect of MPI APIs for compiler optimizations [35], and automating the different CCO transformations [34, 40]. While the safety-analysis to apply CCO is automated/semi-automated using compiler’s dependence analysis, these approaches still rely on developers to properly determine when to apply the optimization to achieve better profitability. CCO optimization usually involves replacing blocking communication with the non-blocking version, inserting MPI_Test, and replicating buffers, which could slow down the overall application performance. To reduce the overheads introduced by the CCO application, [101] combines empirical tuning with network models and quantifies the potential benefit of overlapping communication and computation in large-scale scientific applications; [64] identifies the consumer-produce model from the control graph of the application to guide CCO decision.

Beyond overlapping blocking computation with blocking/non-blocking pt2pt/collective communication on slow network environment, there are also approaches on overlapping MPI with non-blocking computation in multi-threading models such as OpenMP [73]; overlapping CCO one-sided communication[15], the CCO patterns for high-speed networks [65] to overlap large amount of fast messages.
Chapter 3
Integrating higher-level semantics for optimization

As multi-core architectures become ubiquitous in modern computing, optimizing compilers need to automatically extract parallelism for large scale scientific applications composed of many subroutines. For example, a number of research compilers [20, 57, 116, 42] exist that are able to automatically parallelize loops in the source code by inserting OpenMP pragmas. However, these compilers are often forced to be overly conservative when encountering opaque subroutine invocations, which often have unknown side-effects. Inlining is a well-known program transformation which substitutes procedure invocations with their corresponding implementations to eliminate procedural boundaries [24]. However, conventional inlining cannot be applied to recursive procedures or subroutines defined in external libraries where the source code is not available. Moreover, when excessively applied, it can cause code size explosion and curtail the compiler’s effectiveness in applying optimizations (e.g., automatic loop parallelization and register allocation) [32], due to the increased code complexity resulted from inlining. The matter becomes further out of hands when the inlined codes becomes overly complex, e.g., from linearization of array parameters, so that they can no longer be understood at all by static program analysis performed by the compilers. Note that parallelization analysis often requires the higher-level knowledge of the runtime code paths and data access patterns within large applications in order to optimize such applications, and such knowledge are frequently unavailable until runtime even after extensive inlining.

To enable parallelizing compilers to effectively understand the higher-level semantics of large-scale scientific applications with deeply-nested procedures, this thesis investigates techniques that enhance the effectiveness of inlining to support more aggressive loop parallelization by optimizing compilers. In particular, after identifying limitations of conventional inlining while using the Polaris compiler [19] to parallelize a collection of Fortran77 applications from the PERFECT benchmark suit [18], a new annotation-based inlining approach is used to overcome these limitations, thereby significantly improving the effectiveness of interprocedural parallelization. In particular,
we found that disagreement of array shapes within linearization of multi-dimension arrays are often applied to avoid mismatching of dimensions between caller and callee, while the shape of arrays are lost and increase difficulties for later analysis. And the experiment results show that the new approach can eliminate negative impact of conventional inlining while enhancing the effectiveness of interprocedural parallelization for a majority of the PERFECT benchmarks.

The rest of this chapter first discusses the limitations of conventional inlining while studying automatic loop parallelization in the Polaris compiler [?] in Section 3.3. Then, Section 3.1 discusses the annotation language we designed to express the higher-level semantics. Section 3.2 discusses the annotation-based inlining approach to integrate higher-level semantics into the call site. To evaluate the approach, we combine annotation-based inlining to enhance automatic loop parallelization, and the experiment results are presented in Section 3.4. At last, the conclusion of this approach is drawn in Section 3.5.
3.1 The annotation language

To improve the effectiveness of automatic loop parallelization when encountering situations discussed in Section 3.3, my approach seeks to enhance the role of inlining so that higher-level semantics of subroutine invocations can be made readily available to compilers. In particular, we use the following steps to enhance the effectiveness of conventional inlining in supporting automatic parallelization across procedural boundaries.

1. Annotate important subroutines to summarize their side effects and loop structures required for accurate dependence analysis. Then, substitute subroutine invocations with the corresponding annotations instead of the actual detailed implementations.

2. Use Polaris to apply conventional loop dependence analysis and automatically parallelize loops when safe by inserting OpenMP directives.

3. Apply a reverse inlining step which substitutes all the inlined annotations with appropriate invocations of the original subroutines. After this step, the only remaining transformation to the original input code is the parallelization of loops via OpenMP.

The reverse inlining step essentially reverses all the transformations introduced by annotation-based inlining so that the original input code can benefit from advanced compiler optimizations without sacrificing its modularity. The correct application of this step requires all the inlined annotations be recognized and mapped back to correct invocations of the original subroutines, which can be easily accomplished when minimal transformations, e.g., insertion of OpenMP directives, have been made to the inlined code. However, the task is more challenging when interacting with more drastic program transformations, e.g., loop blocking and software pipelining. Section ?? discusses these issues in more detail.
Figure 3.1: The annotation language

3.1.1 Language semantics

The following first discusses the annotation language and then illustrates how to use annotations to summarize the higher-level semantics of subroutines and enable more effective parallelization after inlining.

Figure 3.1 summarizes the syntax of the annotation language, which can be used by developers to describe the side effects and loop structures of important subroutines. When these annotations are used to support subroutine inlining, a compiler can correctly recognize the dependence constraints carried by each subroutine invocation and subsequently successfully parallelize surrounding loops when safe.

The statements supported by the language include loops, if-conditionals, assignments, variable declarations, and return statements. They are used to summarize the control-flow structure and memory side effects of each subroutine. For implementation details that cannot be expressed using these statements, we provide two special operators, \texttt{unique} and \texttt{unknown}, to summarize approximate relations between variables while omitting details of the computation. In particular, \( y=\texttt{unique}(x_1, \ldots, x_n) \) specifies that the value of \( y \) is uniquely computed (determined) from those of variables \( x_1, \ldots, x_n \); that is, if \( y_1 \) is computed from \((x_1=v_1, \ldots, x_n=v_n)\), \( y_1' \) is computed from \((x_1=v_1', \ldots, x_n=v_n')\), and \((v_1, \ldots, v_n) \neq (v_1', \ldots, v_n')\), then \( y_1 \neq y_1' \). Therefore, if the values of \( x_1, \ldots, x_n \) are different at distinct iterations of a loop surrounding \( y=\texttt{unique}(x_1, \ldots, x_n) \), then the values of \( y \) are
guaranteed to be similarly different. In contrast, \( y=\text{unknown}(x_1, \ldots, x_n) \) specifies that the value of variable \( y \) is computed from reading those of variables \( x_1, \ldots, x_n \), but the relationship could be arbitrary. These special-purpose operators serve to abstract away complex implementation details which degrade the effectiveness of compiler analysis, while keeping essential relations among variables visible to the compiler.

Expressions supported by the annotation language include most of the arithmetic operations in Fortran 77 combined with memory references via scalar and array variables. The Fortran 90 notation of array regions are supported so that collective operations can be applied to arrays without requiring explicit loops. The two special operators, \( \text{unknown} \) and \( \text{unique} \), can also be used directly inside expressions, where their results do not need to be saved in variables before used.

3.1.2 Writing annotations

The annotation language serves to accurately summarize the side effects and loop structures of subroutines without exposing their local implementation details that are irrelevant to the surrounding calling context. In particular, for each subroutine, the annotations aim to summarize relations between its input parameters and output values as well as global variables modified by the subroutine while omitting intermediate results and variables that are local to the subroutine. The goal is to minimize adverse side effects of conventional inlining which may result in accidental loss of parallelism in the inlined code. To demonstrate the capacity of this approach, the following illustrates how to use user-supplied annotations to overcome inefficiencies of automatic parallelization discussed in Section 3.3.

**Avoiding Loss Of Parallelism** As discussed in Section 3.3.1, when conventional inlining breaks procedural boundaries by substituting subroutine invocations with detailed implementations, some parallel loops inside the inlined code may become no longer parallelizable by compilers due to unexpected interactions between the caller and callee. The annotation-based approach resolves this issue by preserving all the original procedural boundaries and thereby entirely eliminating the
Figure 3.2: Annotations for the opaque subroutine defined in Figure 3.13

adverse side effects of conventional inlining. In particular, after we enable the compiler to perform interprocedural dependence analysis by substituting subroutine invocations with user-supplied annotations, the reverse inlining step will replace the inlined annotations with the original subroutine invocations, thereby preserving the original procedural boundaries as well as their optimizations.

Summarizing Opaque Subroutines  Since this approach substitutes subroutine invocations with summaries of their semantics supplied by developers, the approach can be easily applied to opaque subroutines with arbitrarily complex implementations. As example, Figure 3.2 illustrates the annotations for the FSMP subroutine in Figure 3.13, which was excluded from inlining by Polaris due to its excess code complexity. In particular, these annotations summarize the regions of global arrays (FE, ME, SE, MNLE, PE) modified by all the subroutines invoked from FSMP, the temporary global variables (XY, IRECT, K1, K2, K12, ISTRES, NDX, NDY, WTDET) modified in the process, and the read-only global variables (IEGEOM, IECURV, among others) used in the computation. The unknown operator is used extensively to omit local implementation details (e.g., intermediate results) of relevant computation, allowing invocations of the subroutine to be more accurately handled by dependence analysis of their surrounding loops.
Debugging and Error Handling  Many subroutines in large applications contain program output statements used for debugging and error handling purposes. The presence of these exception handling statements are typically treated with extreme caution by compilers, where all reordering transformations of the surrounding loops are subsequently disabled. However, since these statements are used for debugging/error handling only, they are not triggered at runtime in most cases, and even when triggered, precise exception handling is often not required. Using the annotation-based inlining approach, developers can choose to relax the consistency requirement of exception handling when parallelizing their program, by omitting these situations in the subroutine annotations. For example, in Figure 3.2, the error checking conditional at lines 14-17 of Figure 3.13 has been omitted in the annotations. Therefore it no longer prevents loops surrounding invocations of the FSMP subroutine from being safely parallelized.

Use Of Temporary Arrays  Complex subroutines often use temporary arrays to hold intermediate results of the internal computation. When these arrays are declared as local variables, the annotations will omit their existence entirely as they do not incur any visible side effects to the outside. However, sometimes these arrays are declared in the global scope and used to pass values from one subroutine to another. An example global array used for this purpose is shown in Figure 3.15, where the global array XY is modified in subroutine GETCR to hold intermediate results and then used in the subroutine SHAPE1 in Figure 3.16. It is conceptually a temporary array within the FSMP subroutine as only those elements defined in GETCR are used in the subsequent calls to SHAPE1 and other subroutines. Similar global temporary arrays in FSMP include NDX, NDY, WTDDET and P, shown in Figure 3.2. In the annotations, these arrays are modified and used as if they are atomic scalar variables. Since modifications to these variables precede all their uses in the annotations of subroutine FSMP in Figure 3.2, they can be treated as private variables when parallelizing a loop surrounding the invocation of FSMP, shown in Figure 3.14. In particular, when parallelizing the K loop in Figure 3.14, Polaris would peel the last iteration of the loop before parallelizing all the other iterations by privatizing temporary arrays in those iterations, so all the
subroutine ASSEMR(ID,RHSE, RHSI, RHSB) {
  do (IN=1:NNPED)
    do (I=1:NDDF)
      if (unique(ID,IN) == NBLOCK)
        RHSB[unique(ID,IN,I)] += RHSE[I,IN];
      else
        RHSI[unique(ID,IN,I)] += RHSE[I,IN];
  end do
end do
}

Figure 3.3: Annotations for the subroutine in Figure 3.17

global arrays have the same values as their original sequential computation after the entire loop is finished.

**Indirect References In Array Subscripts** Many Fortran applications use global arrays to store dynamic relations between different data structures. Most of these arrays are initialized only once throughout the entire program to save a one-to-one unique mapping between the related data. An example subroutine using these special-purpose arrays is shown in Figure 3.17, where ICOND and IWHERD are global arrays which contain one-to-one relations between elements stored in different places. When they are used as subscripts to access RHSB and RHSI in Figure 3.17, the subscript expressions are non-linear, and compilers have to be overly conservative when parallelizing the surrounding loops. To overcome these difficulties, in Figures 3.3, we use the `unique` operator to summarize the values of these arrays in terms of the relevant input parameters and loop index variables (ID, IN, and I). The declaration of the `unique` relation comes from domain-specific knowledge of the developer. When using the annotations in Figure 3.3 to substitute for the invocation of subroutine ASSEMR in Figure 3.18, each unique operator will be replaced with a linear expression which uniquely combines the involved integer variables ID, IN, and I. As a result the compiler can easily recognize that unique elements of arrays RHSB and RHSI are modified at each distinct iteration of the surrounding loop and thereby can safely parallelize the loop in Figure 3.18.

### 3.2 Enhanced inlining algorithm

Figure 3.4 shows the key steps of the algorithm for applying automatic parallelization with enhanced inlining support. The algorithm is comprised of three main phases: annotation-based inlin-
**Input:** program source code and annotations for selected subroutines  
**Output:** optimized source code  
**Algorithm:**

1. **Annotation-based inlining:** For each call statement within the input program where annotations are provided for the callee
   
   (a) Instantiate the corresponding annotations with actual parameters;
   
   (b) Replace the call statement with the instantiated annotations;
   
   (c) Insert tags surrounding the instantiated code fragment from inlining.

2. **Automatic parallelization:** invoke Polaris to optimize the inlined code.

3. **Reverse inlining:** For each tagged code segment in the optimized source code:

   (a) Find the corresponding subroutine annotations from looking into the tag;
   
   (b) Match the tagged code segment against the corresponding annotations to compute instantiation parameters;
   
   (c) Replace the tagged code segment with a subroutine call using the instantiation parameters.

**Figure 3.4:** Automatic parallelization with annotation-based inlining, automatic parallelization, and reverse inlining. The following explains each phase in detail.

### 3.2.1 Annotation-based Inlining

The implementation of this step is similar to conventional inlining, except that subroutine invocations are substituted with user-supplied annotations instead of detailed implementations of the callees. Translating annotations to the underlying programming language (e.g., Fortran) is trivial, except for the two special purpose operators, `unknown` and `unique`. To translate each `unknown` operator, we define a new uninitialized global array, modify the array with all the operands of the `unknown` operator, and then replace the `unknown` invocation with an access to the new array. To translate each `unique` operator, we replace it with a linear expression which uniquely combines all the relevant integer variables. After inlining, a pair of special tags are placed surrounding the inlined source code to support reverse inlining at a later stage. Figures 3.7 shows the result of applying annotation-based inlining to an invocation of the `MATMLT` subroutine in Figure 3.12, using annotations in Figure 3.5.

In the algorithm, only subroutines with annotations are considered for inlining. Note that although the original implementation of `MATMLT` in Figure 3.11 declares the array parameters $M_1, M_2, M_3$ as having single dimensions, the annotations declare them as two-dimensional ma-
subroutine MATMLT(M1,M2,M3,L,M,N)
  dimension M1[L,M], M2[M,N], M3[L,N];
  M3 = 0.0;
  do (JN=1:N)
    do (JM=1:M)
      M3[:,JN] += M1[:,JM] * M2[JM,JN];
  enddo
end subroutine

Figure 3.5: Annotations for the MATMLT subroutine in Figure 3.11

... !$OMP PARALLEL
  !$OMP+DEFAULT(SHARED)
  !$OMP DO
    DO KS=1,15
      IF(KS.GT.1) THEN
        * //@; BEGIN(Code)
        @annot inline MATMLT {
          !$OMP PARALLEL
          !$OMP+DEFAULT(SHARED)
          !$OMP DO
            DO JL=1,4,1
              DO JN=1,4,1
                TM1(JL,JN)=0.0
              enddo
            enddo
          enddo
          !$OMP END DO NOWAIT
          !$OMP END PARALLEL
        }@*/
        !@
        ENDF
      ENDIF
    enddo
  enddo
  !$OMP END PARALLEL
  !$OMP END PARALLEL

Figure 3.6: Call site of MATMLT in Figure 3.7 after parallelization

  ... !$OMP PARALLEL
  !$OMP+DEFAULT(SHARED)
  !$OMP DO
    DO KS=1,15
      IF(KS.GT.1) THEN
        CALL MATMLT(PP(1,1,KS-1),PHIT(1,1),TM1(1,1),4,4,4)
      ENDIF
    enddo
  enddo
  !$OMP END PARALLEL
  !$OMP END PARALLEL

Figure 3.7: Call site of MATMLT in Figure 3.12 after annotation-based inlining

  ... !$OMP PARALLEL
  !$OMP+DEFAULT(SHARED)
  !$OMP DO
    DO KS=1,15
      IF(KS.GT.1) THEN
        CALL MATMLT(PP(1,1,KS-1),PHIT(1,1),TM1(1,1),4,4,4)
      ENDIF
    enddo
  enddo
  !$OMP END PARALLEL
  !$OMP END PARALLEL

Figure 3.8: Call site of MATMLT in Figure 3.6 after reverse-inlining
trices. Subsequently the annotation-based inlining can avoid the unnecessary linearization of array dimensions which may degrade the precision of compiler analysis, as discussed in Section 3.3.1.

3.2.2 Automatic parallelization

After applying annotation-based inlining to the input program source, we optimize the inlined code using the Polaris compiler (with conventional inlining disabled), which performs advanced dependence analysis of the inlined code and automatically inserts OpenMP directives to parallelize loops when safe and profitable (the profitability is determined based on simplistic heuristics, e.g., all parallelized loop needs to exceed a certain number of iterations). Figure 3.6 shows the result of applying loop parallelization to the inlined code in Figure 3.7. Note that the pair of special tags surrounding the inlined annotations remain intact after the parallelization optimizations.

3.2.3 Reverse-inlining

After automatic parallelization, the reverse-inlining step is performed to reverse the annotation-based inlining transformation while keeping the OpenMP directives inserted by the Polaris compiler intact. This step is necessary to ensure correctness of program optimization. Specifically, while the user-supplied annotations are expected to carry equivalent side effects and dependence constraints as the real subroutine implementation, they are typically not semantically equivalent to the original implementation, due to simplification of internal implementation details and the use of the special-purpose summary operations, e.g., the unknown and unique operators. Therefore, the inlined annotations must be reversed back to an equivalent subroutine invocation to guarantee the correctness of the optimized code.

The reverse inlining transformation is applied to all the tagged code segments created by the earlier annotation-based inlining transformation. For each tagged fragment, it first finds the corresponding subroutine annotations and then proceeds to compute an instantiation value for each formal parameter of the subroutine. Specifically, when using these parameter values to instantiate the subroutine annotations, the resulting code must be equivalent to the tagged code segment.
Currently we apply a pattern matching algorithm to compare the tagged code segment and the inlining annotations node by node, while allowing variable substitution, expression reordering, and OpenMP directives inside the tagged segment. Since the only optimization performed by Polaris is the insertion of OpenMP directives, which are simply ignored in the pattern matching process, a set of appropriate parameter values are guaranteed to be found for each tagged code segment. These values are then used as actual parameters to generate a subroutine invocation so that each tagged code segment is replaced with its original function call.

Figure 3.8 shows the resulting code after applying reverse inlining to the parallelized loop in Figure 3.6. Note that while the single optimization applied by Polaris is the insertion of OpenMP directives, the compiler does perform several normalization transformations, e.g., reordering of statements, induction variable substitution, and constant propagation, to the tagged code segments. As a result the reverse inlining transformation cannot simply replace them with the original subroutine calls. The pattern matching algorithm is tolerant of minor modifications to the inlined annotations and can automatically extract the correct actual parameters in subroutine invocation in spite of the normalization transformations.

### 3.2.4 Correctness, efficiency, and generality

The correctness of the enhanced inlining approach depends on the soundness of the user-supplied annotations. Specifically, if the annotations accurately summarize the side-effects and dependence constraints of the subroutines, the automatic parallelization optimization is guaranteed to be safe. Currently, such consistency is not automatically verified, and we use runtime testers to check and verify the correctness of the optimized code. The future work will develop techniques to automatically verify the soundness of user-supplied annotations and to automatically generate inlining annotations when possible.

The compile-time overhead of applying annotation-based inlining is similar to that of conventional inlining except that since user-supplied annotations are expected to be much shorter than detailed implementations, the cost of applying annotation-based inlining is lower, and the inlined
annotations are expected to be much easier to analyze by the compiler. Since all the inlined annotations are reversed back to using the original subroutine invocations, the modularity of the original program is not affected. The cost of applying reverse inlining to each tagged code segment is proportional to the size of the corresponding subroutine annotations, with constant cost associated with tolerating local modifications of the inlined annotations.

The enhanced inlining approach can potentially enable general-purpose compilers to better utilize domain-specific knowledge from developers in supporting more effective inter-procedural optimization of large-scale applications. So far we have used this approach to support only automatic parallelization via OpenMP. When applying pattern-matching to reverse inlined code segments back to appropriate subroutine calls, the reverse inlining transformation can tolerate local modifications to the inlined code such as reordering of expressions, induction variable substitutions, and insertion of OpenMP directives. However, to extend the approach to similarly support other optimizations such as loop blocking and unrolling, it may become much more challenging to reverse the inlined code segments back to appropriate subroutine calls after dramatic modifications to the tagged segments. Therefore to apply this approach more extensively in a general-purpose compiler, a more systematic approach needs to be developed to compute correct instantiation parameters after dramatic modifications to the inlined annotations, which is a subject of the future work.
3.3 Automatic parallelization using Polaris

Polaris is a source-to-source Fortran 77 compiler which supports automatic parallelization of loops based on sophisticated dependence analysis techniques [19]. It uses simple heuristics controlled via command-line options to govern whether each procedure call is inlined before parallelization analysis [54]. The default strategy inlines a procedure call only when the procedure contains no I/O and not many statements ($\leq 150$ by default) and when the invocation is inside a loop nest.

Polaris includes a number of sophisticated dependence analysis techniques which are fairly effective when analyzing regular Fortran DO loops operating on array subscripts that are linear combinations of the surrounding loop index variables. However, it becomes overly conservative when encountering non-linear array subscripts, which could be introduced by the inlining transformation applied before the analysis. The following summarizes the main issues we found to significantly hinder the effectiveness of Polaris loop parallelization analysis when combined with conventional procedure inlining.

3.3.1 Loss of parallelism due to inlining

In languages such as Fortran/C/C++, arrays are treated as pointers into regions of data, and the same data operated by different subroutines can be declared as arrays of different shapes. Further, when optimizing Fortran subroutines, compilers can assume different array parameters are not aliased to each other. When subroutine invocations are inlined, the abstraction layer is broken, and the inlined implementations may become harder to analyze due to excessive code complexity introduced by inlining. As a result, loops that can be automatically parallelized by compilers when inside their respective subroutines may become no longer parallelizable by compilers after inlining, as discussed in the following.
SUBROUTINE PCINIT(X2,Y2,Z2,..)
DIMENSION X2(*),Y2(*),Z2(*)
...
DO 200 N=1,NTYPES
NSP=NSPECI(N)
NS=NSITES(N)
DO 200 J=1,NSP
I=I+1
X2(I)=FX(I)*TSTEP**2/2.D0/DSUMM(N)
Y2(I)=FY(I)*TSTEP**2/2.D0/DSUMM(N)
Z2(I)=FZ(I)*TSTEP**2/2.D0/DSUMM(N)
200 CONTINUE
...

Figure 3.9: A subroutine with automatically parallelizable loops at lines 3 and 6

DIMENSION T(*),S(*),W(*)
COMMON/WINDEX/ IX(99)
...
CALL PCINIT(T(IX(7)),T(IX(8)),T(IX(9)),..)
...

Figure 3.10: A call site of the subroutine in Figure 3.9 (loops in PCINIT become no longer automatically parallelizable after inlining)

Forward Substitution of Non-linear Subscripts

Figures 3.9-3.10 illustrate a situation where non-linear array subscripts are introduced by inlining the invocation of subroutine PCINIT at line 3 of Figure 3.10 with its implementation in Figure 3.9. Here the actual parameters used in the invocation are indirect references pointing to different regions of a global array T. When using these indirect array references to instantiate the formal parameters X2, Y2, and Z2 of PCINIT in Figure 3.9, the array references X2(I), Y2(I), and Z2(I) at lines 8-10 of Figure 3.9 become T(IX(7)+I), T(IX(8)+I) and T(IX(9)+I) respectively. Because the values of IX(7), IX(8), and IX(9) are unknown at compile time, the inlining transformation has created subscripted subscripts (array subscripts that contain additional subscripted array references) which are non-linear and considered non-analyzable by most dependence analysis techniques. As a result, the loops at lines 3 and 6 of Figure 3.9 can no longer be automatically parallelized after inlining, although Polaris dependence analysis can safely parallelize them inside the PCINIT subroutine before inlining.
Figure 3.11: A subroutine with automatically parallelizable loops at lines 22, 23, and 26

Figure 3.12: A call site which invokes the subroutine in Figure 3.11 (loops in MATMLT are no longer automatically parallelizable after inlining)

Linearization of Array Dimensions

Figures 3.11-3.12 illustrate a situation where common arrays operated by two different subroutines are declared with different shapes. In particular, multi-dimensional arrays $PP$, $PHIT$, and $TM1$ are used at line 5 of Figure 3.12 to invoke the subroutine $MATMLT$ defined in Figure 3.11. However, the corresponding formal parameters $M1$, $M2$, and $M3$ are declared as single-dimensional arrays in Figure 3.11. To inline the subroutine invocation, Polaris reconciles the mismatched array declarations by linearizing $PP$, $PHIT$, and $TM1$ in Figure 3.12 into single dimensional arrays without any explicit shape information. After inlining, the compiler can no longer precisely recognize the dependence constraints of the inlined loops. As a result the three loops at lines 22, 23, and 26 of Figure 3.11 can no longer be parallelized after inlining.
3.3.2 Missed opportunities

Conventional inlining substitutes a subroutine invocation with the entire implementation of the callee, where excessive complexity in the callee’s implementation can force compiler optimizations, e.g., automatic loop parallelization, to be overly conservative due to the lack of domain-specific knowledge and runtime information. The following discusses situations where the complexity of subroutine implementations prevents them from being effectively inlined or optimized.

Opaque Compositional Subroutines

Conventional inlining typically leaves out subroutines that make additional non-trivial procedure calls, as inlining a chain of subroutine invocations could result in serious code explosion. For example, the subroutine \textit{FSMP} in Figure 3.13 is excluded from inlining by Polaris as it invokes a fair number of other subroutines. This subroutine serves to initialize a single column of five arrays, FE (lines 12-13), SE (line 18), ME (line 19), MNLE (line 20), and PE (line 23), using a large number of global variables, including both scalar variables and arrays, some of which are modified to hold intermediate results of the internal computation. In spite of the complexity of computation, distinct columns of the five arrays are modified when invoking \textit{FSMP} with different values for \textit{ID} and \textit{IDE}. Figure 3.14 shows an example loop nest which invokes \textit{FSMP} with values for \textit{ID} obtained from a global array \textit{IDBEGS} which returns a unique integer for each given value of \textit{ISS}. After feeding such information to the Polaris compiler via annotations, the compiler is able to tell that distinct values of \textit{ID} and \textit{IDE} are used at different iterations of the inner $K$ loop at line 4. As a result, it can automatically parallelize this loop after annotation-based inlining is applied, discussed in Section 3.1.2.

Debugging and Error Checking

In practical applications, debugging and error checking statements are often used inside subroutines to ensure proper termination of the application when processing erroneous input data. This
SUBROUTINE FSMP(ID, IDE)
  ... CALL GETCR(ID)
  IMAG = IGEOM(ID)
  ICT = IECURV(ID)
  K1 = AK1(ICT)
  K2 = AK2(ICT)
  K12 = AK12(ICT)
  ISTRES = 0
  CALL SHAPE1
  IF (IDEDON(IDE).EQ.0) THEN
    IDEDON(IDE) = 1
    CALL FORMF(FE(1,IDE))
    CALL CHOFAC(FE(1,IDE), NSFE, IERR)
    IF (IERR.NE.0) THEN
      WRITE(6, *) ' F ELEMENT ',IDE,' IS SINGULAR '
      STOP ' F SINGULAR '
    ENDIF
    CALL FORMS(SE(1,IDE))
    CALL FORMM(ME(1,IDE))
    CALL FORMNL(MNLE(1,IDE))
  ENDIF
  CALL GETLD(ID)
  CALL FORMP(PE(1,ID))
RETURN
END

Figure 3.13: A subroutine excluded from inlining by Polaris

DO 35 ISS = 1, NSS
  DO 30 K = 1, NEPSS(ISS)
    ID = IDBEGS(ISS) - 1 + K
    IDE = K
    CALL FSMP(ID, IDE)
  30 CONTINUE
  35 CONTINUE

Figure 3.14: A loop nest invoking the subroutine in Figure 3.13 (the inner K loop at line 4 can be automatically parallelized after annotation-based inlining)
Figure 3.15: Definition of subroutine \textit{GETCR} invoked by \textit{FSMP} in Figure 3.13

Figure 3.16: Definition of subroutine \textit{SHAPE1} invoked by \textit{FSMP} in Figure 3.13

situation is illustrated by lines 14-17 of Figure 3.13, where the whole program would abort if previous evaluation has resulted in logical errors (indicated by the global \textit{IERR} variable). Since debugging and error checking conditionals typically contain program I/O and early termination of the program, conservative compilers need to disable optimizations of the surrounding loops. However, pre-tested input data are often known to not trigger erroneous conditions at runtime, where the error handling statements are never executed. Even when errors do occur, replication of error messages is often acceptable. Such application-specific knowledge can be incorporated in the annotation-based inlining mechanism to support more aggressive loop parallelization, discussed in Section 3.1.2.
Use of Temporary Arrays

Many subroutines use temporary arrays to store intermediate results of computation, where each temporary array is first modified with new values before being used for additional computation. When the whole computation is inside a surrounding loop, compilers can apply array kill analysis to determine whether any value of the array comes from the previous iterations. If the whole array is killed (reinitialized) at each iteration, the temporary array can be privatized (duplicated within each thread) when parallelizing the surrounding loop. However, the array kill analysis may fail when only a subset of the array elements are modified, and those being used later are not obviously covered by the modifications. To illustrate such situations, Figures 3.15 and 3.16 provide two subroutine definitions invoked by the FSMP routine in Figure 3.13. Here a global array $XY$ is used as a temporary array which is modified by the $GETCR$ subroutine in Figure 3.15 and then used by the $SHAPE1$ subroutine in Figure 3.16. Although $GETCR$ modifies only a subset of the elements in $XY$ (specifically, it modifies $XY(1:2,1:NNPED)$, where $NNPED <= ZNNPED$), only those elements being modified by $GETCR$ are used in $SHAPE1$. However, due to the complexity of the multiple conditionals in Figure 3.16, a typical optimizing compiler would fail to discover the coverage relation even after both subroutines are successfully inlined. We resolve this issue by declaring that the whole temporary array is reinitialized via user-supplied annotations, shown in Figure 3.2 and discussed in Section 3.1.2.

Indirect References In Array Subscripts

Due to the lack of knowledge about runtime values of different arrays, conventional loop dependence analysis techniques are overly conservative when array references are used inside the subscripts of accessing other arrays. Figure 3.17 illustrates such an example, where two global arrays, $ICOND$ and $IWHERD$, which serve to save one-to-one relations between data in different arrays, are used to compute the subscripts of modifying two other arrays $RHSH$ and $RHSI$. After forward substitution of variables, the subscripts used to modify $RHSH$ and $RHSI$ at lines 6-13 become
SUBROUTINE ASSEMR(ID, RHSE, RHSI, RHSB)
  ...
DO 40 IN = 1, NNPED
  NODE = ABS(ICOND(IN,ID))
  IBLOCK = IWHERD(NODE,1)
  IREL = IWHERD(NODE,2)
  IF (IBLOCK.EQ.NBLOCK) THEN
    DO 10 I = 1, NDDF
      RHSB(IREL+I-1) = RHSB(IREL+I-1) + RHSE(I,IN)
  10 CONTINUE
  ELSE
    DO 20 I = 1, NDDF
      RHSI(IREL+I-1) = RHSI(IREL+I-1) + RHSE(I,IN)
  20 CONTINUE
  ENDIF
40 CONTINUE
RETURN

Figure 3.17: A subroutine which contains indirect references in array subscripts

DO 10 K = 1, NEPSS(ISS)
  ID = IDBEGS(ISS) + K - 1
  IDE = IESMNO(ID)
  CALL GETEU(ID, XE, X)
  CALL MATMUL(ME(1,IDE), XE, MXE, NDFE, NDFE, 1)
  CALL ASSEMR(ID, MXE, MXI, MXB)
10 CONTINUE

Figure 3.18: A parallelizable loop invoking the subroutine in Figure 3.17

IWHERD(ABS(ICOND(IN,ID), 2) + I - 1, which will always yield different values when given distinct values of IN, ID and I. Therefore different elements of the arrays RHIB and RHSI are modified when given distinct values of IN, ID, and I. However, such application-specific information is not available to the compiler, which must assume both ICOND and IWHERD could have arbitrary unknown values. Consequently the compiler must conservatively assume that arbitrary elements of RHSH and RHSI could be modified and must refrain from optimizing any loop that invokes ASSEMR even if the implementation of ASSEMR has been inlined. Figure 3.17 illustrates such a call site, where the K loop invokes ASSEMR at line 6 with the values of ID uniquely determined by the loop index variable K. We discuss how to enable Polaris to safely parallelize this loop via annotation-based inlining in Section 3.1.2.
### Table 3.1: Summary of the PERFECT benchmarks

<table>
<thead>
<tr>
<th>Applications</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADM</td>
<td>Pseudospectral air pollution simulation</td>
</tr>
<tr>
<td>ARC2D</td>
<td>Two-dimensional fluid solver of Euler equations</td>
</tr>
<tr>
<td>FLO52Q</td>
<td>Transonic inviscid flow past an airfoil</td>
</tr>
<tr>
<td>OCEAN</td>
<td>Two dimensional ocean simulation</td>
</tr>
<tr>
<td>BDNA</td>
<td>Molecular dynamic package for the simulation of nucleic acids</td>
</tr>
<tr>
<td>MDG</td>
<td>Molecular dynamics for the simulation of liquid water</td>
</tr>
<tr>
<td>QCD</td>
<td>Quantum chromodynamics</td>
</tr>
<tr>
<td>TRFD</td>
<td>A kernel simulating a two-electron integral transformation</td>
</tr>
<tr>
<td>DYFESM</td>
<td>Structural dynamics benchmark (finite element)</td>
</tr>
<tr>
<td>MG3D</td>
<td>Depth migration code</td>
</tr>
<tr>
<td>TRACK</td>
<td>Missile tracking</td>
</tr>
</tbody>
</table>

#### 3.4 Experiment results

To evaluate the effectiveness of the enhanced inlining approach when used to enable more aggressive automatic parallelization of loops, we selected 12 applications from the PERFECT benchmark suite [100], summarized in Table 3.1. For each benchmark, we applied both conventional inlining and the enhanced inlining combined with automatic loop parallelization by the Polaris compiler. To measure the amount of parallelism enabled by and the degree of code explosion resulted from inlining, we counted the number of loops being parallelized after optimization and the line numbers of the resulting source code. We then use two multi-core machines, an Intel Macintosh running MacOS 10.5 with two quad-core 3GHz Intel processors (32KB L1 cache per core) and an AMD Opteron running Linux with two dual-core 3GHz AMD Opteron processors (128KB L1 cache per core), to measure the performance of the parallelized code. All benchmarks are compiled using gfortran 4.2.1 on the Intel Mac and iFort 11.1 on the AMD Opteron, using the -O3 optimization flag.

#### 3.4.1 Enhancing automatic loop parallelization

Table 3.2 compares the number of automatically parallelized loops by the Polaris compiler using three different inlining configurations: disable inlining of all subroutines (i.e., no inlining); inline implementations of subroutines with \( \leq 150 \) lines of source code via conventional inlining (the default inlining strategy adopted by Polaris); and annotation-based inlining, where the enhanced inlining approach is used. For each configuration, we counted the number of automatically
Table 3.2: Automatically parallelized loops using different inlining strategies

<table>
<thead>
<tr>
<th>Applications</th>
<th>Total # of loops</th>
<th>No inlining</th>
<th>Conventional inlining</th>
<th>Annotation-based inlining</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># par-loops</td>
<td>code size</td>
<td># par-loops</td>
<td>#par-extra</td>
</tr>
<tr>
<td>ADM</td>
<td>268</td>
<td>179</td>
<td>8261</td>
<td>153</td>
</tr>
<tr>
<td>ACAN</td>
<td>147</td>
<td>30</td>
<td>36774</td>
<td>30</td>
</tr>
<tr>
<td>QCD2</td>
<td>157</td>
<td>102</td>
<td>3498</td>
<td>102</td>
</tr>
<tr>
<td>MDG</td>
<td>52</td>
<td>37</td>
<td>1962</td>
<td>41</td>
</tr>
<tr>
<td>TRACK</td>
<td>87</td>
<td>54</td>
<td>3107</td>
<td>54</td>
</tr>
<tr>
<td>BDNA</td>
<td>219</td>
<td>129</td>
<td>6713</td>
<td>128</td>
</tr>
<tr>
<td>OCEAN</td>
<td>133</td>
<td>106</td>
<td>8607</td>
<td>106</td>
</tr>
<tr>
<td>DYFESM</td>
<td>197</td>
<td>133</td>
<td>4713</td>
<td>134</td>
</tr>
<tr>
<td>MG3D</td>
<td>150</td>
<td>51</td>
<td>22878</td>
<td>51</td>
</tr>
<tr>
<td>ARC2D</td>
<td>208</td>
<td>182</td>
<td>5185</td>
<td>139</td>
</tr>
<tr>
<td>FLO52Q</td>
<td>175</td>
<td>149</td>
<td>3547</td>
<td>149</td>
</tr>
<tr>
<td>TRFD</td>
<td>38</td>
<td>27</td>
<td>1703</td>
<td>14</td>
</tr>
<tr>
<td>Totals</td>
<td>1831</td>
<td>1179</td>
<td>106948</td>
<td>1101</td>
</tr>
</tbody>
</table>

∗Code size is computed as the number of source code lines with all comments removed.

parallelized loops (#par-loops) and the overall code size (the number of source code lines with all comments removed) after optimization by Polaris. Note that when conventional inlining is applied, Polaris could fail to parallelize some loops which were parallelizable when no inlining is applied, as discussed in Section 3.3.1. These loops are categorized as #par-loss in Table 3.2. Inlining may also enable additional loops being parallelized beyond those parallelized using no-inlining, these loops are categorized as #par-extra in Table 3.2. If parallelized, each loop in the original benchmark is counted only once, even when inlining has made multiple copies of the original loop and all copies are subsequently parallelized.

From table 3.2, inlining (including both conventional inlining and annotation-based inlining) is able to improve the effectiveness of automatic parallelization for 6 out of the 12 PERFECT benchmarks. For the other benchmarks, Polaris was not able to identify additional parallelism from loops which contain subroutine calls. Note that we have manually annotated a subset of subroutines from the PERFECT benchmarks based on careful inspection of their implementations. It may be possible to parallelize more loops by annotating additional subroutines. We have verified the correctness of all the automatically parallelized loops via both manual inspection and runtime testing of the parallelized code.

When combined with annotation-based inlining, the Polaris compiler is able to identify 37 additional parallelizable loops in different PERFECT benchmarks when compared with no-inlining.
Most of these loops invoke complex subroutines which in turn invoke other routines, and the complexity of their implementations would overwhelm most state-of-the-art program analysis techniques. Since the annotation-based inlining allows developers to intervene with their application-specific knowledge, we are able to summarize the intended semantics of these subroutines to enable more effective parallelization of their surrounding loops. Examples of such annotations are illustrated in Section ??.

When applying annotation-based inlining, the code explosion problem is avoided entirely, as the reverse inlining step has restored all the original subroutine invocations (the small increase in code size is mostly due to the extra OpenMP directives inserted to parallelize loops).

In contrast, conventional inlining enabled Polaris to parallelize only a small subset (12 out of 37) of the extra parallel loops identified by annotation-based inlining. Additionally, after conventional inlining, Polaris can no longer parallelize 90 loops which were categorized as parallelizable when no inlining is performed, due to issues discussed in Section 3.3.1. After conventional inlining, the code size increased by about 10% even when only small subroutines were inlined. The increase is likely significantly higher when more extensive inlining is applied via the conventional approach.

* Baseline: performance of the original benchmarks with no optimization;

**Figure 3.19**: Performance of automatically parallelized code
3.4.2 Performance of optimized code

Figure 3.19 presents the runtime speedups achieved by the automatically parallelized benchmarks when using different inlining configurations. Note that a majority of the PERFECT benchmarks do not benefit from loop parallelization due to their small input data size, which is a known issue for these benchmarks [94]. To avoid degradation of performance by excessive parallelization of loops, we used empirical performance tuning to disable a selected set of loops from being parallelized if their parallelization incurs a slowdown of the overall execution time. As demonstrated in Figure 3.19, at most 10% performance improvement is achieved by automatic loop parallelization combined with different inlining configurations. Annotation-based inlining is able to achieve the best performance for two benchmarks (ADM and MDG) and has achieved similar performance as other inlining configurations for the other benchmarks.
3.5 Conclusion

The chapter presents an annotation-based approach to integrate higher-level semantics to enhance inter-procedural loop parallelization. First, a study is presented that exposes some serious limitations of conventional inlining when using the Polaris compiler [19] to parallelize a collection of Fortran applications from the PERFECT benchmark suite [18]. Then, a new annotation-based inlining approach is designed to overcome the limitations. The experiment results show that the new approach can eliminate most of the negative impact of conventional inlining while significantly enhancing the effectiveness of automatic loop parallelization across procedural boundaries.
Chapter 4

Analytical Performance Modeling Of Scientific Applications

A key challenge to optimize large-scale science applications is to understand application behavior on conceptual systems at a high level, gain insights into their performance bottlenecks, and discover optimization opportunities for both applications and hardware design. In particular, scientific applications are one of the main driving forces for the design of the next generation large scale systems. With the scale and complexity in both software and hardware reaching an unprecedented level, their co-design is crucial to enhancing application performance on both existing and prospective future computing platforms.

Two key questions about application behavior are: (1) what are the *hot regions* (i.e., the most time-consuming regions of the application), and (2) what are their performance bottlenecks. The selection of those hot spots is largely based on empirical performance study on an existing system. The hot regions of a workload are often determined by its execution flow, defined as the input-dependent run time traversal of the application code. The concept includes two aspects: hot spots and hot paths. A hot spot is a single code region (e.g., a loop or a function) that consumes a large portion of the application run time; a hot path is the subset of the application execution flow that connects all the hot spots.

Profilers such as gprof [51] are often used to find hot regions; however, they are only available on existing, accessible systems. Micro-architecture simulators, however, often take an enormous amount of time to simulate even a single kernel with a reasonable sized data set. Moreover, simulators treat applications as black boxes and are oblivious to workloads’ control flow structure. Implementing a profiling tool over a simulator, however, is a daunting task and may lengthen the simulation time even further.

To quickly gain first-order insights about hot regions and reasons behind their performance bottlenecks, this thesis presents an approach that analytically models the holistic execution flow of an application and then employs parameterized hardware performance models to project appli-
cation performance for varying architecture design configurations. As shown in Figure 1.6, the overall workflow of the approach includes three steps. First, a source-to-source application analysis engine, named skeletonizer, analyzes the input code to generate a code skeleton, a structural description of the application in the form of the SKOPE workload modeling language [85], to depict the application’s control flow, data flow, communication, and computation intensity, together with frequencies of its control-flow branches gathered from profiling the application on a local machine. During this step the analysis engine also automatically profiles the application on a local machine to obtain frequencies of the control-flow branches and incorporates the branch outcome distributions into the code skeleton. Second, based on the resulting code skeleton, the performance analysis engine automatically constructs a static model for the application execution flow, which is then characterized using a hardware performance model parameterized with the appropriate architecture configuration. Finally, the performance projection for potential hot regions and their performance bottlenecks are projected and reported.

This approach essentially uses statistical execution flow models of applications and coarse-grained performance models of hardware, to project application performance on a prospective architecture without requiring any simulation or instrumented execution of the application on the target hardware. The statistical execution models are constructed from the generated skeletons, which is in turn generated by the skeletonizer. By mathematically modeling the repetitive control flow caused by looping, the analysis and projection time does not scale with the input data size or the actual execution time of the application. By modeling the application execution flow, this approach is able to gain “contextual” insights such as the control flow leading to individual hot spots, the relations among hot spots, and sizes of input data associated with each invocation of a hot spot. Moreover, it can project whether computation or memory access is the bottleneck.

This remainder of this chapter is organized as follows. Section 4.1 discusses the semantics of the skeletons, which are used to summarize application’s static computation structure and performance characteristics. Section 4.2 discusses the skeletonizer approach to automatically generate code skeletons. Section 4.3 discusses the execution flow modeling of the application’s
runtime code path using the Bayesian Execution Tree (BET). Section 4.4 discusses the performance projection process to estimate the execution frequencies and execution time for each runtime code paths in BET. Section 4.5 discusses the approach to identify potential hot spots and extract hot paths from BET. The experiment results are given in Section 4.6. At last, the conclusion of this approach is drawn in Section 4.7.
4.1 Application skeleton

Figure 4.1: An example code skeleton, and its corresponding block skeleton tree and bayesian execution tree.

The code skeleton, or skeleton in short, is a uniform description of the computational behavior of a workload, regardless of the original language in which the workload is written. It is implemented in the SKOPE language [85]. A skeleton is agnostic of any system hardware but can infer potential transformations and help users understand how workloads may interact with and adapt to emerging hardware. According to the semantics and the structures in the code skeleton, the backend can explore various transformations, synthesizes performance characteristics of each transformation, and evaluates performance with various types of hardware models.

Figure 4.1(a) shows an example skeleton of two functions, main and foo. The skeleton is represented by a group of hierarchical blocks of statements. The blocks correspond to the code blocks in the original source code. And the statements could be classified into the following four categories: (1) control flow structures, e.g. functions, branches, and loops; (2) performance characteristics, e.g. instruction counts and memory accesses; (3) critical data-flow represented by the computation needed to estimate the runtime code path of the control flow; and (4) runtime hints represented by the runtime-dependent values of input parameters and code path frequencies.
4.1.1 Control flow structures

The control flow structures in the code skeleton are in a superset of those in the source code. The functions, loops, and branches are preserved in the skeleton, while the branch probabilities can be optionally specified. An example branch is shown in Figure 4.1 line 10, where the percentage represents the fall-through probability of the if-branch.

4.1.2 Performance characteristics

Each block in the skeleton could have statements of hardware-agnostic characteristics describing the computation intensity and memory accesses of the basic block, including counts of floating-point and fixed-point instructions and memory loads/stores. These characteristics are selected as they are needed by the roofline hardware performance model to project performance of a workflow. In Figure 4.1 line 8 and line 12, \texttt{ld B[n]} means there is a memory load to array \texttt{B} in the for-loop, and \texttt{xp 100} means there are 100 fixed-point instructions in the if-branch.

4.1.3 Critical data-flow

The critical data-flow is represented by the computation needed to estimate the runtime code path of the control flow. These computation include the expressions of loop controls, branch probabilities controls, parameters of function calls, and array sizes, as well as all other computation that are needed to compute the above expressions. Additionally, the evaluation of the critical computation will not contribute to the computation intensity or memory accesses, which has been described by hardware-agnostic characteristics. In the example Figure 4.1 line 16 and line 17, the assignments to the \texttt{knob} variable are considered critical, because they will determine the parameters to invoke \texttt{foo} at line 20.
4.1.4 Runtime hints

The runtime hints are represented by the values assigned to the variables in the expressions of the skeleton. The hints are used to specify runtime-dependent values of input parameters, input data sizes, fall-through branch probabilities, and loop controls, which might not exist in the original source code. The values of those hints can be filled in by automatic profilers, or performance engineers, if desired. Some example hints are shown in Figure 4.1 line 1 and line 2, where the values of $M$ and $N$ are the sizes of the input data to the application.
4.2  Skeleton generation

4.2.1  Design principles

As shown in Figure 1.4, the objective of my skeletonizer is to automatically extract code skeletons from the source code and sample input data by performing the following tasks:

- Completeness: Preserving major control flow structures in the application to guarantee the correctness of the skeletons when the runtime code paths are unknown, the generated skeletons need to keep.

- Conciseness: Removing computation details that do not affect computing the runtime control flow and performance characteristics. For large applications, this will reduce the computation complexity for later evaluating the skeletons to construct the execution flow.

- Hardware portability: Calculating hardware-agnostic performance characteristics of computation intensity and memory access statistics needed by the roofline model so that the skeletons don’t have to be regenerated after the target hardware configuration changes.

- Language portability: Unifying array accesses and memory layout for both C and Fortran languages.

- Runtime extensibility: Generating runtime hints to integrate input-dependent code path profiles as the runtime-dependent input parameters and data sizes are absence in application’s the source code.

4.2.2  Generating control flow structures

The skeletonizer traverses the IR of the application’s source code to generate skeletons for all functions, loops, and branches. The line numbers of these structures in the source code are saved during the generation process to allow to automatically integrating runtime hints later.
4.2.3 Computing performance characteristics

For each expression in the source code, the skeletonizer counts the instruction numbers from the operations in the expression and estimates the memory accesses from the array accesses.

The skeletoner categorizes the instructions in the source code into floating-point instructions \((\text{FP})\) and fixed-point instructions \((\text{XP})\). All instructions that do not take floating-point numbers as operands are approximated to fixed-point instructions. This could increase the inaccuracy of the skeleton because, for example, the instruction latency of multiplication is much larger than that of addition. Currently, the roofline model only considers \(\text{FP}\) and \(\text{XP}\) two kinds of instructions. The accuracy of the model could be further improved by adding more instruction categories basing on their latencies.

Memory loads and stores are identified by examining all array access expressions in the source code. The scalar variables are not considered to have loads and stores, because the roofline model assumes the hardware having infinite number of registers that will never spill. Additionally, if the same memory location is read or written more than one times, only one memory load or store will be issued, because the time to access caches are ignored in the roofline model.

4.2.4 Removing computation details

The skeletonizer selectively generates skeletons for the expressions basing on whether the expressions are critical to the execution flow and performance modeling. The generation of the expression skeletons includes two steps: (1) find the set of critical expressions using dependence analysis; and (2) skeletonize the expressions in the set.

To find the set of critical expressions, the skeletonizer first create an initial set of expressions appearing in loop controls, branch conditions, array sizes, and function parameters. And then, using dependence hoisting [121], all expressions that the existing critical expressions depend on are added to the set.

For each critical expression, the skeletonizer will generate SKOPE statements from the SSA of
the expression. The SSA form is needed, which allows us to later override the values of macros easier using runtime hints.

4.2.5 Generating runtime hints

There are two kinds of runtime hints: input parameters and input-dependent branch fall-through probabilities. The input parameters can be extracted directly from the input data. The fall-through probabilities of the branches are collected by profile the source and input data using \texttt{gcov} coverage tools. The branch frequencies in \texttt{gcov}'s results can be automatically matched with the branch in the skeleton basing on the source code line number.

4.2.6 Modeling library calls

Library functions present a challenge for model-based performance analysis. Although their source code may not be available, they may take a significant amount of time to compute and thus should be considered as hot spots. It is difficult to analytically model the performance of library functions without knowing their source code, as their control flows can be input-dependent, and their underlying instruction latencies can be hardware-dependent.

Library functions are modeled in a semi-analytical manner. In particular, it assumes that for the same input, the numbers of dynamic instructions stay mostly the same across different hardware. Hardware counters are used to empirically obtain the mixture of dynamic instructions through profiling on a local architecture and then use the information as input to our analytical roofline model to project their performance on the target architecture. For situations where the number of dynamic instructions vary when operating on different inputs, we randomly generate a sufficient number of input instances, profile dynamic instructions for each instance, and average the statistics to obtain the average mixtures of dynamic instructions.
4.2.7 Processing Fortran arrays

The semantics of array accesses and memory layout of Fortran is different from C language. When processing Fortran arrays, the column-major order is converted to row-major order. Additionally, the skeletonizer generates SSE instructions for Fortran vector operations.

4.2.8 Generality and accuracy

To automatically generate concise and portable skeletons, the skeletonizer trade-off the accuracy of the performance characteristics in the skeleton. The counted performance characteristics of the workflow might be slightly deviated from the actual code because of the absence of compiler optimizations, simplification of instruction cycles, and the obliviousness of registers and caches.

Absence of compiler optimizations

The compiler optimizations, especially loop optimizations, could significantly improve performance for memory-bound applications.

Approximating instruction counts

In the SKOPE language, instructions are categorized as either fixed-point or floating-point instruction, which, however, could have various instruction latencies. For example, integer division will take much more cycles than addition. Additionally, the number of cycles for the same instruction could vary depending on the operands. The information of the runtime operations is lost in the generated skeleton.

Approximating memory accesses

Cache locality and register spilling are not considered while counting memory loads and stores.
4.3 Execution flow modeling

As shown in Figure 1.4, after obtaining the code skeleton of an input application, My framework invokes an execution flow modeling component to reason about its expected run time control flow behavior, e.g., probabilistic of branching outcomes, iteration counts of loops, and dependence constraints among the operations. The key challenge is to meet the following requirements simultaneously.

1. Critical information needs to be preserved so that the user can gain an understanding of the high level control flow semantics.

2. Statistics of computation and memory operations need to be recorded to offer performance insights.

3. Constructing and analyzing the model need to take a minimal amount of time that is asymptotically independent of the input data size of the application.

Conventional representation of the program execution flow, e.g., using a program trace obtained from simulation [119], cannot meet the above requirements due to the difficulty of extracting structural or semantic information from the trace and due to its undesirable property of scaling at least linearly with the size of data being operated. Similarly, call graphs [84] are not a good fit as they only capture function-level interactions without performance details within the individual functions.

I present a new data structure, Bayesian Execution Tree (BET), to statically model the execution flow of programs based on their input data combined with various run time statistics. The key insights are:

1. The execution flow is often repetitive in nature due to loops. By keeping track of the loop iteration counts without repetitively evaluating the loop, the modeling and analysis overhead can be made independent of the data size.
2. While branches may affect subsequent control flow, their effects can be statistically captured based on branch outcome specifications.

3. Performance characteristics are often dependent on only a small collection of variables (e.g., branch conditions, loop boundaries), which can often be analyzed statically given the input data. Even for branches that depend on dynamically generated values, their fall-through probability can be obtained from the knowledge provided by the user or a local profiler when constructing the code skeleton.

The following first presents my Bayesian Execution Tree (BET) representation of the program execution flow and then describes my algorithm for automatically constructing the BST from an input code skeleton.

4.3.1 Bayesian Execution Tree

Figure 4.1(c) illustrates an example BET corresponding to the workload in Figure 4.1(a). A BET node refers to the dynamic execution of a code block with a given context. Each BET node is associated with a conditional probability describing the chance of reaching it given the execution of its parent node. For example, Line 14 in Figure 4.1(a) is a branch whose outcome would affect the branch at Line 25. This branch leads to two contexts with different values of knob, each is associated with a different probability and each creates a BET node corresponding to the function call for foo (rightmost nodes in Figure 4.1(c)).

In contrast to the BST (Block Skeleton Tree) in Figure 4.1(b), the BET contains the knowledge of the run time values of a number of input variables and the sizes of varying data being operated. It conceptually traverses the BST to “mimic” the run time execution starting from the BST node that corresponds to the main function. The BSTs of different functions are then connected based on the input-dependent execution, where the BST of the callee is mounted as a child within the caller’s BST upon each functional invocation.

The BET essentially models the dynamic execution flow by recording the sequence of branches,
loops, and functional calls, as well as their hierarchical structure. Since the control flow often depends on the branch outcome, it computes which branch paths may take place and what are their probabilities according to the input data. To do so, it tracks the probability distribution of context values, defined as the set of variables that would affect branch outcomes, loop boundaries, and data accesses. As a result, the BET models each invocation of a code block along with the invocation’s probability and context.

4.3.2 Constructing the Bayesian Execution Tree

To construct the BET, it first builds the initial context with the values of input variables of array dimensions. This context accounts for 100% probability. It then starts from the main function and traverse its BST. A BET node is created upon every encountered BST node for each current context. The algorithm traverses the BST in pre-order and recursively creates BET nodes.

At each function call, the entire BST of the callee is copied and mounted in place, with the value of arguments set according to the current context. BST copies of the same function can be mounted to different places wherever it is being called; each invocation would have a different context. After being mounted, the traversal continues into the mounted BST.

Upon a loop statement, a single BET node is created. It merely populates the values of the loop boundaries using variables in the current context without iterating over the loop body. Iterations with different control flows are represented as different child nodes with their corresponding probabilities.

Upon conditional branches and switch statements, the conditional probability of a BET is set to the probability distribution of its context multiplied by the probability of the associated branch outcome. Branching statements may also spawn more contexts if variables are assigned different values according to the branch outcome.

Return statements are modeled by setting the probability of the associated context to zero, so that the following statements are no longer executed with this context. The zero-probability context is also promoted to ancestor BET nodes, until it reaches a BET node that corresponds to a
function call.

Continue statements are modeled in a similar way as a branch, except that their probabilities are promoted to ancestors until a BET node corresponding to a loop statement is encountered. Break statements is similar to continue statements and it further modifies the number of loop iterations. To model its effect, the conditional probability of executing the break statement is promoted to the closest ancestor BET node that corresponds to a loop. The expected number of loop iterations is then calculated by $\frac{1-(1-p)^n}{p}$, where $p$ is the conditional probability of break and $n$ is the size of the loop’s range. When $p = 0$ (the loop never breaks), the expected iteration count is $n$.

The resulting BET corresponds to the execution flow with all functions inlined and all performance-sensitive variables evaluated according to the input. Note that this traversal is extremely lightweight compared to even functional emulation of the original program: loops are not iterated since it is treated merely as a single node in the BST; no computation is performed to produce data values except for those variables preserved in the code skeleton, which usually have to do with data-dependent control flow.

Due to the branching of dynamic contexts, the number of nodes in the BET can be significantly larger than that of BST. In the worst case scenario, every branch creates two different contexts, each leading to a different subsequent execution flow. In this case, the BET may be $2^B$ larger than the BST, where $B$ is the number of static branch instructions in the code skeleton. However, this only occurs when the workload is a chain of branches with independent branch conditions; according to our experience, workloads often exhibits nested structures, and the branch outcomes are often correlated. In any case, the size of the BET does not grow with the input size. For all the benchmarks, the size of the BET averages at 88% of that of the source code statements, and it never exceeds a factor of two.
4.4 Performance projection

4.4.1 The roofline performance model

The roofline model is selected to estimate the performance for each code block because of the simplicity and the speed of the model. The roofline model [115] estimates the application’s performance potential based on its operational intensity (i.e., the ratio between the number of floating point operations and the number of bytes of data moved to and from the memory). Given the maximum peak flop rate and the maximum memory bandwidth of the target platform, the operational intensity can be used to derive whether the workload is computation-bound or memory bandwidth-bound and then estimate performance projection accordingly. The roofline model is typically used for estimating macro-level, first-order performance. Due to its simplicity, it is being increasingly adopted in application performance modeling.

4.4.2 Modeling code block performance

Given a BET as input, we first characterize the performance metrics of each code block in a bottom-up fashion. Each BET node that corresponds to a loop, a branch, or a function defines a code block, and its immediate children nodes form statements within the block. For each code block, we scan its statements and aggregate metrics including floating point operation count, fixed point operation count, number of loads and stores, and size of data types. Note that the number of loads and stores are not the actual number of DRAM accesses; they only roughly tell the number of data elements needed for computation without taking into account caching effects.

Once performance characteristics are collected, they are used as inputs to a roofline model to project the performance of the corresponding code blocks. To reflect the target hardware, the roofline model is parameterized with key hardware parameters such as the peak flop rate, frequency, instruction latency, issue width, vector width, shared cache access latency, memory latency, and peak memory bandwidth. The model computes (1) the time needed to process the given number of operations ($T_c$) and (2) the time needed to transfer the required amount of data ($T_m$) and
then outputs the maximum of the two assuming there is perfect overlapping between computation and memory accesses. As a first-order estimation, we assume a constant cache miss rate.\footnote{The cache miss rates for both L1 and LLC are set to 85%; our performance study during daily operation indicates that most workloads’ cache miss rate fall between 75\% and 95\%. This constant is not tuned specifically for benchmarks presented in this paper.}

To estimate the actual run time instead of the asymptotic performance bound, we have extended the default roofline model by considering the possibility that there may not be enough computation to overlap with memory accesses, especially for blocks with a small number of operations. Assuming \( T_o \) is the amount of time that the computation overlaps with memory accesses, the projected performance for a code block is \( T = T_c + T_m - T_o \). We compute \( T_o \) by \( \min(T_c, T_m) \times \hat{o} \), where \( \hat{o} \) is the degree of overlapping. Based on the heuristic that the chance of computation and memory overlapping increases with the number of floating point operations in the code, we compute \( \hat{o} \) as \( 1 - \frac{1}{\text{Num}_{\text{fp.ops}}} \), where \( \text{Num}_{\text{fp.ops}} \) is the number of floating point operations. While this is only a rough approximation of the hardware’s capability, we find that such level of accuracy is able to identify the correct hot spots in most cases.

After using the roofline model to estimate the execution time of a single instance of each code block, the total amount of time spent on the BET node is computed as \( T \times \text{ENR} \), where \( T \) is the estimated time for one invocation of the code block, and \( \text{ENR} \) is its expected number of repetitions. \( \text{ENR} \) is further computed as \( \text{num}_{\text{iters}} \times \text{prob} \times \text{ENR}_{\text{parent}} \), where \( \text{num}_{\text{iters}} \) is the number of iterations if the corresponding block is a loop, \( \text{prob} \) is the conditional probability associated with the block, and \( \text{ENR}_{\text{parent}} \) is the \( \text{ENR} \) of the parent, whose value is 1 for the main function (the root node).
4.5 Hot region analysis

The hot region analysis takes the BET as the input and produces two outputs: the hot spots and the hot paths. Hot spots are small code blocks that consume a significant amount of time and thus are potential performance bottlenecks. Hot paths depict where the hot spots are invoked during the execution flow and how they connect to each other. In particular, the same hot spot may be reachable from several control flow paths, with each invocation operating within a different runtime context and taking a different amount of time. A hot path is conceptually a stripped down version of the workload with only hot spots and the control flows that lead to them. Hot paths can also be used for constructing miniapplications.

There are three steps in the hot region analysis: per-block performance estimation, hot spot identification, and hot path construction. And per-block performance estimation is based on the roofline model. The following describes the roofline model and each of the three steps in detail.

![Diagram](image)

**Figure 4.2:** Building the hot path for Figure 4.1.

4.5.1 Identifying hot spots

The definition of hot spots is often subjective and needs to be flexible to satisfy varying requirements. We offer two criteria that a user can use to configure hot spot selection based on his/her individual needs:
1. Time coverage, defined as the minimal percentage of a workload’s execution time spent in the hot spots. Together, the hot spots should consume a significant portion of the total run time.

2. Code leanness, defined as the maximal percentage of the total number of instructions that should fall into the identified hot spots. The hot spots should contain a relatively small amount of code compared to the entire application.

The above two criteria cannot be simply replaced with instruction intensity, which is the average amount of time spent per static instruction, because a code block with relatively lower intensity may have more run time coverage if it has much more instructions. The code leanness criteria take precedence of the time coverage criteria. In particular, if no code block can be selected to satisfy both criteria, we maximize the time coverage under the constraint of the code leanness criteria. Overall, the problem is similar to the knap-sack problem and is NP complete. We solve it using a greedy algorithm.

Since several BET nodes may refer to different invocations of the same basic block, we first sum up the estimated performance for BET nodes that correspond to the same code block. This projects the total amount of time spent for each block. We then determine the hot spot selection that meet the aforementioned constraints, by first sorting all the blocks based on their projected total execution time and then picking the top-ranking blocks until the constraints are met.

4.5.2 Extracting hot paths

A hot path summarizes the hierarchical sequence of function calls, loops, and branches, that eventually lead to the hot spots. Since each hot spot corresponds to a BET node, we can obtain the control flow path leading to it by simply back-tracing the BET node’s parent until we reach the root node (the main function). This step produces a path for a single hot spot. Figure 4.2(a) shows the individual paths leading to each hot spot in the example of Figure 4.1. We then “merge” the hot paths for all hot spots by scanning all the obtained paths, starting from the root node. Shared
nodes and edges are included in the hot path, distinct nodes and edges become branches in the hot path. Figure 4.2(b) shows the resulting hot path that connects the hot spots in Figure 4.2(a).

Since the BET keeps track of the context values of each code block, the derived hot path, which is a subset of the BET, includes the contexts of each node such as the number of iterations, its branching probability, and the data sizes involved. Such information is helpful for application developers and performance engineers to track down the algorithmic causes of the performance bottlenecks. The hot path also depicts the execution order of the hot spots and thus can help performance engineers analyze the data flow and catch interactions among the hot spots. Such information can also be passed to a compiler to enable path-based optimizations.
Table 4.1: The hot spot selection quality, defined as the run time coverage of the first $N$ projected hot spots compared to that of the first $N$ profiled hot spots. $Q.\text{Modl}$ and $X.\text{Modl}$ refer to hot spot selections for BG/Q and Xeon based on our modeling, respectively. $Q.\text{Xeon}$ refers to hot spot selection for BG/Q based on Xeon profiling. $X.\text{BG/Q}$ refers to hot spot selection for Xeon based on BG/Q profiling.

4.6 Experimental evaluation

This section uses the results generated by profilers on BG/Q and Xeon to verify our model-based hot-region analysis. For each application and each machine, we collected two sets of data. The first set, $\text{Prof}$, refers to the hot spots identified using the profilers, with the sequence of hot spots ranked in descending order according to their profiled runtime. The second set, $\text{Modl}$, refers to the hot spots obtained from my model-based analysis and ranked in descending order according to their projected runtime. To further investigate how portable the hot spot selection is across different hardware, we also attempted to select the hot spots on each machine according to the runtime profile over the other machine.

For each set of hot spots in $\text{Modl}$, $\text{Modl}(p)$ refers to the projected runtime coverage of the hot spot selection suggested by my performance model, while $\text{Modl}(m)$ refers to the actual runtime coverage of the same hot spot selection measured by executing the application on the actual hardware. For all experiments, we set the hot spot selection criteria so that code leanness is less than 10% of the instructions and time coverage is more than 90% of the total runtime. A hot spot selection of size $N$ would compare the first $N$ hot spots in $\text{Prof}$ with the first $N$ hot spots in $\text{Modl}$. Among my benchmarks, SORD is a full application and is the most complex. It is therefore discussed in detail.
Table 4.2: Differences between the projected hot spot selection and the measured hot spot selection based on profiling. Zero means the set of \( N \) hot spots equals the top \( N \) hot spots. \texttt{Q.Modl} and \texttt{X.Modl} refer to hot spot selections for BG/Q and Xeon based on our modeling, respectively. \texttt{Q.Xeon} refers to hot spot selection for BG/Q based on Xeon profiling. \texttt{X.BG/Q} refers to hot spot selection for Xeon based on BG/Q profiling.

<table>
<thead>
<tr>
<th></th>
<th>Sizes of hot spot selections</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>SORD Q.(Modl)</td>
<td>0</td>
</tr>
<tr>
<td>SORD Q.(Xeon)</td>
<td>1</td>
</tr>
<tr>
<td>SORD X.(Modl)</td>
<td>1</td>
</tr>
<tr>
<td>SORD X.(BG/Q)</td>
<td>1</td>
</tr>
<tr>
<td>CFD Q.(Modl)</td>
<td>0</td>
</tr>
<tr>
<td>SRAD Q.(Modl)</td>
<td>0</td>
</tr>
<tr>
<td>CHARGEI Q.(Modl)</td>
<td>0</td>
</tr>
<tr>
<td>STASSSUJ Q.(Modl)</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.3: Profiled and projected hot spot selections for SORD on BG/Q. \texttt{Prof.Q} refers to the hot spot selections resulted from profiling on BG/Q and it serves as the baseline. \texttt{Modl.Q(p)} and \texttt{Modl.Q(m)} show the projected and measured runtime coverage on BG/Q for the model-projected hot spot selections, respectively. \texttt{Prof.Q(x)} shows the measured runtime coverage on BG/Q for the hot spot selections obtained from Xeon profiling.

Figure 4.4: Profiled and projected hot spot selections for SORD on Xeon. \texttt{Prof.X} refers to the hot spot selections resulted from profiling on Xeon and it serves as the baseline. \texttt{Modl.X(p)} and \texttt{Modl.X(m)} show the projected and measured runtime coverage on Xeon for the model-projected hot spot selections, respectively. \texttt{Prof.X(q)} shows the measured runtime coverage on Xeon for the hot spot selections obtained from BG/Q profiling.
Figure 4.5: The projected performance breakdown for SORD’s hot spots on BG/Q.

Figure 4.6: The projected performance breakdown for SORD’s hot spots on Xeon.

Figure 4.7: The profiled performance insights for SORD’s hot spots on BG/Q. We show the issue rate and the computation intensity, calculated as the number of completed instructions divided by the number of L1 misses.

4.6.1 SORD: Case study for a full application

Figure 4.3 shows the hot spot selection results for SORD. Prof.Q and Prof.X refer to the runtime coverage of the hot spot selections obtained from profiling on BG/Q and Xeon, respectively. They serve as my baselines for validation. Modl.Q(m) and Modl.Q(p) refer to the measured and projected runtime coverage on BG/Q using the hot spot selections obtained from my models; Modl.X(m) and Modl.X(p) are the corresponding hot spot selections on Xeon. The similarity
Figure 4.8: The SORD hot path for the top 5 hot spots on BG/Q. Note that a hot spot may be invoked from multiple places.

between the $\text{Prof}.Q$ curve and the $\text{Modl}.Q(m)$ curve shows that the measured selection and the projected selection of hot spots have nearly the same coverage for all selection sizes. The $\text{Prof}.Q$ curve has three distinct segments, each with a relatively constant slope – the first three spots, the next three spots, and the rest of them. The slopes of the first two segments are almost identical, which means that the first six hot spots have relatively similar runtime coverage. As Table 4.2 shows, my modeling has correctly identified all six of them, but not in the same order. Actually, the very first hot spot was identified correctly even though it only takes 9.9% and the second spot takes 9.7%! The curves marked $\text{Modl}.Q(p)$ and $\text{Modl}.Q(m)$ show the sensitivity of my hardware model and the accuracy of projected runtime coverage. They present the same selection of hot spots with projected versus measured runtime coverage and have a reasonably good match. Similar observations can be made from $\text{Prof}.X$, $\text{Modl}.X(m)$, and $\text{Modl}.X(p)$, the profiled and measured hot spot selections on Xeon. The selection quality using my model-projected hot spots is above 80% for BG/Q and above 94% for Xeon.

In fact, the hot spot selections on BG/Q ($\text{Prof}.Q$) and Xeon ($\text{Prof}.X$) are consisted of different sets of hot spots; the fist six out of ten hot spots are different between the two machines. They come in different ordering as well. Therefore, the hot spot selection on Xeon is a poor representative for the hot spot selection on BG/Q, and vice versa. As can be observed Figure 4.3,
Prof.\( Q(x) \), the measured selection quality when the Xeon-suggested hot spots are used to represent BG/Q execution, is quite different from \( Prof.\ Q \). The same is true when comparing \( Prof.\ X(q) \) and \( Prof.\ X \). While the hot spot selection is not portable between machines, my model correctly projects the first 10 hot spots for both machines.

Figure 4.9: Profiled and projected hot spot selections for CFD. \( Prof,\ Modl \) refer to the hot spot selections resulted from BG/Q profiling and performance modeling. \( Modl(p) \) and \( Modl(m) \) show the projected and measured runtime coverage on BG/Q using the model-projected hot spots, respectively.

Figure 4.10: Profiled and projected hot spot selections for SRAD. \( Prof,\ Modl \) refer to the hot spot selections resulted from BG/Q profiling and performance modeling. \( Modl(p) \) and \( Modl(m) \) show the projected and measured runtime coverage on BG/Q using the model-projected hot spots, respectively.

An additional advantage of using model-based performance analysis is that the model can provide insights for each hot spot, which are often not available from results aggregated over the entire execution. Figure 4.5 shows the model-projected performance breakdown on BG/Q for each hot spot with regard to the time spent in computation, memory accesses, as well as the time during which computation and memory accesses overlap. While the profiled measurements do not provide such insights directly, Figure 4.7 shows the profiled issue rate and computation intensity, which indicate that for the latter 6 hot spots, the hardware pipeline is stalled often, and there is a
dramatic decrease in the number of instructions per L1 cache miss, which are likely performance bottlenecks. Such observation closely correlates to my projected insights in Figure 4.5. Further, since my model keeps track of the execution context, we are able to additionally provide the input flags and the data sizes for each invocation of the hot spot (not shown in the figures).

We also used modeling to analyze performance bottlenecks for Xeon. Figure 4.6 shows that the performance bottlenecks for the hot spots on Xeon are different from those on BG/Q. In general, there is a significant increase in the percentage of time spent in memory accesses. This is because the Intel Xeon machine has smaller L1 cache and larger memory latency but faster processing speed comparing to BG/Q. In addition, Xeon provides wider SIMD than BG/Q and the compiled binary is highly vectorized by default.

Finally, combining the hot spot selection with my model of the execution flow, we can reconstruct the hot path of SORD which tells how the hot spots are invoked and connected during the execution. The hot paths for BG/Q and Xeon are different because of the difference in hot spot selection. Due to space limitation, we only illustrate the hot path for BG/Q in Figure 4.8. The hot path shows all control flows reaching the hot spots starting from the main function. We can further distinguish different invocations of the same hot spot. Information about how many times a hot spot is repeated, and the probability of each control flow reaching the hot spot, is also revealed. With such information, one can quickly gain a bird-eye view of the application behavior on BG/Q, identify the most important control flow path to optimize, or build a miniapplication accordingly.

![Figure 4.11: Profiled and projected hot spot selections for CHARGEL. Prof, Modl refer to the hot spot selections resulted from BG/Q profiling and performance modeling. Modl (p) and Modl (m) show the projected and measured runtime coverage on BG/Q using the model-projected hot spots, respectively.](image)
4.6.2 Results for mini applications and kernels

For the CFD benchmark, we have successfully identified all top 10 hot spots, as shown in Table 4.1. The runtime coverage for measured and projected selections, presented in Figure 4.9, match reasonably well, and the actual selection quality is better than 80%. Due to space constraints, we only present our projections for the hot spot selection on BG/Q, which can later be used to construct the hot path. When we compare the actual versus projected runtime coverage (the curve $\text{Modl}(m)$), we have identified that the 6th hot spot was significantly underestimated in runtime. My careful investigation has revealed that this hot spot was expected to take less than 3% of total runtime, but it took 15%. The hot spot was performing the calculation of velocity from a given density and momentum, and involved a series of divisions. My hardware model does not currently differentiate the varying kinds of floating point instructions and treat all of them equal. On BG/Q, the division is a relatively expensive operation and it is expanded by a compiler into a sequence of predefined instructions based on a reciprocal estimate instruction and a Newton-type iteration refinement, thereby resulting in a much more computationally intensive workload. Once we have picked the “offending” hot spot, the runtime coverage quickly converged.

For the SRAD benchmark, the top three measured hot spots took 37%, 28%, and 25% of the runtime, respectively. Figure 4.10 shows that my projected hot spot selections have a runtime coverage that is almost identical to that of the measurement-based selections. As seen from Table 4.2, we have switched the order between the second and the third host spots, but because they have a very similar runtime coverage, my hardware model did not differentiate between the two. Note that the first and the third hot spots are standard $\exp$ and $\text{rand}$ functions in the math library, and we are applying my empirical modeling technique introduced in Section ?? and estimate the overhead from these calls which results in a reasonable runtime coverage.

Figure 4.11 shows the runtime coverage of the hot spot selection of the CHARGEI benchmark. As measured, the CHARGEI benchmark presents two dominating hot spots, one accounting for 44% of the runtime and the other for 38% of the runtime. It is shown in Table 4.2 that my model
projects the correct ranking of the hot spots as well as their runtime coverage, albeit inverting the order of spot number 4 and spot number 5. The runtime coverage for these spots are around 3% and very close to each other, which is too small to differentiate by my hardware model.

Profiling reveals that the STASUIJ kernel has the top hot spot taking 68% of the runtime and the second taking 23% of the runtime. As Table 4.2 shows, the model correctly identifies the hot spot selection and ordering. The Prof and Modl(m) curves in Figure 4.12 perfectly overlap. The projected runtime coverage and the execution time for the first hot spot are overestimated by my framework. Further investigation has indicated that this hot spot is the loop which takes an element of a sparse matrix and applies the scaling of the complex vector by this element. The IBM XL Compiler was converting this spot into highly vectorized code, while in the hardware model we do not account for vectorization features.

![Figure 4.12: Profiled and projected hot spot selections for STASSUIJ. Prof, Modl refer to the hot spot selections resulted from BG/Q profiling and performance modeling. Modl(p) and Modl(m) show the projected and measured runtime coverage on BG/Q using the model-projected hot spots, respectively.](image)

4.6.3 Sources of inaccuracy

The inaccuracy in selecting the hot spots is caused by the jittering in the projection error across different code blocks. Such jittering can be caused by different degrees of computation and memory access intensities, for which the roofline model may exhibit different sensitivities. We also assume perfect instruction level parallelism, which may not hold true for different code blocks. Projection error can also be caused by performance interactions among the code blocks due to caching effects. For example, the code for the first hot spot in SORD is almost identical to the code of the fourth
hot spot, yet the latter can reuse data brought in from the memory by the former, therefore taking less amount of execution time.

Inaccuracy may also stem from my approximation of the workload characteristics. The instruction count and instruction mix obtained from the static analysis may not reflect the actual number of instructions in the binary. There are also memory accesses that are not captured by the code skeletons, such as loading and storing stack variables. Other effects that we neglect from the modeling, such as CPU pipeline stalls or register spilling may also add to inaccuracies as well.

### 4.6.4 Estimating instruction counts

<table>
<thead>
<tr>
<th>Loop</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>skeleton</td>
<td>32</td>
<td>9</td>
<td>9</td>
<td>12</td>
<td>2304000000</td>
<td>2304000000</td>
</tr>
<tr>
<td>profile</td>
<td>32</td>
<td>9</td>
<td>9</td>
<td>12</td>
<td>2270047059</td>
<td>2304000000</td>
</tr>
<tr>
<td>error %</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>1.5%</td>
<td>0%</td>
</tr>
</tbody>
</table>

**Table 4.3:** Performance characteristics for SORD’s top three hot spots. The number in the table represents the instruction count of row type generated by the column method. *ld*: mem loads, *st*: mem stores, *fp*: floating-point inst, *xp*: fixed-point inst.

<table>
<thead>
<tr>
<th>hotspot#1</th>
<th>Skeleton</th>
<th>O0</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>O0</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>9</td>
<td>32</td>
<td>32</td>
<td>16</td>
<td>25</td>
<td>104</td>
<td>36</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>st</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fp</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>xp</td>
<td>47</td>
<td>73</td>
<td>73</td>
<td>27</td>
<td>46</td>
<td>214</td>
<td>50</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>total</td>
<td>65</td>
<td>115</td>
<td>115</td>
<td>50</td>
<td>77</td>
<td>222</td>
<td>53</td>
<td>53</td>
<td>53</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hotspot#2</th>
<th>Skeleton</th>
<th>O0</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>O0</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>8</td>
<td>31</td>
<td>31</td>
<td>29</td>
<td>39</td>
<td>95</td>
<td>29</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>st</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fp</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>xp</td>
<td>43</td>
<td>68</td>
<td>68</td>
<td>36</td>
<td>55</td>
<td>193</td>
<td>46</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>total</td>
<td>59</td>
<td>108</td>
<td>108</td>
<td>71</td>
<td>101</td>
<td>200</td>
<td>53</td>
<td>44</td>
<td>44</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hotspot#3</th>
<th>Skeleton</th>
<th>O0</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>O0</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>9</td>
<td>32</td>
<td>32</td>
<td>15</td>
<td>25</td>
<td>104</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>st</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fp</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>xp</td>
<td>47</td>
<td>73</td>
<td>73</td>
<td>21</td>
<td>43</td>
<td>211</td>
<td>44</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>total</td>
<td>65</td>
<td>115</td>
<td>114</td>
<td>42</td>
<td>74</td>
<td>219</td>
<td>52</td>
<td>45</td>
<td>45</td>
</tr>
</tbody>
</table>

**Table 4.4:** Loop counts in the hotpath of SORD’s #1 hotspot

This section uses the assembly code generated by compilers for two architectures to validate the estimated numbers of computation and memory access instructions in the automatically-generated
code skeletons. The assembly code is generated using GCC with -O0 to -O3 optimization levels on two machines, BlueGene/Q of IBM PowerPC and Xeon of Intel x64 architecture. For each hot spot in the SORD application, we counted the numbers of different categories of instructions in the corresponding code block in the generated skeleton and assembly code. The instructions are classified into four categories: \( ld \) i.e. memory loads, \( st \) i.e. memory stores, \( fp \) i.e. floating-point operations, and \( xp \) i.e. fixed-point operations.

The instruction counts for the top 3 hotspots of SORD is shown in able 4.3. The other 7 hotspots have similar counts. The numbers in the Skeleton columns are the instruction counts in Skeleton’s hotspot block. The numbers in BG/Q and Xeon columns are the number of instructions of the hotspot block in the assembled files. Additionally, in the Xeon column, the total number of instructions is smaller than the sum of all numbers above it. This is because on the x64 architecture, memory load/store and fp/integer operation are usually performed in one instruction. Overall, my estimation of memory stores and floating-point operations are usually similar to the numbers of assembly instructions, where the difference of the worse case is about 50% for all 10 hotspots. My estimated total instruction counts are similar to -O2 optimizations with the difference within 20%. We underestimated the number of memory loads, which could be up to 10 times less than the assembly counts. A summary of the major sources of the differences is as follows:

- Combining arithmetic operations. Some of the arithmetic instructions could be combined into a single instruction. For instance, \( fmadds = \text{add} + \text{multiply} \), and \( fnmsub = \text{subtract} + \text{multiply} \). As a result, in SORD, the estimation of floating point operations could be half of the actual assembly instructions.

- Register spilling. My performance model assumes the machine to have infinite number of registers for simplification reason. As a result, the effect of register spilling is not considered while estimating memory accesses. In SORD, the estimation of memory loads are usually much less than the real memory loads. However, the spilled registers are usually in the cache. Additionally, in the Table 4.3, Xeon’s memory loads are much larger than BG/Q’s. Because
BG/Q has doubled number of general registers comparing to Xeon, and hence the impact of register spilling is not as significant as Xeon.

- Computing array addresses. In the generated skeletons, we approximate the number of instructions to compute the index of a $N$-dimension array access to be $N \times p$. In the real compiler, this number could varies from less than $N$ to $N \times 2$ depending on source code and the optimization level. In the highly-optimized code, array indices might be computed using \textit{lea} to combine a list of linear arithmetic operations; while in the less-optimized code, one array index could be computed using one integer addition and one integer multiplication, which doubles the total instructions.

- Unrolling Fortran array vector operations. In the highly-optimized code, the Fortran array vector operations could be unrolled which added branches to the generated assembly code and increased the total number of instructions.

### 4.6.5 Estimating loop execution frequency

In the constructed BET structure, the number of loops is the main source of the inaccuracy of BET, which is estimated by evaluating the \textit{critical data-flow} in BET. To evaluate the correctness of the loop counts in BET, we compared the loop execution count in SORD’s hotpath with that profiled by instrumenting the application. The count of loops in SORD’s #1 hotspot is shown in Table 4.4. For all 10 hot spots, my estimation is within 97% comparing to the profiling results. The differences of the count comes from the arithmetic errors of branch probabilities. The accuracy of the fall-through probabilities reported by gcov is up to 1%. For example, the probability of the parent branch of #5 loop in Table 4.4 is $2/3$. The error rate is $1%/67% = 1.5\%$, which is consistent with the 1.5\% error rate we have in the table.
4.7 Conclusion

This chapter presents an approach to analytically model the application execution flow to gain first-order insights into its hardware-dependent performance characteristics without detailed micro-architecture level simulation. The obtained performance insights include hot spots and their run time coverages, limiting performance factors for each hot spot, and the hot path that connects all hot spots in the execution flow. By capturing the statistical behavior of the application control flow and integrating the estimated characteristics with hardware performance models, the technique is able to statically project and analyze the performance within a few minutes, and the projection time remains invariant to the input data size. The framework is validated on two distinct systems using production codes, including an earthquake simulation application, and showed that the hot spot selection quality averages at 95.8% and is no worse than 80% in all cases. The execution flow modeling is independent of hardware performance models. In this chapter, the roofline model is used to project hardware performance. However, more sophisticated models can be used. In the next Chapter 5, the LogGP communication model [3] is integrated with this approach to estimate communication hot regions.
Chapter 5

Promoting computation-communication overlap for MPI applications

As a proof of concept, this thesis applies our annotation-based inlining and analytical performance modeling framework to optimize large MPI scientific applications on two modern architectures Intel Xeon x86 cluster interconnected using Infiniband, and Intel Xeon x64 cluster interconnected using 1 Gbps Ethernet. The optimization objective is to mitigate network communication latency by enabling better overlapping of communication time with independent local computations. Two challenges are addressed. (1), the selection of computation and communication code regions to overlap, based on the input data sizes, number of processes, and the network topology of the computing platform; and (2), enabling accurate dependence analysis to ensure the safety and profitability of the optimization, by integrating higher-level semantics of the deeply-nested function calls and the expected input data and communication patterns. The annotation-based inlining is used to help reduce the complexity of the deep-nested inlined code as well as providing programming interface for the developer to specify the runtime code path of the inlined functions. The analytical modeling approach was used to (1) integrate hardware model and profiling to estimate communication bottlenecks, as well as (2) perform course-grained whole-program analysis to find small optimization scope. Both approaches are used to integrate domain knowledge and reduce the code size for later fine-grained overlap optimization analysis.

Fig 1.8 shows the optimization workflow for automatically enabling overlapping of communication and computation in large MPI applications and thereby enhancing their performance portability. The workflow contains three key components: (1) the \textit{performance modeling} component, which analyzes the runtime statistics of an MPI application to extract a Bayesian Execution Tree\cite{85} representation of its execution flow, including the frequencies of various runtime code paths and their performance characteristics such as computation intensities, working set sizes, communication characteristics of MPI operations; The computation intensity is represented by the numbers of fixed-point and floating-point instructions. The memory accesses include the allo-
cated memory sizes and numbers of loads and stores. The communication behaviors consist of the MPI functions being invoked, and the runtime parameters of data types and transferred data sizes. and identifies top time-consuming communication and computation hot spots as the MPI function calls having the largest communication cost, or the basic blocks takes the largest computation time. Then, the computation and communication time for each code path is projected from the performance characteristics using Roofline[115] and LogGP[3] models. (2) the CCO analysis component, which identifies hot computation and communication regions that are likely to benefit from the CCO optimization and summarizes the optimization configuration based on profitability and safety analysis of the optimizations; and (3) the CCO optimization and tuning component, which applies the appropriate program transformations by replacing the blocking MPI operations with non-blocking ones, by reordering the computations and communications involved, and and by inserting MPI_Test operations with a frequency determined by empirical tuning of the optimized code.

The optimization analysis could be divided into the following stages:

- Hotspot analysis. It utilizes the performance model and profiling to find the hot communication to optimize. The output is either a single communication statement or a group of decoupled communication. The purpose is to reduce the amount of code to optimize for large applications.

- Analysis of code regions to overlap. It takes the hot communication from the hotspot analysis, and output a code region where the computation in it could be overlapped with the communication. The analysis only does control flow analysis. The portability analysis which varies by different hardware is not applied.

- Annotation-based linining of functions. After the overlap region is selected, the inlining stage use annotation-based inlining to break the procedural boundaries of the scope to reduce the complexity for later optimizations. The annotation is automatically generated to expresses only the control flow and the data dependence while ignoring the computation
details.

- Computation of independent and decomposable workflows. After the inlining stage, there will be a code region where intra-procedural dependence analysis can be applied. This stage will build a dependence group to find the candidate computation that are either independent from the communication, or can be decomposed in the same way as the communication.

- Generation of CCO pragmas The overlap region, hot communication, independent and decomposable workflows from the previous stages are encoded into CCO transformation pragmas inserted to the source code.

The optimization framework takes the application’s source code as well as the profiling results and hardware models, and generates the source code optimized for the target runtime environment. The workflow that can be divided into the following four stages:

1. Hardware calibration to get the parameters for the computation and communication hardware performance models. The hardware performance models are used to analytically estimate computation and communication time for a workflow. The roofline model is used to model computation. And we develop a profiling-based communication model for estimating balanced MPI two-sided and collective communication time.

2. Skeleton-based modeling for course-grained analysis to find hot communication and candidate overlap code regions. First, an analytical execution-flow model will be constructed for the application. Then, combining with hardware models, the hot region analysis will be applied to find hot communication. And finally, an inter-procedural analysis will be applied on the execution-flow model to compute the candidate overlap region that might contain overlappable hot communication and computation.

3. Optimization analysis to compute overlappable computation in the estimated overlap code region. Given the hot communication and the enclosing overlap region, the optimization analysis will first apply annotation-based inlining to erase the procedure boundaries, and
then use data dependence analysis and control flow analysis to find the computation that are either independent from the communication, or can be decomposed to interweave with the communication.

4. Program transformation to generate the optimized code for the target hardware.

Currently the framework takes the source code, input data of the application, and the runtime MPI process number as input, and output optimization hints that summarize the inter-procedural structure of computation and communication to overlap as well as the places to insert MPI_Test. The transformation is then applied manually by first decoupling the blocking communication into non-blocking functions and then reorder them in the parent loop so that the communication in the current iteration can overlap with the computation from previous and next iterations. The frequency of MPI_Test is parameterized and tuned using binary search on the target platform.

The rest of this chapter is organized as follows. Section 5.1 discusses the analytical performance modeling for MPI communication. Section 5.2 discusses the profitability analysis to select hot communication and computation to overlap. Section 5.3 describes the safety analysis with annotation-based inlining to enhance accuracy of dependence analysis. Section 5.4 discusses applied transformation and tuning. Section 5.5 discusses the experiment results to evaluate the approach. At last, conclusion is drawn in section 5.6.
5.1 Analytical performance modeling for MPI application

Identify the hot communication and computation to optimize, analytical performance modeling is extended to construct Bayesian Execution Tree (BET) for MPI applications. One BET will be constructed for single MPI process with specific rank. In my current approach, the BET of the master process whose rank is 0 is used to identify potential performance bottlenecks. During the construction, the number of MPI processes and MPI rank are specified for the return values of MPI_Comm_size() and MPI_Comm_rank(). Additionally, the LogGP communication model is used to estimate the application’s communication cost. Following subsections will introduce the LogGP model and the estimation of communication cost of BET.

5.1.1 The LogGP communication model

The LogGP model [3] can estimate the communication cost of fixed-sized messages through the use of five parameters: maximum communication latency ($L$), overhead ($o$), gap between messages ($g$), gap per byte for long messages ($G$), and the number of processors ($P$). Based on the model, for example, the communication cost for a point-to-point MPI_Send can be estimated by:

$$\text{cost}_{\text{send}} = \alpha + n \cdot \beta$$ (5.1)

Where $n$ is the message size in bytes, $\alpha$ is the startup cost per message, and $\beta$ is the reciprocal of bandwidth, i.e. cost per byte for long messages. The communication cost in the equation is not the same as the actual communication time, which might also include the wait time between processes because of unbalanced communication. If the values of $\alpha$ and $\beta$ are independent from the message size $n$, they can be estimated by profiling MPI_Send and apply linear regression to the communication time and message size.

The LogGP model could also be used to estimate MPI collective communication cost. Below is an example formulation of MPI_Alltoall of MPICH 3.1 [53] (which could be found in MPICH’s...
source code’s comments):

\[
cost_{a2a|n<threshold} = \log P \cdot \alpha + (n/2) \cdot \log P \cdot \beta
\]  
(5.2)

\[
cost_{a2a|n>threshold} = (P - 1) \cdot \alpha + n \cdot \beta
\]  
(5.3)

Where \( P \) is the number of processes. It has different cost formulations for small and large messages. The threshold message size could be found by executing `mpivars` on the target cluster.

Base on the LogGP model, any MPI point-to-point or collective communication cost can be estimated from the number of processors \( (P) \), message size in bytes \( (n) \), startup cost \( (\alpha) \), data movement speed \( (\beta) \), and threshold values on the target cluster \( (mpivars) \), as illustrated in the following equation:

\[
cost_f = f(P, n; \alpha, \beta, mpivars)
\]  
(5.4)

### 5.1.2 Project MPI communication cost

After constructing BET for a MPI process, communication time for the communication node in BET is projected using the LogGP communication model. The performance projection takes three steps:

1. The message size arguments for MPI function calls are computed by evaluating BET.

2. Communication cost is estimated for each MPI function from the computed message sizes, user-specified number of processes, and the type of the MPI function.

3. The communication cost for the BET node of each MPI function calls are is estimated by the communication cost of the function multiplying by the execution frequency of the node.
5.2 Optimization profitability analysis

This section will discuss the profitability analysis for the CCO optimization analysis component in Figure 1.8. After construction the Bayesian Execution Tree (BET), the profitability analysis that identifies the hot computation and communication in BET that are likely to benefit from the CCO optimization. The analysis takes three steps:

1. Identify potential communication hot spots.

2. Select the candidate loops to apply CCO optimization as the enclosing loops for the communication hot spots.

3. Find local computation hot spots for each CCO loops.

After finding the computation and communication hot spots, the hot path [55] of the loop to overlap can be extracted from BET.

Figure 5.1: Sample hot path of NAS FT
5.2.1 Identify hot communication

Given a Bayesian Execution Tree (BET) for a MPI process with the estimated communication time for each MPI function calls, the potential communication hot spots is identified as the MPI function calls that take most of the estimated communication time. The number of communication hot spots to keep is determined by the selection criteria specified by users. In my current approach, the criteria is to keep the top hot spots whose total communication time takes more than 80% of the overall communication time and to keep at most 10 hot spots. An example of NAS FT is shown in Figure 5.1, where the MPI_Alltoall at the bottom is the communication hot spot that takes more than 95% of the overall communication time.

5.2.2 Identify CCO loops

As shown in Figure 1.7, the loop to overlap is the enclosing loop of the communication. After finding the communication hot spots, the candidate loop to apply CCO optimization is identified as the first enclosing loop outside the each hot communication in the Bayesian Execution Tree. The loop can be in the same or different functions of the communications to overlap. An example is shown in Figure 5.1, where the loop to overlap for MPI_Alltoall() is the loop immediately below the main() function.

5.2.3 Identify hot computation

Given a CCO loop, the local computation hot spots in the loop, which takes most of the computation time of loop, is selected as the candidate computation to overlap with the hot communication.
5.3 Optimization safety analysis

Given the loops that containing the computation and communication to overlap, the safety analysis will check if it is safe to reorder them, and what variables are needed to be privatized to enable the overlap transformation. Because the computation and computation to overlap are usually in different functions, the safety analysis becomes a global optimization problem. Additionally, due to the opaque data movement and dynamic code path, data dependence analysis needed by the safety analysis is difficult to be precisely computed with insufficient runtime knowledge. To address the challenge, we combined conventional data dependence analysis for loops with annotation-based inlining to integrate required domain knowledge from developers. Following sections will first discuss the dependence requirement to apply CCO optimization, and then the analysis being applied.

5.3.1 Safety requirement for CCO

As shown in Figure 1.7 for a CCO loop, the CCO loop can be split into separation of computation can communication. After reordering the computation and communication for CCO, three loop iterations $I - 1$, $I$ and $I + 1$ will be interleaved. $After(I + 1)$ will be executed before $Communication(I)$ and $Before(I - 1)$. To guarantee the safety of the reordering, there must be no loop-carried dependence from $After(I + 1)$ to $Communication(I)$ and $Before(I - 1)$.

However, in the applications such as NAS FT, there are often buffer arrays shared by different iterations that could introduce such loop-carried dependence. When that happens, the buffers are needed to be privatized to each loop iteration in order to apply CCO optimization. If the buffers are also used as the output of the loop, the privatized buffers are needed to be copied to the output array. But if the dependence from $After(I - 1)$ to $Before(I + 1)$ cannot be removed by privatizing variables, it is not safe to apply CCO optimization to the loop.
5.3.2 Annotation-based inlining

One of the challenges to apply safety analysis is absence of knowledge of runtime behavior of the application. Before applying dependence analysis, annotation-based inlining [56] is used to integrate domain knowledge from developers in the CCO loop to improve the accuracy of dependence analysis. The runtime information integrated by annotations are as follows:

- Runtime taken branches. For example, NAS FT can solve 0D, 1D or 2D layout problem, that determined by the input parameters. The runtime code path for the three layouts are totally different. Annotation can be used to express the code path of the target layout (1D) to optimize.

- Memory side effects of MPI functions. The source code of MPI functions are not accessible at compile time. Even if it is available, it would be very complicated to extract the memory side effects from the large code base of MPI implementation. As shown in Figure 5.2, annotations can be used to express the memory side effects using array accesses.

- Normalize array dimensions. The same arrays could be expressed in different dimensions in different functions. For example, in NAS FT, the arrays used in the communication functions are 1D dimension, while those used in the computation functions are 3D. Annotation could be used to rewrite all array accesses in 3D dimension.

```c
def MPI_Send(double buf[], int count, datatype, dest, tag, comm) {
    int i;
    for (i = 0; i < count; i+=1)
        _annot_read(double[i]);
}
```

Figure 5.2: Annotation for MPI_Send
5.3.3 Loop dependence analysis

After applying annotation-based inlining, function calls in the CCO loop will be eliminated. Conventional loop dependence analysis can be applied to report the variables that have the loop-carried dependence discussed in the first section. Based on the location of the communication, all the statements in the CCO can be divided into three groups: (1) computation statements before the communication, (2) communication statements, and (3) computation statements after the communication. Then, the variables that have loop-carried dependence from (3) to (1) and (2) can be identified using loop dependence hoisting analysis [121].
5.4 Program transformation and tuning

After modeling the performance of the MPI application and finding the loops with computation and communication to overlap, the program transformation will be applied to generate optimized code like in Figure 1.7b by three steps: first, the loop optimize will be divided into separated computation and communication functions; then, these functions will be reordered to overlap them; and at last, MPI_Test will be inserted according according to the analytical performance model of the MPI application.

5.4.1 Outline statements

Based on the position of the hot MPI functions in the CCO loop, all statements in the loop are outlined into separated computation and communication functions. An example of NAS FT is shown in Figure 1.7a, where the loop is divided into Comm() for the communication to overlap, Before() for the computation statements before the communication, and After() for the computation statements after the communication. The loop index variable $I$ defined as the additional parameter of the outlined functions. If the communication is already unblocking, the corresponding Wait() function outlined into the CCO loop.

5.4.2 Reorder computation and communication

After outlining functions, blocking communication functions are first replaced by non-blocking communication plus a wait functions, and then the computation and communication are reordered as shown in Figure 1.7b so that the communication $I_{comm}(I)$ can overlap computation in previous Before($I - 1$) and next After($I + 1$) iterations.

5.4.3 Insert MPI_Test

MPI_Test will be inserted to the hot computation blocks according the optimization configuration. If MPI_Test is in loops, a condition will be inserted before the MPI_Test as shown in Figure 5.4.
Figure 5.3: Single or multiple computation loops with MPI_Test inserted

to control how many MPI_Test to execute in the loops, or the frequency of MPI_Test. When there are multiple computation loops to insert MPI_Test, multiple frequencies are needed to be inserted. Our experiment results show when MPI_Test is evenly distributed in the computation time, it can achieve the best performance. To make the computation statements in between each pair of MPI_Test to have roughly equal computation time, the MPI_Test frequencies are determined by the estimated computation time for loops using analytical modeling so that given two loops L1 and L2:

\[
time(\text{Computation}_1) \times Freq_1 = \time(\text{Computation}_2) \times Freq_2
\]  

(5.5)

Given the above relation, the frequencies of multiple MPI_Test can be control by a single frequency, which is then tuned using binary search on the target runtime environment, i.e. the frequency is constantly increased/decreased when the total execution time is reduced.
5.5 Experiment results

My optimization approach targets on optimizing pt2pt and collective communication for MPI applications. Therefore, the 7 applications in NAS Parallel Benchmark (NPB) are used to evaluate the efficiency of the optimization. NAS DT (Data Traffic) and EP (Embarrassingly Parallel) are excluded, which cannot benefit from our optimization approach. Additionally, because my approach requires identification of hot computation and communication, we also evaluated the analytical performance modeling for identifying hot MPI functions in NPB. And to verify the optimal distribution of MPI_Test, we designed a micro benchmark to test the performance for different distributions of MPI_Test.

The benchmarks are evaluated on two architectures:

- Intel Xeon x86 cluster located at Argonne National Laboratory (referred as Intel x86) with 2.6GHz processors interconnected using Infiniband. Benchmarks are compiled using ICC/Ifort 13.1.

- Intel Xeon x64 cluster located at the University of Colorado at Colorado Springs (referred as Intel x64) with 3.2GHz processors interconnected using 1 Gbps Ethernet. Benchmarks are compiled using GCC/Gfortran 4.4.7.

On both clusters, MPICH 3.1.1 [53] is used as MPI implementation. Execution time is collected using Tau[104] and instrumented timers. Because my current analytical performance model cannot estimate intra-node MPI communication time, MPI processes are distributed 1 per node so that MPI communication is inter-node.

5.5.1 Distribution of MPI_Test

To evaluate the impact of MPI_Test distribution to the overlapping of computation and communication, we designed a micro benchmark as shown in Figure 5.5. It contains a single loop of MPI_Alltoall, and a loop of computation emulated using nanosleep(). The execution time is shown in Figure 5.5 and Table 5.1. We tried different numbers of MPI_Test when even distributed,
Loop J = 1 ... M
MPI_Ialltoall
Loop I = 1 ... N
If I % Freq == 0
    MPI_Test
    nanosleep
    MPI_Wait

Figure 5.4: The micro benchmark used to test distribution of MPI_Test

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Time (sec)</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>orig</td>
<td>5.424621</td>
<td>original benchmark without overlapping</td>
</tr>
<tr>
<td>orig-comm</td>
<td>2.224972</td>
<td>total communication time of the original benchmark</td>
</tr>
<tr>
<td>orig-comp</td>
<td>3.2</td>
<td>total computation time of the original benchmark</td>
</tr>
<tr>
<td>5-test</td>
<td>3.56385</td>
<td>evenly insert 5 MPI_Test</td>
</tr>
<tr>
<td>8-test</td>
<td>3.229617</td>
<td>evenly insert 8 MPI_Test</td>
</tr>
<tr>
<td>10-test</td>
<td>3.15611</td>
<td>evenly insert 10 MPI_Test</td>
</tr>
<tr>
<td>16-test</td>
<td>3.249664</td>
<td>evenly insert 16 MPI_Test</td>
</tr>
<tr>
<td>100-test</td>
<td>3.590359</td>
<td>evenly insert 100 MPI_Test</td>
</tr>
<tr>
<td>1000-test</td>
<td>17.265191</td>
<td>evenly insert 1000 MPI_Test</td>
</tr>
<tr>
<td>unequal1-10-test</td>
<td>3.939304</td>
<td>unevenly first insert 8 slow MPI_Test, and 2 fast MPI_Test</td>
</tr>
<tr>
<td>unequal2-10-test</td>
<td>3.624152</td>
<td>unevenly first insert 2 slow MPI_Test, and 8 fast MPI_Test</td>
</tr>
</tbody>
</table>

Table 5.1: Micro benchmark of overlapped MPI_Alltoall

Figure 5.5: Micro benchmark of overlapped MPI_Alltoall
Table 5.2: Differences between the projected hot spot selection and the measured hot spot selection based on profiling with 80% threshold for class B data on 4 nodes. Zero means the set of \( N \) hot spots equals the top \( N \) hot spots.

<table>
<thead>
<tr>
<th></th>
<th>Sizes of local hot spot selections</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>FT</td>
<td>0</td>
</tr>
<tr>
<td>IS</td>
<td>0</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>LU</td>
<td>0</td>
</tr>
<tr>
<td>MG</td>
<td>1</td>
</tr>
</tbody>
</table>

which shows the 10 MPI_Test configuration has the best performance, and the execution is monotonically increasing when the number of MPI_Test is increasing or decreasing. This implies the best MPI_Test frequency can be tuned through binary search. Given 10 MPI_Test, we also tried non-evenly distributed cases of `unevell-10-test` and `uneven2-10-test`, where the sleep time between 2 MPI_Test is will be 4 times slower than the sleep time between the other 8 MPI_Test. Both the non-evenly distribution cases are slower than the evenly distributed MPI_Test.

5.5.2 Evaluate communication modeling

To evaluate the accuracy of the communication modeling, we compared the communication hot spots selected using LogGP modeling with those found by profiling for NPB applications using class B input on 4 nodes. We used the top 10 and 80% thresholds for selecting hot spots. The results are shown in Table 5.2, where the modeling approach found the same hot spots as the profiling approach.

Table 5.2 shows the differences between predicted hot spots comparing to the profilers. The selection criteria are to keep the communication hot spots whose total takes more than 80% of the application’s communication time. The table does not include NPB BT and NPB SP whose communication are non-blocking that cannot be profiled, and it does not have the ranks for non-blocking MPI_Irecv in NPB CG and NPB MG. Our modeling work successfully find all correct hot spots.
<table>
<thead>
<tr>
<th>Sizes of local hot spot selections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>FT 0 1 1 1 2 1 0</td>
</tr>
</tbody>
</table>

**Table 5.3**: Differences between the projected hot spot selection and the measured hot spot selection based on profiling. Zero means the set of $N$ hot spots equals the top $N$ hot spots.

**Figure 5.6**: Speech up of NPB with input B class on Intel x86

### 5.5.3 Evaluate overlap optimization

To evaluate the optimization approach, we optimized 7 benchmarks application in NAS Parallel Benchmarks on the x86 cluster with various input data and number of processes. (NAS DT is skipped which is I/O bound. NAS EP’s hot communication is in loops that cannot be optimized using my approach.) The results are shown in Figure 5.6, and 5.7. In the figures, our optimization achieve up to x1.8 speedup.
Figure 5.7: Speech up of NPB with input B class on Intel x64
5.6 Conclusion

This chapter implements the optimization approach that integrates domain knowledge and performance model to enhance overlapping of computation and communication (CCO). The approach addressed the profitability and safety challenges of the CCO optimization. It formulated how to extract the computation and communication loop to overlap based on the hierarchical control flow of the application and the already identified hot spots. The optimization analysis uses annotation-based inlining approach to integrate domain knowledge to enhance the accuracy of the safety analysis. Additionally, it addressed the importance of MPI Test distribution to the overlapping profitability, and developed methods that utilizing performances modeling to prune the tuning space for MPI Test through binary search.
Chapter 6

Conclusion

This thesis aims to optimize large-scale scientific applications for parallel efficiency. It is essentially based on two approaches: first, annotation-based inlining that summarizes higher-level semantics from developers into annotations and integrates them with compiler optimizations; and second, analytical performance modeling that models application’s execution flow and performance metrics, and identifies potential hot regions and performance bottlenecks. The two approaches are then combined to promote the overlapping of computation and communication for MPI scientific applications.

The annotation-based inlining approach first interacts developers to extract the algorithmic-level control-flow and data-flow for each procedures into annotations. The semantics of the annotations are designed after studying the deficiency of conventional inlining to enable inter-procedural optimization. Then, these annotations are integrated into the call sites to optimize by substituting the function calls with the annotation-generated code. The integration process is similar to conventional inlining except that the annotation with higher-level semantics instead of the original function definition is inlined. At last, compiler optimization is applied to the code after applying annotation-based inlining. To evaluate the approach, it is used to enhance inter-procedural loop parallelization using Polaris to PERFECT benchmark.

The analytical performance modeling approach first extract the computation structure and performance characteristics of the application into code skeletons. Then, combining with code coverage profiling results, a Bayesian Execution Tree (BET) is constructed from the code skeletons to model the execution flow of the application. The computation or communication performance can then be projected for each code path in BET by integrating the Roofline and LogGP hardware performance models. At last, potential hot spots of the application can be estimated by traversing BET, and the hot paths to optimize can be extracted from BET. To evaluate the approach, it is used to identify hot spots in 5 scientific applications from the Argonne National Lab.
The MPI optimization framework integrates the annotation-based inlining and analytical performance modeling approaches to enhance the overlapping of computation and communication. In particular, annotation-based inlining is used to address the safety optimization challenge that integrate higher-level semantics from developers to enhance the accuracy of inter-procedural dependence analysis. Analytical performance modeling is used to address the profitability optimization challenge by identifying the computation and communication to overlap that could potentially benefit from the optimization. The performance model is also used to prune the large search space of MPI_Test. To evaluate the overall approach, it is used to optimize NAS Parallel Benchmark by overlapping the computation and communication in the benchmark applications.

6.1 Publication

The work in this thesis is published or in progress in the following papers:

- Enhancing the Role of Inlining in Effective Interprocedural Parallelization (Jichi Guo, Mike Stiles, Qing Yi, Kleanthis Psarris) published in ICPP11 [56]. The paper discussed the annotation-based inlining approach to enhance Identifying inter-procedure parallelizable loops.

- Analytically Modeling Application Execution for Software-Hardware Co-Design (Jichi Guo, Jiayuan Meng, Qing Yi, Vitali Morozov, Kalyan Kumaran) published in IPDPS14 [55]. The paper discussed the analytical modeling approach for identifying computation hot spots and performance bottleneck.

- Analytical Execution Flow Modeling for Identifying Performance Bottlenecks in Scientific Applications, (Jichi Guo, Jiayuan Meng, Qing Yi, Vitali Morozov, Journal), the work has been submitted for Journal. The journal summarized the analytical performance modeling approach and added the work on automatic code skeleton generation.

- Overlapping Communication With Computation in MPI Applications (Jichi Guo, Qing Yi,
Jiayuan Meng, Junchao Zhang, Baraji Pavan), the paper is being written up. The paper is on enhancing overlapping of MPI communication with computation.
BIBLIOGRAPHY


[31] COINS. Conis compiler framework.


[67] Intel. Cilk plus.

[68] Intel. Threading building blocks.


[87] Open64. Open64.


[92] PIPS. Pips.


