A COMPILER-BASED FRAMEWORK FOR AUTOMATIC EXTRACTION OF PROGRAM SKELETONS FOR HARDWARE/SOFTWARE CO-DESIGN

by

AMRUTH RUDRAIAH DAKSHINAMURTHY
B.E. Visvesvaraya Technological University, 2006

A thesis submitted in partial fulfilment of the requirements
for the degree of Master of Science
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2013

Major Professor: Damian Dechev

ABSTRACT

The design of high-performance computing architectures requires performance analysis of large-scale parallel applications to derive various parameters concerning hardware design and software development. The process of performance analysis and benchmarking an application can be done in several ways with varying degrees of fidelity. One of the most cost-effective ways is to do a coarse-grained study of large-scale parallel applications through the use of program skeletons. The concept of a "program skeleton" that we discuss in this paper is an abstracted program that is derived from a larger program where source code that is determined to be irrelevant is removed for the purposes of the skeleton. In this work, we develop a semi-automatic approach for extracting program skeletons based on compiler program analysis. We demonstrate correctness of our skeleton extraction process by comparing details from communication traces, as well as show the performance speedup of using skeletons by running simulations in the SST/macro simulator. Extracting such a program skeleton from a large-scale parallel program requires a substantial amount of manual effort and often introduces human errors. We outline a semi-automatic approach for extracting program skeletons from large-scale parallel applications that reduces cost and eliminates errors inherent in manual approaches. Our skeleton generation approach is based on the use of the extensible and open-source ROSE compiler infrastructure that allows us to perform flow and dependency analysis on larger programs in order to determine what code can be removed from the program to generate a skeleton.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

The design of new computers requires benchmarks and proxy applications that designers can use to evaluate the ability of these new systems to achieve the performance goals of their ultimate end users. Traditional approaches based on generic benchmarks provide a limited view into the workloads that high-performance computing platforms will be faced with in production environments. Ideally, the set of benchmarks available to system designers will include exemplars of specific applications that are critical to the groups purchasing the new machines. Unfortunately, the creation of benchmarks from specific applications is a time-consuming and tedious process. In this paper, we propose a semi-automatic approach to generating these custom benchmarks based on large scale applications through the use of program analysis and code generation.

The creation of application-specific benchmarks or miniature applications ("miniapps") is work intensive as it requires the creator to not only understand the important performance aspects of the application, but how the entire source code base relates to program elements that contribute to that performance aspect. Performance aspects that are commonly of interest in high-performance computer design include message passing patterns, memory traversal footprints, I/O activity, and numerical computation load. Given an arbitrary line of code in a large parallel program, it is difficult to manually discern what role (if any) it plays for a given performance aspect. Modern static analysis and compilation tools make it possible for a machine to aid in answering these kinds of questions, such as "does this line of code affect the value of the payload for an MPI function?"

Our work presents a step towards automating the skeleton generation process. We employ the ROSE compiler framework to gain access to program analysis algorithms and code generation tools to implement a source-to-source translator. Due to the limited information available to the tool from the raw source code, we take a semi-automated approach to generation in which the

user of the skeleton generation tool provides information that cannot easily be obtained through pure static analysis. In this paper we will discuss both the analyses that the prototype skeleton generator uses, as well as the mechanism by which the tool user introduces guidance in an iterative process. Our results presented in Section 6 demonstrate what can be achieved with a small number of iterations through the tool with user guidance.

Skeleton Driven Simulation

It is common to use simulation to predict the performance of systems before they are available. Simulation of a full parallel application would be prohibitively expensive given that most production applications take a significant amount of time to execute on the bare system itself. The efficient utilization of large-scale parallel event simulators such as SST/macro [4] requires that skeleton models of underlying software systems and architectures be created. Implementing such models by abstracting the designs of large-scale parallel applications requires a substantial amount of manual effort and introduces human errors. Our approach reduces both the effort and likelihood of errors in the skeleton by using established algorithms for program dependency analysis and code generation. These skeleton models can then be combined with appropriate models of the software stack and system hardware to generate a wealth of information about execution pattern such as application communication characteristics and network utilization for high-performance computing architectures. This information is useful in understanding the impact of various design decisions concerning hardware or software and will enable co-design practices to be applied to the design of future exascale systems and provide an environment to prototype ideas for future programming models and software infrastructure for these machines.

Our primary target for this work is within the software/hardware co-design community concerned with designing next-generation (and beyond) high-performance computing (HPC) systems. Sim-

ulation tools are used extensively in the design of computing systems, especially in a co-designed way, to overcome the lack of compiler tools, well-defined ISA features, or complete micro-architectural definitions [4].

CHAPTER 2: LITERATURE REVIEW

The work described in this paper focuses on using program analysis to derive program skeletons from large-scale parallel applications. Modeling the behavior of a large-scale application in terms of performance metrics is challenging since it is often executed on new and dynamic enviroments. The process is complicated by the lack of any feedback between application development and machine design [9]. The strategy is to employ the embedded sytem design practices to come up with hardware-software co-design methodology where there is a tighter integration between application development and machine design. This methodology has to be developed through an interative process with an extensive use of simulation tools [9].

A number of approaches have been proposed for implementing an effective hardware-software co-design methodology. All approaches use simulation extensively since it is cost-effective and does not demand high-end hardware to be available as in the case of direct execution of an application [4]. A trace-based approach where the full program is executed in order to understand the program's behavior, suffers from the cost of high execution times. Simulation using program skeletons provides a number of advantages over direct execution and the trace-based approaches when performing coarse-grained studies of large-scale parallel applications. Skeletonizing allows designers to gauge performance without large costs, and allows skeletons to be derived relative to specific performance aspects of the program. These aspects could include usage of MPI calls, IO calls, or the memory access pattern of the code.

To reduce execution times and hardware costs, developers can come up with a proxy for the application that captures an application's control flow and communication pattern [9] [4]. This is due to the fact that the scalability of a large-scale parallel application is largely determined by the communication model it employs. Most of the computations that are part of the program are generally

executed on individual nodes which have little influence on the program's scalability. Eliminating code that forms the computational part significantly reduces execution time of the program [1].

Deriving skeletons from large-scale programs manually requires a significant programmer effort, and is prone to human errors. Some efforts have investigated automatically synthesizing application skeletons from communication traces [10], but this requires extensive trace collecting, and may not capture behavior produced with particular application parameters. It has also been shown that static analysis techniques can be used to identify computations or routines that do not influence the performance of a program significantly [1]. The source code transformation approach shows how to automate the analysis and optimization of parallel scientific applications by combining dynamic runtime information in the form of communication patterns with static information [7].

The system dependence graph (SDG) and a two-phase graph-reachability algorithm on the SDG [2] is an effective to to compute interprocedural slices. The paper addresses the critical problem of correctly accounting for the calling context of a called procedure in interprocedural slicing by having the system dependence graphs to include some data dependence edges that represent transitive dependences due to the effects of procedure calls, in addition to conventional direct-dependence edges [2]. The algorithm proposed computes the slice in 2 phases. The traversal in Phase 1 follows flow edges, control edges, call edges and parameter-in edges by identifying vertices that can reach vertex s, and are either in procedure P itself or in a procedure that calls P.. The traversal in Phase 2 follows flow edges, control edges, call edges and parameter-out edges by identifying vertices that can reach s from called by P or from procedures called by procedures that call P. The result of an interprocedural slice consists of the sets of vertices identified by Phase 1 and Phase 2 and the set of edges induces by this vertex set. This work can employed to derived program skeletons based on the support provided by tools to construct SDG and to carry out data flow analysis.

Our work addresses many of the issues pertaining to skeleton construction by providing a framework that automatically extracts skeletons from applications leveraging static analysis capabilities offered by the ROSE compiler.

CHAPTER 3: KEY COMPONENTS

We begin with a discussion of the key components that make up the core of our automatic skeleton extraction framework. The components are presented in brief along with their functionality.

ROSE Compiler Framework

ROSE is an open source-to-source compiler infrastructure for building a wide variety of customized analysis, optimization and transformation tools [6]. It enables rapid development of source-to-source translators from C, C++, UPC, and Fortran codes to facilitate deep analysis of complex application codes and code transformations. ROSE consists of front-ends for parsing code, a mid-end for code analysis, optimizations and transformations, and back-ends to generate source code [6]. The currently available front-ends are the Edison Design Group (EDG) front-end for C and C++, and the Open Fortran Parser (OFP) front-end for Fortran. Using ROSE program analysis techniques, we have developed an early version of a source-to-source translator to automate the extraction of skeleton applications from their parent applications containing MPI communication patterns.

Program Skeletons

A program skeleton is an abstract form of its parent application where we retain only those portions of the code that are relevant to the performance dimension that the skeleton is intended to probe. In the case of message passing behavior over MPI, the skeleton is intended to be consistent with the original program flow such that it retains the communication patterns of the original application. To achieve this, a skeleton is essentially a sliced version of the original program that is constructed

by removing fragments of redundant or irrelevant computations and message data whose values do not affect application scalability, but retaining those code fragments specific to the set of properties of interest for performance analysis.

Applications that utilize high performance computing systems often implement compute and time intensive computations that form the content of messages passed between parallel processors at runtime, but the message passing pattern that the program exhibits often is independent of the values that these computations produce. This observation that some portion of the computational load is irrelevant when considering performance aspects such as message passing patterns is the basis of our work. However, there do exist a number of complex applications where the message passing pattern is determined by the computations that are performed. Common examples of this include programs that utilize unstructured data in which the decomposition of data across compute elements changes as its state evolves. Addressing programs where dynamic message passing patterns are present driven by the results of the computational part of the program is a topic of ongoing research.

Due to the complexity of large parallel applications, and the limits of purely static analysis for languages like C, C++, and Fortran, we adopt a semi-automated approach where the programmer is involved in the skeleton generation process through program directives and other parameters that guide skeleton generation tools. Once a satisfactory skeleton is created, it can act as a model to enable scalability studies using the SST/macro simulator or other performance analysis and simulation tools to emulate systems under design, such as interconnection networks, for systems containing millions of cores.

SST/macro Discrete Event Simulator

The SST/macro discrete event simulator is an open source simulation package that enables evaluation of large-scale parallel machines [4]. The simulator is implemented in C++ with a modular design, permitting multiple computation and communication models to be employed. SST/macro is extensible by supporting user-defined network topologies, allocation and scheduling strategies, and performance models by externally linking to the simulator core. It can be driven either by direct execution of application code using communication libraries like MPI, HPX, or OpenSHMEM, or through communication traces like the DUMPI format that is included in the distribution. The MPI implementation and network congestion models have been validated against current HPC machines, including a Cray XE6. The simulator enables us to investigate the effects of varying environment parameters such as type and tuning of network topologies, hardware layout (e.g. processors per node) and system parameter choices (e.g. bandwidths, latencies).

CHAPTER 4: APPROACH

Our approach follows an iterative process which performs static slicing of the code with the aid of annotations. The key components of our skeleton generator and how they fit together is shown in figure 4.1. The ROSE compiler serves as the base framework by allowing a user to construct program analysis modules with common forms of program analyses such as dependence analysis, control flow, and call graph etc. The SST/macro simulator provides the enviroment for executing application and its skeleton with model choices provided by the user. The target MPI application annotated with program directives is fed as an input to the ROSE front end. The front end parses the input source code and generates an Abstract Syntax Tree (AST) called Sage III Intermediate Representation (IR). This AST serves as the input to our skeleton generator.

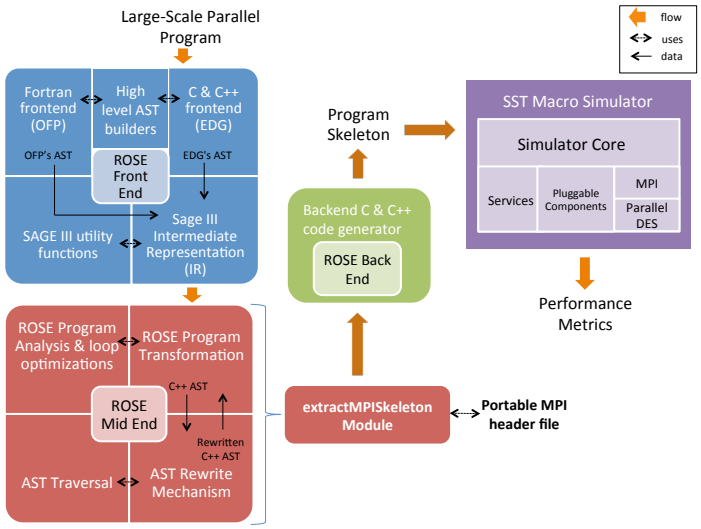

Figure 4.1: extractMPISkeleton Module.

The primary activity performed by the skeleton generator is the identification of code that can be removed based on constraints provided by the tool user. These constraints are externally provided instead of hardwired into the tool because they depend on the performance dimensions that the

skeleton is intended to probe. For example, the set of API function calls preserved for a skeleton representing MPI activity is completely different from one representing disk I/O operations. The constraints that the user specifies are expressed in two forms. First, one or more API specification files that contain information about the API functions that should be preserved are provided. Second, a set of in-source directives in the form of compiler pragmas are available for targeted control over the internal decisions made by the generator.

The skeleton generation tool that we designed has the following overall structure:

- The program code is put into Static Single Assignment (SSA) form, allowing def-use analysis to be performed.
- API calls of interest are identified within the program abstract syntax tree and used to identify code that they depend upon that must be preserved leading up to the calls. This analysis uses the definition-use information obtained via the SSA form.
- Programmer directives are processed to override information obtained by analyzing the def-use chains based solely on the API of interest. These directives include overriding choices to exclude or include code, both to overcome limitations in the current dependency analysis algorithms used and to inject knowledge about the desired skeleton that the programmer has that cannot be inferred from syntax alone.

Our skeleton generator using ROSE adds SSA property to the AST, in which every assignment to a variable creates a new version of the variable. This simplifies the process of definition-use analysis to determine all uses of a variable from its definition without any other intervening definitions. The skeleton generator then implements a form of *program slicing*. Slicing relative to one or more statements uses dependency analysis to determine all program elements that the statements

depend on to execute correctly. Any code that these statements are not dependent upon can be removed. Our skeleton generator implements a specialization of slicing in which we do not treat all dependencies equally. Dependencies are classified by a "role" that they play relative to the API that we slice relative to. In MPI programs, we distinguish dependencies that influence the contents of a message from those that influence the message passing communication pattern. This allows us to transform code based on its role—in some cases, we preserve the dependency code with no change as in traditional slicing. In other cases, we may replace the dependency code with a surrogate that aids in reducing the complexity of the skeleton. For example, we may preserve all code that relates to message passing topology, but will replace code related to the message payloads (that can be shown to not influence the topology) to insert initialization to constant values in place of compute-intensive numerical computations.

API Specification

The skeleton generator skeletonizes programs relative to one or more API specifications. This means that functions that are part of the API are required to be preserved in the skeleton, and further code is preserved based on their dependencies. Each function that we would like to treat as part of the API is specified by the function name, number of parameters, and a list of dependency types for each parameter. The dependency types allow us to categorize code based on how it impacts the API functions. For example, in the case of MPI, we will have code related to the data that is sent (the "payload"), and code that is related to the communication end-points and abstract communicators used within the program. Code that influences these parameters to the API functions is often very different, and the API specification allows the tool to tag program elements based on their role. This information is not currently used in any detail other than distinguising code by role for role-specific removal or preservation. Deeper analysis of code that takes on certain roles (e.g., payload

creation) is the subject of further work. For example, symbolic simulation techniques may be useful for replacing code with a simpler surrogate instead of removing it entirely if the payload data is important in determining the communication pattern.

An example of a small portion of the API specification for MPI is provided below.

```
(api-spec MPI
  ( dep-types payload topology tag other )
  ( default_deptype other )

  (
    (MPI_Init         2 )
    (MPI_Finalize     0 )
    (MPI_Abort        2 )

    (MPI_Comm_rank    2  (topology 1) )
    (MPI_Comm_size    2  (topology 1) )
    (MPI_Comm_split   4 )


    (MPI_Send         6  (payload 0 1 2)
                         (topology 3)
                         (tag 4) )
  )
)
```

Each API is labeled by name via the `api-spec` tag. This is followed by an s-expression that con-

tains a list of dependency types, prefaced with the `dep-types` tag. The names of the dependency types are user-specified, allowing roles of arguments to API functions to be analyzed at different levels of detail. For example, if the `tag` role is not of interest to the skeleton generator, it could be eliminated and absorbed into another category. This flexibility allows the user to tune the generator based on what they know is important about their program and how it uses the API. The `other` category is a special category that is a catch-all for arguments that are considered irrelevant for skeletonization.

Each API call is described by its name, argument count, and an enumeration of arguments by position that are to have roles associated with them. Arguments not listed are by default placed in the `other` category.

```
(API_FUNCTION_NAME ARGUMENT_COUNT (deptype argA ..)
                                  (deptype argB ..)
                                  ...)
```

For example, on `MPI_Send` above, we see that it has 6 arguments, the first three of which are related to payload, the fourth is topology, and the fifth is tag. Note that argument numbering is zero-based. The sixth argument is not specified, and therefore takes on the default dependency type ”other”.

Given a set of API specifications, we then allow an API collection to be specified that is used by the skeleton generator to know what set of APIs to skeletonize relative to, and how to do so. For example:

```
(api-spec-collection
 (include-api "mpi_api.spec" (omit-deps payload))
```

```
 (include-api "stdio_input.spec" (omit-deps buffer))
)
```

This tells the tool to include API specifications for MPI and a subset of C STDIO functions. This is important in practice since skeletons frequently require parameter files to be read that are used to configure the application. Given that this code often is performed outside the MPI API, we must allow the skeleton generator to be aware of additional APIs that are important in generating a legitimate skeleton. For each API, a specification of the dependency type to use for selecting code for elimination is also provided. In the example above (provided for illustrative purposes only), code that relates to computation of payload data for MPI calls is eliminated, and buffer management code is eliminated that relates to the STDIO API. This allows a user to have relatively fine control over what is and is not removed at the API level.

Unfortunately, in non-trivial programs it is not possible to simply discard all code that isn't a call to an MPI library function. Doing so would likely result in a skeleton that is missing critical code that is necessary to ensure that these MPI function calls can execute correctly. For example, removal of code that allocates buffers for message passing calls will likely cause a program to crash due to dereferencing a null pointer. Similarly, code that establishes the message passing topology (such as the ranks used for send and receive calls) must be preserved. If too much code is removed and replaced with, say, a default value of zero being used for the rank, we likely would observe a very different message passing pattern (e.g., rank zero being overwhelmed with messages) than we would see in practice, rendering the skeleton useless as a performance analysis surrogate for the original program.

The general approach that we take is to define a set of functions that form the API that we wish to skeletonize relative to (in this case, the MPI API as defined for MPI-2). For each function, we define a role for each argument that is used to label code that influence their values. For example,

an MPI call often has one or more arguments that relate to the payload and its contents, and other arguments related to the message passing topology. These are important to distinguish, as we may preserve code for some roles exactly, while for other roles we may seek to replace it with lighter weight alternatives. A simple concrete example would be to preserve all code that influences the message passing topology, but replace code that initializes buffers with code that simply populates them with a constant value. Given this API specification, we then use dependency analysis as provided by ROSE to label statements within the program based on the role they play with respect to the API. This is then used to prune the program AST before generating source code representing the reduced skeleton.

Source-level Directives

Annotations used by this tool are specified via compiler directives:

```
#pragma skel [specific pragma text here]
```

For example,

```
#pragma skel loop iterate atmost(10)
```

Annotations are used to provide fine grained control of the skeletonizer at a finer level than that is captured in the API specification. The simplest annotations that we provide allow the user to preserve or remove program elements to override the decision made by the skeletonizer based on dependency analysis. These are placed above the corresponding program element:

```
#pragma skel preserve
x = y + z;
```

Loop Annotations

It is not uncommon for skeletonized code to no longer have the looping behavior of the original due to the removal of computations which provide the values that define a termination criterion. For example, say we have an iterative solver with the following stucture:

```
do {
  // numerical computation
  for (i=0;i<n;i++) {
    prev[i] = current[i];
    current[i] = a_big_computation();
  }

  // compute delta based on computation
  for (i=0;i<n;i++) {
    delta += fabs(current[i] - prev[i]);
  }

  // do some message passing
  MPI_Send(...);
} while (delta > eps);
```

If the numerical computation is removed, we can be in a situation where `delta` will not ever change and will never drop below the threshold for termination—so the loop will iterate forever. Similarly, we may find that through some choice of initial values, the skeleton may initialize `delta` to be zero, so the loop will iterate only once (or never, depending on the type of loop

used). In both cases, we would see behavior that is not representative of the real program. An annotation can be added to force the skeleton to contain a loop that iterates a certain number of times by having the skeleton generator introduce additional counters and code to increment and test their values.

Three loop annotations are available:

```
#pragma skel loop iterate exactly(n)
#pragma skel loop iterate atmost(n)
#pragma skel loop iterate atleast(n)
```

These correspond to forcing an exact, upper, and lower bound on the iteration count. The pragma must be placed immediately preceeding the loop of interest. Loops constructed with 'for', 'while', or 'do while' are all supported as well as loops containing break and continue statements. We treat break statements as an exception that overrides the 'exactly' or 'atleast' annotations. If a break is reached in a loop before the n-th iteration, the loop is exited but a warning is printed to `stderr` indicating the violation of the pragma.

In the above annotations, `n` is a C expression that is interpreted in the current scope of the program.[1] This allows the programmer to specify program variables to be used in the stopping criteria that they are confident will contain meaningful values at skeleton execution.

Insertion of code to implement these loop control pragmas occurs after dependency analysis is performed and relevant code removed. This is implemented as an additional pass over the program AST seeking loop structures that have pragmas associated with them. These are guaranteed to be present if the loop body contains an API call that was invoked due to the presence of a control

[1]Due to a limitation in current versions of ROSE, floating point constants cannot be specified directly within the pragmas. Constants must be expressed as rational numbers, such as 12345/100000 in place of 0.12345.

flow relation between the call and the loop. Code insertion is performed after code is flagged for removal to avoid the inserted loop control logic from itself being flagged for removal.

Data Declaration Annotations

If a program contains an array that should be preserved in the skeleton, it is useful to have control over how it is initialized since often the skeleton will not contain the computational code that populates the array elements. The initializer pragma allows these element values to be specified.

```
#pragma skel initializer repeat(x)
int myArray[14];
```

Where `x` is a C expression interpreted in the current scope of the program.

This will result in code being generated that iterates over the array elements assigning the value `x` to them. The variable initialization annotation supports arrays in the "auto" storage class (but not yet arrays in the "static" storage class). Future versions of the skeletonizer will support initialization of static arrays and dynamically allocated arrays. Future versions will also support initializing non-array variables.

Dependency Analysis

The core static analysis algorithm that we rely on is the computation of the Static Single Assignment (SSA) form of the program. This computation is provided as part of the ROSE framework. Once SSA form has been obtained for a program, use-define chains are available for relating uses of variables to their defining statements. For each call site of an API function to be included in

the skeleton we obtain a set of variables corresponding to the contents of expressions that form the function call arguments. Each argument is tagged with the dependency type that is defined with the API specification, such as “payload” or “topology” in the MPI API discussed above.

The def-use chains for these expressions are then traversed, where each program element leading up to the call is labeled as a dependency along with the dependecy type that is inherited from the API specification. The labeling process takes the union of dependency types that are derived for each program element. This labeling allows the user of the tool to have finer control over the transformation that removes code - instead of preserving all code that an API call depends on, the transformation can preserve based on API-specific roles.

A current limitation of the skeleton generator is full support for whole program dependency propagation in programs containing multiple independently compiled program units. Program units (e.g., C++ source files) are analyzed and transformed independently. A result is that when one C++ file references code implemented in another, we lack information about any dependencies that should be propagated out to the caller. An elegant solution to this problem is to take an approach similar to languages like Fortran and Haskell, where processing of independent source files results in an additional file being created (such as a Fortran .MOD or Haskell .HI file) that contains metadata beyond the basic API types for use when compiling other program units that reference the corresponding file. An alternative approach is to modify our generator to be aware of the full set of source files that a project requires such that they can be loaded and processed at once (in an appropriate order). These engineering additions to our prototype are the subject of our ongoing development work, and do not affect the basic slicing concept at the core of the skeleton generator.

Code Transformation

After dependency analysis has been performed and AST nodes are annotated with their role in the skeleton, we perform a transformation on the AST before final code generation in the language of the original program. For a given program construct, there are three possible choices at transformation time: complete removal, complete preservation, or replacement. For code that is determined to not be in the dependency chain of API functions, removal is the default behavior. This must be overridden by using the `preserve` pragma described above. Code that is in the dependency chain is preserved by default unless the full set of roles associated with it appear in the `omit-deps` parameter of the API specification parameter file.

If a program element is flagged for omission, program statements are removed but variable declarations are preserved. This allows the API call to remain syntactically correct by having its full set of parameters available. A consequence of this is that initialization code may be removed, which can be particularly problematic in the case of variables with pointer types. The `initializer` directive can be used to provide initial values for declarations, as well as the use of the `preserve` to keep whatever amount of setup and initialization code that the user of the tool determines to be necessary. We also have additional directives that can be used by the user to replace conditional tests with probabilistic tests that can be driven by branch probability information obtained by dynamic analysis of the original program (e.g., via trace collection).

CHAPTER 5: EXPERIMENTAL SETUP

We describe the setup required to execute SST/macro simulations using application skeletons. A couple of code moficiations are necessary to enable skeleton-driven simulations of MPI codes. The first change is including SST/macro-specific header files in place of the MPI header files, so that traffic is routed through the simulator and not the host machine. This code modification does not affect the skeleton code since the SST/macro's MPI interface is nearly identical to the original MPI interface [5]. The second change is renaming the *main* function to *user_skeleton_main*, which is called by the simulator after it has started up. The simulator provides a *makefile* which can be used to build the skeleton and generate an executable, linking against the simulator core. In addition, SST/macro provides a configuration file containing a set of basic simulation parameters such as network parameters, node parameters, and application parameters. These parameter choices allow for the tailoring of architecture characteristics and model approximations.

Target Applications

To demonstrate the practical use and accuracy of our approach we studied the following HPC proxy applications (implemented in C):

1. The HPCCG application [8] implementing the conjugate gradient method for solving a linear system.
2. A simple implementation of the Fast Fourier Transform (FFT) algorithm.
3. A 2D Jacobi iteration implementation.

MPI is used in all three applications to implement their communication model. The Jacobi iteration program that solves Laplace's equation is a simple representative of programs that utilize simple stencil-based iterative methods. The HPCCG program solves a more complex system based on a 3D diffusion problem using a 27 point implicit finite difference scheme. The FFT algorithm is a basic implementation in which a data set of size N where N is a power of two is decomposed into $\frac{N}{p}$ element subproblems that are solved on p processors, with collective communication operators used to exchange their results before completing the computation. This set of test cases represents varying communication patterns (both point-to-point and collective), and exhibit different granularities of local computations that are performed between communication operations.

Each program is transformed by our skeleton generation tool after the introduction of annotations to aid the program slicing process. In some cases, preprocessor directives containing macros (`#define`) are replaced with static variables to prevent expansion during the skeleton creation. The original macros are restored before using the skeletons with SST/macro. Each program has been directly executed before skeletonization in SST/macro in order to collect runtime tracing information. We use the trace file execution logs to validate the accuracy of our skeleton-driven simulation approach.

CHAPTER 6: RESULTS

Here we present our experimental validation of the derived program skeletons' correctness against the application from which they are derived. The validation is done by comparing binary traces of both the application and the skeleton. We then explore the scalability of the skeletons under simulation with respect to the original applications via SST/macro.

Validation of Program Skeleton Correctness using DUMPI Traces

Skeleton correctness is of utmost importance in studying the scalability of an application using skeleton-driven approach. In order to use a skeleton in place of an application for scalability studies, the runtime behavior of the skeleton has to match the application's behavior both in terms of control flow and communication pattern.

Our approach relies on deriving this information from a binary trace that captures control flow and communication pattern in sufficient detail. However, the trace might vary according to the set of inputs the program receives and the network model setup on the simulator. This is due to the execution of branch statements that make program behavior depend on the set of inputs provided as well as the intermediate runtime values that the program calculates. As such, for each evaluation on the simulator, the inputs and hardware/network parameters are kept constant for both the application and the skeleton when trace files are generated.

The binary trace we generate is in DUMPI format, a custom MPI trace file format developed as part of the SST/macro simulator. The trace records MPI events with full signature of MPI calls, return values and MPI request information [4]. A trace is collected for each run of the full application and

for the skeleton execution on the simulator. SST/macro also ships with a set of trace analysis tools, which produce an XML file that contain statistical MPI trace comparison information grouped together under different tags. In order to match the skeleton with the application, we compared DUMPI trace reports generated from trace files of both the skeleton and the application.

Tables 6.1, 6.2 and 6.3 show the number of times an MPI event occurs during the execution of the application and the skeleton. The results shown in the tables are derived from the DUMPI trace report. The MPI events are grouped by the function name and the count is shown for each function.

Table 6.1: HPCCG - MPI Event Count

MPI Function Name	**Count (Application)**	**Count (Skeleton)**
MPI_Init	128	128
MPI_Send	38862	38862
MPI_Irecv	38862	38862
MPI_Wait	38862	38862
MPI_Barrier	128	0
MPI_Allreduce	38912	38912
MPI_Finalize	128	128

Table 6.2: FFT - MPI Event Count

MPI Function Name	**Count (Application)**	**Count (Skeleton)**
MPI_Init	16	16
MPI_Barrier	16	16
MPI_Gather	16	16
MPI_Scatter	16	16
MPI_Alltoall	16	16
MPI_Finalize	16	16

The number of MPI events that occur in an application are determined by the evaluation of different branch conditions and the number of iterations the loops undergo. For a skeleton to match the application in terms of control flow and loop behavior, the number of MPI events (grouped by

MPI function) that occur in both the program and its skeleton must be equal (or relatively close to equal).

Table 6.3: Jacobi - MPI Event Count

MPI Function Name	Count (Application)	Count (Skeleton)
MPI_Init	16	16
MPI_ISend	4800	4800
MPI_Irecv	4800	4800
MPI_Wait	9600	9600
MPI_Barrier	16	16
MPI_Finalize	16	16

The trace files we dervied in our experimental evaluation (as shown in Tables 6.1 6.2 and 6.3) show that for each MPI function call, the number of events corresponding to the application and skeleton are essentially the same. This demonstrates the correctness of our skeleton extraction mechanism, with skeletons exhibiting the same behavior as that of the corresponding application. The communication pattern is determined by the point-to-point and collective MPI routines in the program. These MPI routines always occur in pairs with one process/rank sending the data and another process/rank receiving the data or all processes participating in the collective operations. One way to match the skeleton with the application in terms of communication patterns is to check whether the data transmitted by each rank match. The tables 6.4, 6.5, and 6.6 show the data transmitted in bytes by each rank and the total data transmitted in bytes both for the skeleton and the application. The results shown in the tables are derived from the DUMPI trace report.

The results shown in Tables 6.4, 6.5, and 6.6 demonstrate that skeletons exhibit similar communication patterns to the corresponding applications with each rank transmitting the same number of bytes. The DUMPI trace report also contains a transmission matrix, which shows the amount of communication between all ranks. These were compared for the full application and skeleton, and for readability, Table 6.7 presents the results for 3 ranks (out of a total of 128 ranks).

Table 6.4: HPCCG - Data Transmitted by Each Rank

Rank	**Transmitted in Bytes (Application)**	**Transmitted in Bytes (Skeleton)**
Rank 0	769020	769020
Rank 1 to 126	1.09772e+06	1.09772e+06
Rank 127	769020	769020
Total	1.39851e+08	1.39851e+08

Table 6.5: FFT - Data Transmitted by Each Rank

Rank	**Transmitted in Bytes (Application)**	**Transmitted in Bytes (Skeleton)**
Rank 0	6.0398e+08	6.0398e+08
Rank 1 to 15	6.71089e+07	6.71089e+07
Total	1.61061e+09	1.61061e+09

Table 6.6: Jacobi - Data Transmitted by Each Rank

Rank	**Transmitted in Bytes (Application)**	**Transmitted in Bytes (Skeleton)**
Rank 0,3,12,15	400000	400000
Rank 1,2,4,7,8,11,13,14	600000	600000
Rank 5,6,9,10	800000	800000
Total	9.6e+06	9.6e+06

Our analysis demonstrates a one-to-one matching between each skeleton and the application transmission matrices.

Table 6.7: Transmission Matrix

SrcDest	**Rank_0**	**Rank_1**	**Rank_2...**
Rank_0	3440	332140	3440
Rank_1	332140	3440	332140
Rank_2	3440	332140	3440

Scalability

In this section, we discuss how skeletons scale with respect to their corresponding applications. We compare the total execution time of a skeleton-driven simulation versus a simulation driven by the program's direct execution. In addition, we study the efficiency of our automatic skeletonization process by comparing the performance of our HPCCG automatically generated skeleton against an optimzied hand-written skeleton of HPCCG developed during a previous study with SST/macro.

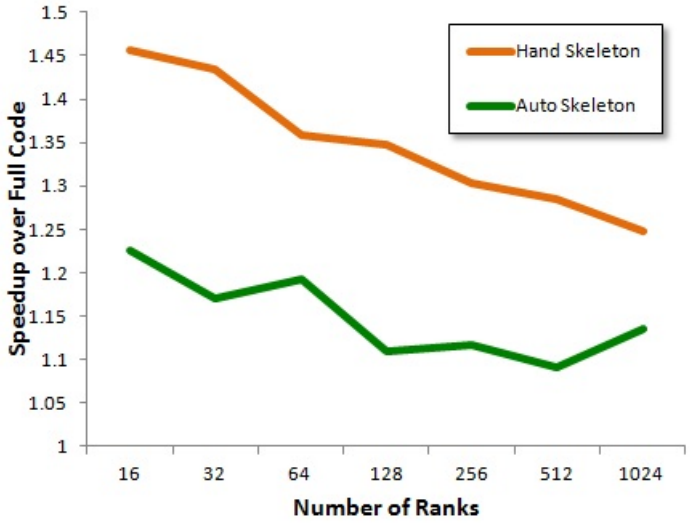

Figure 6.1: HPCCG - Speedup of Hand Written Skeleton Vs Auto Generated Skeleton

Figure 6.1 shows our experimental evaluation of both HPCCG skeletons. Careful inspection revealed that some routines in the auto-generated skeleton files remained unaltered with respect to the parent program because of statistical dependencies, leaving in some code that could have been removed. Both skeletons exhibit significant speedup over the full application, and the auto-generated one appears to scale well as the number of processors grows. The plot of simulation wall times for HPCCG full program and its skeleton as shown in figure 6.2 indicates that increments in scalability factor are almost similar with an increase in number of ranks.

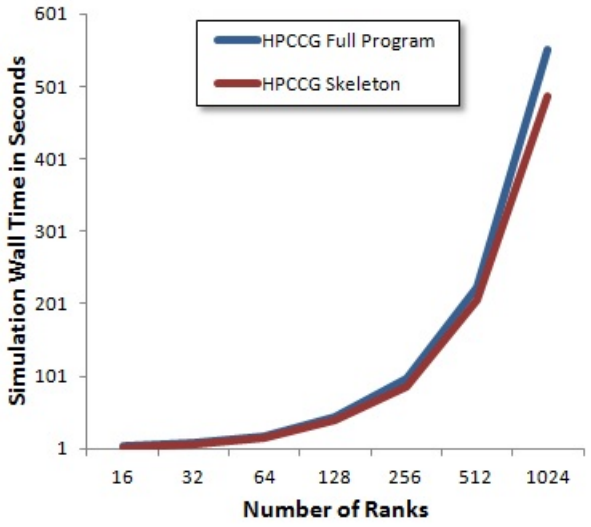

Figure 6.2: HPCCG - Full Program Vs Skeleton

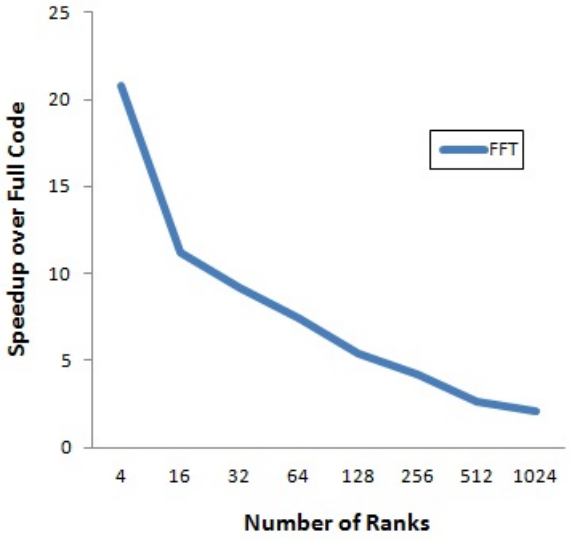

Figure 6.3: FFT - Speedup of Auto Generated Skeleton over Application

Figure 6.3 shows the speedup of the auto-generated FFT skeleton over its parent application.

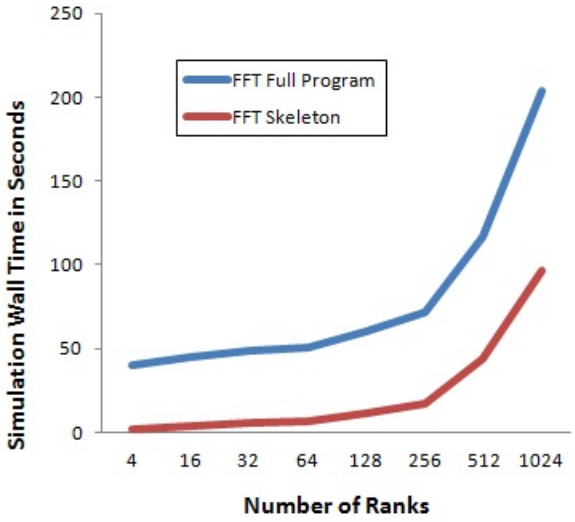

Figure 6.4: FFT - Full Program Vs Skeleton

The implementation is a simple sequence of computations where each processor performs a local computation that is distributed via collective communications to other processors, followed by a final computation that combines the results computed locally with those computed remotely. The plot of simulation wall times for FFT full program and its skeleton as shown in figure 6.4 also shows that the increase in scalablity factor remains constant for both the program and its skeleton with an increase in number of ranks. Our skeleton generator was successful in removing all the computational steps, leaving only the collective communication operations, thus comparison to a hand-written version is not needed. As processor count grows, the cost of simulating collectives grows super-linearly, decreasing the savings we see from the skeletonization process. Also, the problem was strong-scaled, reducing the amount of computation per processor as processor count grows.

Figure 6.5 shows the speedup of the auto generated Jacobi skeleton over its parent application.

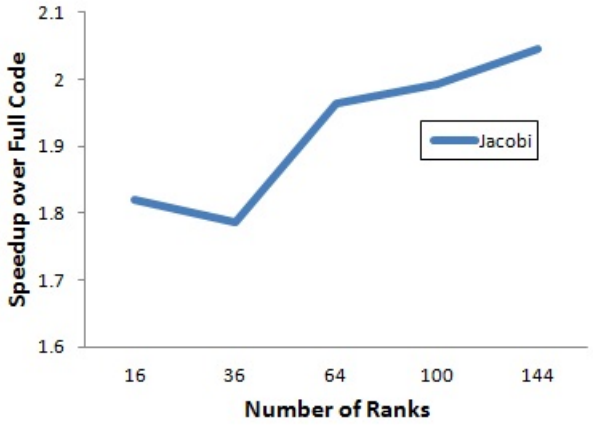

Figure 6.5: Jacobi - Speedup of Auto Generated Skeleton over Application

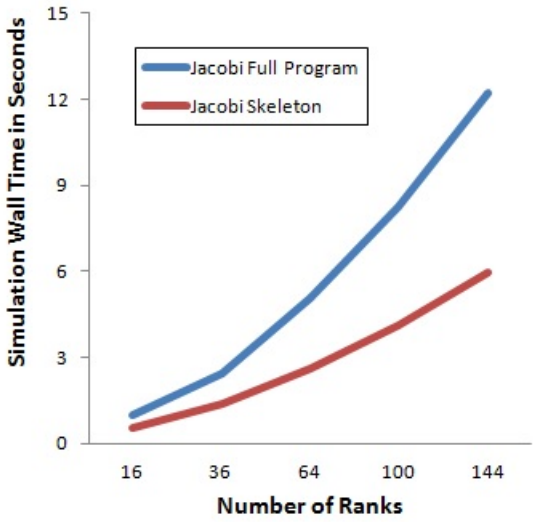

Figure 6.6: Jacobi - Full Program Vs Skeleton

The skeleton demostrates a consistent speedup of about 2 over its parent program with a gradual increase in the speedup with the increase in the number of processes. The plot of simulation wall times for Jacobi full program and its skeleton as shown in figure 6.6 shows that the increments in scalability factor is somewhat linear. The full program spends more time in computations due

to input size being a function of the ranks, which makes the problem size bigger with an increase in the number of ranks.

All skeletons we explored demonstrated performance gains when compared to their parent applications while at the same time preserving the application's behavior and correctness characteristics. The performance gain is the result of eliminating code that forms the computational part of the program and is not required to ensure that MPI operations execute correctly (e.g., buffer allocation code). If there is a significant reduction in the time spent in code unrelated to message passing in the skeleton, we observe significant performance gains in the skeleton relative to the parent program. In some cases where the computational load decreases as the processor count increases, we see scaling drop as the parent program performs less work per processor leading to less overhead in the parent application due to computation. This is true for our FFT example, in which a fixed problem size was decomposed across a set of processors such that as the processor count increases, the per-processor problem size decreases. In other cases where the problem size per processor is fixed regardless of the processor count (e.g., the Jacobi iteration example), the scaling of the skeleton versus the parent application would be a function of both this computational overhead along with the communication-specific scaling behavior.

Given that the canonical examples we studied in this work represent some of the core computational solver methods employed by many HPC applications, we expect that our automatic skeletonization tool can provide a direct benefit for the study of large-scale HPC codes. We also expect the results presented here to be a lower-bound on the savings achieved from skeletonization, given that these codes have relatively little computation to begin with.

CHAPTER 7: CONCLUSIONS

In this paper we presented a methodology based on static analysis for generating program skeletons for large-scale performance analysis. We demonstrate that dependency analysis augmented with dependency role information allows slicing techniques to be controlled by the user of the skeleton generator. Compiler directives allow the tool user to convey information to the algorithm that cannot be inferred automatically (or, cannot be inferred easily). Our experiments show that an iterative process of skeleton generation and injection of directives provides a relatively straightforward approach to skeleton creation versus completely manual techniques.

Our results illustrate that under the SST/macro simulator, the program skeletons being small in size execute faster than the original applications that they were derived from, while still giving an accurate estimation of original application's behavior. The results strongly show that program analysis is an effective strategy to generate skeletons which can be used to evaluate the scalability of large-scale parallel applications. Our static analysis approach is effective in reducing the time to create the skeleton as well as reducing the likelihood of errors being accidentally introduced during manual skeleton creation. The skeleton driven simulation of applications can be extended to accurately study the effect of varying hardware design parameters such as number of nodes, processors per node, network topology, memory bandwidth and latency on the scalability of an application.

CHAPTER 8: FUTURE WORK

We plan to carry foward our work by exploring other approaches which might provide an improvement over the correct mechanism i.e. using SSA to accurately derive progrram slices. We plan to implement skeleton extraction framework by employing static slicing that uses system dependence graph(SDG). SDG is a dependency graph which represents the program as a directed graph with dependencies of several nodes towards each other. SDG is constructed by merging Control Dependence Graph (CDG), Data Dependence Graph (DDG), Function Dependence Graph (PDG) and Inter Procedural Information (IPI). Once all program constructs have been added to the SDG, performing `InterproceduralAnalysis` on SDG performs the connection of `call-sites` to all possible called functions and establishes summary-edges. Each PDG represents a function in the SDG. Summary edges are established by linking each call-site to the PDG associated with the function it calls. The call-site node is linked to the entry node with a call edge. Each actual-in node is linked to the formal-in node with a call edge. Each formal-out node is linked to the actual-out node with a return edge. Once the SDG is generated, we set our slicing criteria nodes as MPI calls in the program. We then do data flow analysis on the SDG to identify dependencies between MPI nodes and rest of the nodes in the program. This information is used to identify all the nodes which affect the topology parameters in an MPI call. All thoses nodes identified are set to be retained as part of the program skeleton, and rest of the nodes are marked to be eliminated if not found to syntactically affect the program. Now the slice contains only those lines of code that could affect the result of the chosen statements.

The slicing algorithm we plan to employ follows the approach defined according to the paper by Horwitz et al [2]. We invoke the `getSlice` method on SDG to perform backward slicing. The algorithm uses two phases to perform slicing. In the first phase, the algorithm operates on the concept of backwards reachability to mark nodes while not traversing return edges. Thus it ignores

function calls. In the second phase, algorithm on the same concept of backwards reachability from all marked nodes while not traversing call edges. Thus it ignores calling functions. The final set of reachable nodes is the interprocedural slice. The slicing, while eliminating nodes corresponding to computational routines, reads the annotation information present in them and inserts cpp directives in place of them indicating the routine name and its execution time. We then carry out MPI transformations on the sliced-out AST.

We have already explored this approach with ROSE to construct SDG graphs for a number of small applications, but we found it lacking in terms handling many program constructs in an application and doing data flow analysis on the SDG. SDG does not use aliasing analysis which makes it difficult to produce precise result if the program contains pointers and arrays. However, we intend to explore one more commercially available tool called CodeSonar which is a source code analysis tool that performs a whole-program, interprocedural analysis on C and C++ [3].

LIST OF REFERENCES

[1] V. S. Adve, R. Bagrodia, E. Deelman, and R. Sakellariou. Compiler-optimized simulation of large-scale applications on high performance architectures. *J. Parallel Distrib. Comput.*, 62:393–426, March 2002.

[2] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In P. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 206–222. Springer Berlin / Heidelberg, 1993. 10.1007/BFb0019410.

[3] GrammaTech. CodeSonar. `http://www.grammatech.com/codesonar`.

[4] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo. A simulator for large-scale parallel architectures. *International Jrnl. of Parallel and Distributed Systems*, 1(2):57–73, 2010.

[5] S. N. Laboratories. The Structural Simulation Toolkit. `http://sst.sandia.gov/`.

[6] Lawrence Livermore National Laboratory. Rose Compiler. `http://www.roseCompiler.org`.

[7] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. Supinski, and D. J. Quinlan. Using mpi communication patterns to guide source code transformations. In *Proceedings of the 8th international conference on Computational Science, Part III*, ICCS '08, pages 253–260, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Sandia National Laboratories. HPCCG. `http://bec.syr.edu/hpccg.html`.

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

[10] J. Subhlok and Q. Xu. Automatic construction of coordinated performance skeletons. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –5, april 2008.