ScalaBenchGen: Auto-Generation of Communication Benchmarks Traces

Xing Wu
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
Email: xwu3@ncsu.edu

Vivek Deshpande
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
Email: vrdeshpa@ncsu.edu

Frank Mueller
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
Email: mueller@cs.ncsu.edu

***Abstract*—Benchmarks are essential for evaluating HPC hardware and software for petascale machines and beyond. But benchmark creation is a tedious manual process. As a result, benchmarks tend to lag behind the development of complex scientific codes.**

This work contributes an automated approach to the creation of communication benchmarks. Given an MPI application, we utilize ScalaTrace, a lossless and scalable framework to trace communication operations and execution time while abstracting away the computations. A single trace file that reflects the behavior of all nodes is subsequently expanded to C source code by a novel code generator. This resulting benchmark code is compact, portable, human-readable, and accurately reflects the original application's communication characteristics and runtime characteristics. Experimental results demonstrate that generated source code of benchmarks preserves both the communication patterns and the wallclock-time behavior of the original application. Such automatically generated benchmarks not only shorten the transition from application development to benchmark extraction but also facilitate code obfuscation, which is essential for benchmark extraction from commercial and restricted applications.

I. Introduction

Benchmarks are widely used for evaluating and analyzing system performance. They also assist in assessing migration costs of HPC applications to new platforms with different architectures. Benchmarks tend to be easy to port, modify and run, and they are said to closely resemble the characteristics of HPC applications.

But most benchmarks do not capture the complexity and scale of realistic HPC applications as they do not feature the intricate interplay of computation, communication and I/O operations. Many benchmarks also tend to lag behind the development cycle of their corresponding full-scale application. To some extent, this is due to the excessive manual effort involved in manually extracting benchmarks from full-scale applications. To another extent, benchmarks derived from applications subject to distribution restrictions require a lengthy review process before they can be released. To overcome these challenges, an automated method for benchmark generation from applications is needed. As benchmarks are created from application traces, the generated source code of such benchmarks obfuscates the original application source code.

This paper contributes a novel auto-generation process for *communication benchmarks*. The resulting benchmarks are human readable, compact and easier to port across architectures than their full-fledged application counterparts. They closely resemble the execution time and communication characteristics in terms of message frequencies and sizes of the original application. Yet, their source code is obfuscated in the auto-generation process, which supports an expedited release for otherwise restricted application domains.

Our auto-generation process works with HPC applications that utilize message passing communication via MPI (Message Passing Interface) [1]. This is illustrated in Figure 1. During an application run, the application's communication patterns/events and elapsed wallclock times between events are captured in a communication trace. The obtained trace provides the input to the benchmark generator, which is the central focus of this work. The generator outputs a communication benchmark in C code (including MPI calls for communication) that can be executed on a target machine.

Figure 1: Benchmark Generation Workflow

We utilize ScalaTrace [2] to collect communication traces. ScalaTrace is a unique approach to parallel application tracing as this scalable framework captures the communication in lossless and near constant size in terms of trace representation independent of the number of the nodes while preserving the structural information of the nodes and iterations. It also employs pattern-based intra-node and inter-node compression techniques to extract the application's communication structure.

Our communication benchmark generator is evaluated using the NAS Parallel Benchmark Suite [3] and Sweep3D [4]. We show that an auto-generated benchmark preserves the application's semantics in terms of its communication pat-

This work was supported in part by NSF grants 1058779, 0958311, 0937908. It used resources of the National Center for Computational Sciences at ORNL and was performed in part at ORNL, managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725.

tern along with communication volume and the ordering of events relative to the original HPC application. Furthermore, the overall execution time of benchmarks is close to that of their original applications. Thus, the communication benchmark generator is able to generate benchmarks that closely resemble the original application in terms of communication behavior and execution time.

The contributions of this work are (1) a demonstration and evaluation of the feasibility of automatically converting parallel applications into human-readable benchmark codes and (2) an approach and algorithmic description of generic portions for auto-generated benchmark codes that resemble the original application performance. This process becomes feasible due to structural, domain-specific compression of traces.

The benefits of this work extend to application developers, communication researchers, and HPC system designers alike. Application developers can benefit in multiple ways. First, they can quickly gauge the application performance of a target machine before investing in the effort to port their applications to that machine. Second, they can use the generated benchmarks for performance debugging as the benchmarks can separate communication from computation to help isolate observed performance anomalies. Third, application developers can examine the impact of alternative application implementations, such as different data decompositions (causing different communication patterns) or the use of computational accelerators (reducing computation time without directly affecting communication time). Communication researchers can benefit by being able to study the impact of novel messaging techniques without the need to build complex applications and without access to source code that is not freely distributed or even classified. Finally, procurement of HPC systems can benefit by contracting vendors to deliver a specified performance on a given auto-generated benchmark without having to provide those vendors with the actual application.

In summary, we have developed a tool that automatically generates C code of a communication benchmark with MPI calls from HPC applications such that the characteristics of the original application are preserved in terms of time and structure. The generated code is human readable, compact, easily portable and obfuscated with respect to the original application.

II. ScalaTrace

Our work builds on ScalaTrace, an MPI tracing toolkit with aggressive and scalable trace compression. ScalaTrace's compression can result in trace file sizes orders of magnitude smaller than previous approaches or, in some cases, even near constant size regardless of the number of nodes or application run time [2].

The tool collects communication traces using the MPI Profiling layer (PMPI) [5] through Umpire [6] to intercept MPI calls during application execution. On each node, profiling wrappers trace all MPI functions, recording their call parameters, such as source and destination of communications, but without recording the actual message content.

ScalaTrace performs two types of compression: $intra-node$ and $inter-node$. For the intra node compression, the repetitive nature of timestep simulation in parallel scientific applications is used. Intra-node compression is performed on-the-fly within a node. Further, the inter-node merge exploits the homogeneity in behavior across different processes running the application due to the HPC-prevalent single-program-multiple-data (SPMD) programming style. Inter-node compression is performed across nodes by forming a radix tree structure among all nodes and sending all intra-node compressed traces to respective parents in the radix tree. At the parent, the respective trace representations are merged, reduced and then compressed exploiting domain-specific properties of MPI. Once propagated to the root of the radix tree, this results in a single compressed trace file capturing the entire application execution across all nodes. The compression algorithms are discussed in detail in other papers [7], [8].

As a result of these techniques, ScalaTrace achieves near constant size traces by applying pattern based compression. It uses extended regular section descriptors (RSD) to record the participating nodes and parameter values of multiple calls to a single MPI routine in the source code across loop iterations and nodes in a compressed manner [9]. Power-RSDs (PRSD) recursively specify RSDs nested in a loop [10].

Example: Consider the code snippet shown in Figure 2 with ring-style communication across N nodes.

```
for(i=0; i<100; i++){
    MPI_Irecv(LEFT, ...);
    MPI_Isend(RIGHT, ...);
    MPI_Waitall(...);
}
```

Figure 2: Ring-style MPI Communication Code

ScalaTrace represents these events as three RSDs in the trace (see Figure 3) to denote the non-blocking send, receive and waitall MPI operations of a single loop iteration, where $\langle rank \rangle$ represents a value within $0 \ldots N-1$ in each per-node trace. ScalaTrace then detects the loop structure and outputs a single PRSD to denote a single loop of 100 iterations. This intra-node compression is performed on-the-fly to reduce the time for trace generation and the memory overhead.

RSD1: {$\langle rank \rangle$, MPI_Irecv, LEFT}
RSD2: {$\langle rank \rangle$, MPI_Isend, RIGHT}
RSD3: {$\langle rank \rangle$, MPI_Waitall}
PRSD: {100, RSD1, RSD2, RSD3}

Figure 3: Intra-node Compressed Trace

Further, during the inter-node compression, the local traces on each node are combined into a single global trace when the application terminates (i.e., within the PMPI interposition wrapper for MPI_Finalize). Inter-node compression detects similarities among the per-nodes traces and merges the RSDs by combining their participant lists in a final participant list. For the example above, each MPI routine is called on each node with the same parameters resulting in the following inter-node trace depicted in Figure 4.

RSD1: $\{0, 1, \ \ldots, N-1$, MPI_Irecv, LEFT$\}$
RSD2: $\{0, 1, \ \ldots, N-1$, MPI_Isend, RIGHT$\}$
RSD3: $\{0, 1, \ \ldots, N-1$, MPI_Waitall$\}$

Figure 4: Inter-Node Compressed Trace

The participant node information is encoded and represented as a *ranklist*. A ranklist is a recursive representation that describes the participating ranks by showing the starting rank, the nesting depth, and the iteration count and stride along each dimension. Hence, even multi-dimensional information is captured in this encoding format. There are special cases in which events with matching calling context can have non-matching function parameters. These non-matching function parameters are compressed using a vector representation so that the particular event can be concisely represented in the trace.

Another important feature of ScalaTrace is the time preservation of captured traces. Instead of recording absolute timestamps, the tool records delta time of computation durations between adjacent communication calls. During RSD formation, instead of accumulating exact delta timestamps, statistical histogram bins are utilized to concisely represent timing details across the loop. These bins are comprised of statistical timing data (minimum, maximum, average and standard deviation). ScalaTrace records histograms of delta times for each instance of a particular computation, i.e., distinguishing disjoint call paths by separate histogram instances.

III. Benchmark Generator Design

In this section, we introduce the system design in detail and discuss the considerations behind the design decisions.

A. System Overview

The process of automatic benchmark source code generation from communication traces is accomplished by traversing through the trace of a parallel application obtained from ScalaTrace. The trace traversal framework is designed to walk through all the RSDs and PRSDs. For each RSD and PRSD, the code generator is invoked to generate the respective C code and MPI calls. The code generator uses the predefined interfaces provided by the traversal framework, making the code generator a pluggable module. Thus, the same platform can be used to generate the code for different languages by writing code generators for those languages providing flexibility in generating code other than C.

While a trace can be seen as a linked list of RSDs and PRSDs, the trace traversal framework does not simply linearly traverse the list. Instead, it follows the hierarchical trace structure by traversing "into" PRSDs. In essence, PRSDs captures the loop structures in the source code. Hence, the traversal occurs recursively and code is emitted for each node (PRSD) to reflect the original program structure within the generated benchmark. At the entry and exit points of the recursive invocations, *for* loop entries/exits are generated:

$$for(i_n = 0;\ i_n < x;\ i++)\{\}$$

The nesting depth is tracked and a series of iterator variables that are dynamically assigned/reassigned:

$$i_1, i_2, ..., i_n$$

The corresponding declarations are added to the header file that is generated after the traversal. The trace traversal framework can be configured to iterate only once or multiple times for a PRSD. The latter is typically used in dynamic scenarios such as trace replay.

During traversal, the RSDs that represent point-to-point communication are converted to respective point-to-point MPI calls in C code. For example, blocking sends and receives are transformed to MPI_Send and MPI_Recv. Collective calls are generated using MPI collective routines in C, such as MPI_Barrier, MPI_Reduce, MPI_Alltoall and so on. The generation of MPI request handles and communicators will be discussed in Section III-B. Behavioral constraints captured by traces are reflected in the generated code using conditionals on ranks of the processes participating in a particular event:

$$if(is\_member(myrank, ranklist)).$$

Figure 5 is a simple example of a C program generated from a trace. The delta-time sleep simulates a computational phase in the original application. In a nutshell, because ScalaTrace traces structurally resemble the original source codes, the benchmarks generated from them are concise and highly readable. Table I provides a list of the generated program files and their respective functions.

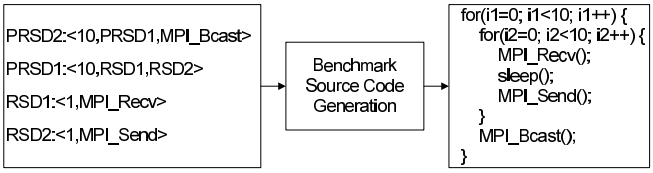


Figure 5: The Code Generation

Table I: The Generated Files and their Functions

File Name	Function
main.h	Header file with declarations of iterator variables
main.c	Main function implementing the communication skeleton
util.h	Header file with declarations of utility functions
util.c	Utility functions including the request handle management, communicator management, MPI function wrappers, etc.

B. Request Handles and Communicators

MPI_Isend and MPI_Irecv generate request handles. Subsequently, MPI_Wait and MPI_Waitall use those handles to block the processes until the sending or receiving is complete.

Instead of analyzing how many MPI_Request variables are needed and generating unique names for them, we employ a ring buffer to hold all the request handles in the generated code. During benchmark execution, a request handle is added to the tail of the ring buffer. An index into this buffer signifies the location of the request pointer for a non-blocking event. This pointer is used within matching MPI_Wait and MPI_Waitall calls. After the wait call, the request handle is invalidated in the request buffer to ensure that the same request handle can be reused (reinitialized) by subsequent wait scenarios.

Communicators are handled in a similar way. In the generated code, each communicator, including the default ones such as MPI_COMM_WORLD and the user-defined ones, is assigned an index that specifies the location where the communicator is stored within an array. This index is subsequently used to locate a communicator handle passed to an MPI routine as a parameter. Whenever a new communicator is generated by events such as MPI_Comm_split or MPI_Comm_dup, the next unused element in the communicator buffer will be assigned. With the values of the *color* and *key* parameters recorded in the trace, the generated code is thus able to correctly recreate the user-defined communicators.

C. Generating the Computation Phases

As was discussed in Section II, ScalaTrace preserves the timing information of an application. The length of a computation phase between consecutive MPI events is recorded as a delta time associated with the latter event. An event may be associated with multiple time records if it has multiple different preceding events, *i.e.,* the event can be reached from disjoint execution paths.

In a trace replay scenario, the replay engine chooses the correct sleep time by searching the call stack signature of the previous event in the list of time records of the current event. However, it cannot be done by the code generator for two reasons: (1) Unlike the replay engine, the code generator traverses a loop structure only once so that certain execution paths will never be reached during the traversal. (2) Unlike the replay engine in which each node preloads its per-node trace, the code generator is a single-node program that traverses the complete trace. As a result, an event e_a immediately preceding e_b at runtime may be recorded far ahead in the trace.

Algorithm 1 Computation Phase Generation Algorithm

Precondition: T: input trace, e: current event

```
1: procedure GENERATE_COMPUTE_TIME(T, e)
2:     TimeRecords = e.timeRecords
3:     for t ← TimeRecords.head, TimeRecords.tail do
4:         rl = e.ranklist
5:         iter = e.prev
6:         while iter != NULL && rl != NULL do
7:             intersec = rl.INTERSECT(iter.ranklist)
8:             if intersec != NULL && iter.SIGMATCH(t.sig)
   then
9:                 GENERATE("if(is_member(myrank, intersec"))
10:                GENERATE("compute(t.time);")
11:                rl = rl.SUBTRACT(intersec)
12:            end if
13:            iter = iter.prev
14:        end while
15:    end for
16: end procedure
```

To address this problem, Algorithm 1 was designed for the code generator. For each time record t_i of an MPI event e, the code generator traverses the trace backward as far as necessary. During traversal, it searches for events whose call stack signature matches that of t_i, but only the events whose ranklist has a non-empty intersection with the ranklist of e are considered. Every time such an event is found, a conditional on the common ranks is generated so that these ranks (under a particular permutation of iterator values) will sleep for the corresponding time units. The backward traversal continues until every participating rank of e has found its preceding event with the matching call stack signature.

```
for(i1=0;i1<a;i1++) {
    for(i2=0;i2<b;i2++) {
        if(is_member(myrank, "...")){
            if(i2 == 0 && i1 == 0)
                compute(x);
            if(i2 != 0)
                compute(y);
            if(i2 == 0 && i1 != 0)
                compute(x);
            ...
        }
    }
}
```

Figure 6: Compute Times Generated for Multi-nest Loops

Events at the beginning of the multi-nest loops are associated with multiple time records as they can be reached

from loops of different depth. For these events, we generate conditionals on particular permutations of the iterator values, as illustrated in Figure 6. This is implemented by comparing the call stack signature of each time record with each of the tail events of loops of different depth. Once a match is found, a predicate (conditional) of the iterator variables is generated according to the depth of the loop.

During code generation, it is helpful to exploit the statistical information that was recorded as histograms within the trace, particularly for applications with drastically different timing behavior across nodes. With this capability, unbalanced workloads can be accurately represented in the generated code.

D. Generating Concise and Readable Programs

The conciseness of the generated benchmarks is primarily decided by the degree of compression ScalaTrace can achieve. For example, the most aggressive lossless compression techniques are supported by ScalaTrace with the following configuration options:

1) Ranklist encoding of the participant list: efficiently describes the spatial distribution of nodes instead of listing ranks one by one.
2) Ignore call stack signatures: match events according to their types instead of by call stack signatures.
3) Vectorization of MPI parameters: record the non-matching function parameters across loop iterations with vectors
4) WLCS-based recursive inter-node compression: generate shortest traces during inter-node compression [11].

With these techniques, the final trace tends to be structurally very simple and similar to the original source code. Thereby, the generated code also resembles the original program.

Nonetheless, while the vectorization of communication parameters , such as *src*, *dest*, and *count* greatly improves trace compression, it introduces readability problems to the generated code. To handle the vectorized communication parameters, there are two options. First, we can unroll the vectorized parameters by generating a sequence of conditional statements with respect to the iterators. But for highly compressed traces, this approach may lead to excessively long programs even for a single RSD. Therefore, we adopted the second approach. It consults the vector representations during code generation and lets the generated code parse them on-the-fly. This approach makes the generated code much shorter and more readable as long as the parameter vectors are of moderate sizes. For the events that are called multiple times in loops, the parameter vectors are parsed only once when they are reached the first time. To further improve the readability, we generate wrapper functions for the MPI events. These wrapper functions call the real MPI routines while hiding the tedious processing steps, such as parsing the ranklists and vectorized parameters, retrieving communicators, generating request handles, and handling errors.

IV. Experimental Framework

To evaluate our communication benchmark generation tool, we generated C code with MPI calls for the NAS Parallel Benchmarks (NPB) suite (version 3.3 for MPI) using class C and D input sizes [3] and for the Sweep3D neutron-transport kernel [4]. These codes all have either mesh-neighbor communication patterns or rely heavily on collective communication. Some of them (e.g., SP and BT) require communicator handling, others (e.g., IS) require averaging of parameters in MPI_Alltoallv and some (e.g., LU) require the recording of wildcard receives. Hence, the key features of our code-generation framework are thoroughly tested in these experiments.

Benchmark generation is based on traces obtained on (a) ARC, a cluster with 1728 cores on 108 compute nodes, 32 GB memory per node and an Infiniband Interconnect and (b) Jaguar, a petascale HPC installation at Oak Ridge National Laboratory with 18,688 compute nodes where each compute node contains dual hex-core processors, 16 GB memory, and a SeaStar2+ router. Benchmark generation is performed on a stand-alone workstation.

V. Experimental Results

We performed the following experiments for the evaluation of our benchmark generation tool.

A. Correctness

Our first set of experiments verifies the correctness of the generated benchmarks, *i.e.*, the benchmark generator's ability to retain the original applications' communication pattern. For these experiments, we acquired traces of our test suite on ARC, generated communication benchmarks, and executed these benchmarks also on ARC. To verify the correctness of the generated benchmarks, we linked both the generated codes and the original applications with mpiP [12] (see Figure 7, upper half). The mpiP tool is packaged as a lightweight MPI profiling library that gathers run-time statistics of MPI event counts and the message volumes exchanged. Experimental results (not presented here) showed that, for each type of MPI event, the event count and the message volume measured for each generated benchmark matched perfectly with those measured for the original application.

We then conducted experiments to verify that the generated benchmarks not only resemble the original applications in overall statistics but also that they preserve the original semantics on a per-event basis. To this end, we instrumented each generated benchmark with ScalaTrace and compared its communication trace with that of its respective original application. Due to differences in the call-site stack signatures between the original application and

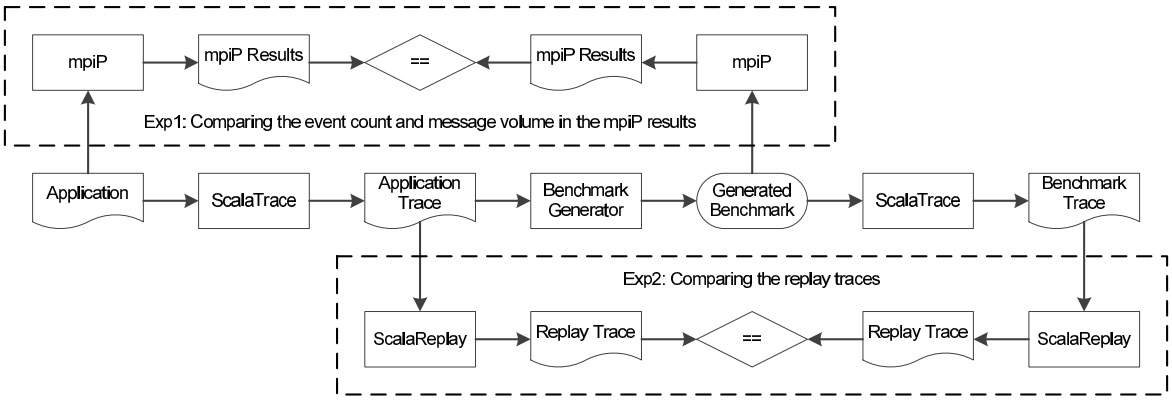


Figure 7: Experimental Framework

the generated benchmark, these traces cannot be identical, they can only be semantically equivalent. Therefore, we replayed both traces with the ScalaTrace-based ScalaReplay tool [13] to eliminate spurious structural differences and thus allow a fair comparison of traces as depicted in Figure 7 (lower half). The results (again, not presented here) show that the original applications and the generated benchmarks have equivalent traces. That is, the semantics of each of the original applications was precisely reproduced by the corresponding generated benchmark.

B. Accuracy of Timing Results

After evaluating that the generated code preserves the communication of the original application in terms of ordering of events and message volumes, we assessed the ability of a generated benchmark to retain the performance in terms of wall-clock time relative to the original application. To measure the execution times of the original applications, we extended the PMPI profiling wrappers of MPI_Init and MPI_Finalize to obtain the start and end timestamps, respectively. The corresponding timing calls were also added to the generated benchmarks. We executed both the original application and the generated benchmark on the ARC system, measured and compared the elapsed times. The results obtained are shown in Figure 8. The x-axis shows the node sizes and problem sizes for the NPB experiments. For example, C-16 means Class C input size and 16 MPI processes were used. In this set of experiments, we mostly used Class C input sizes. Nonetheless, for the NPB CG, EP, IS, and MG codes, whenever Class C is not large enough to scale to produce a reasonable workload per MPI task (computation to communication ratio) for a certain node size, we switched to and report the results obtained from Class D inputs.

We observe from the graphs that the timings obtained for the generated benchmarks are very close to that of the original applications indicating very high accuracy. Quantitatively, the mean percentage error obtained by the formula

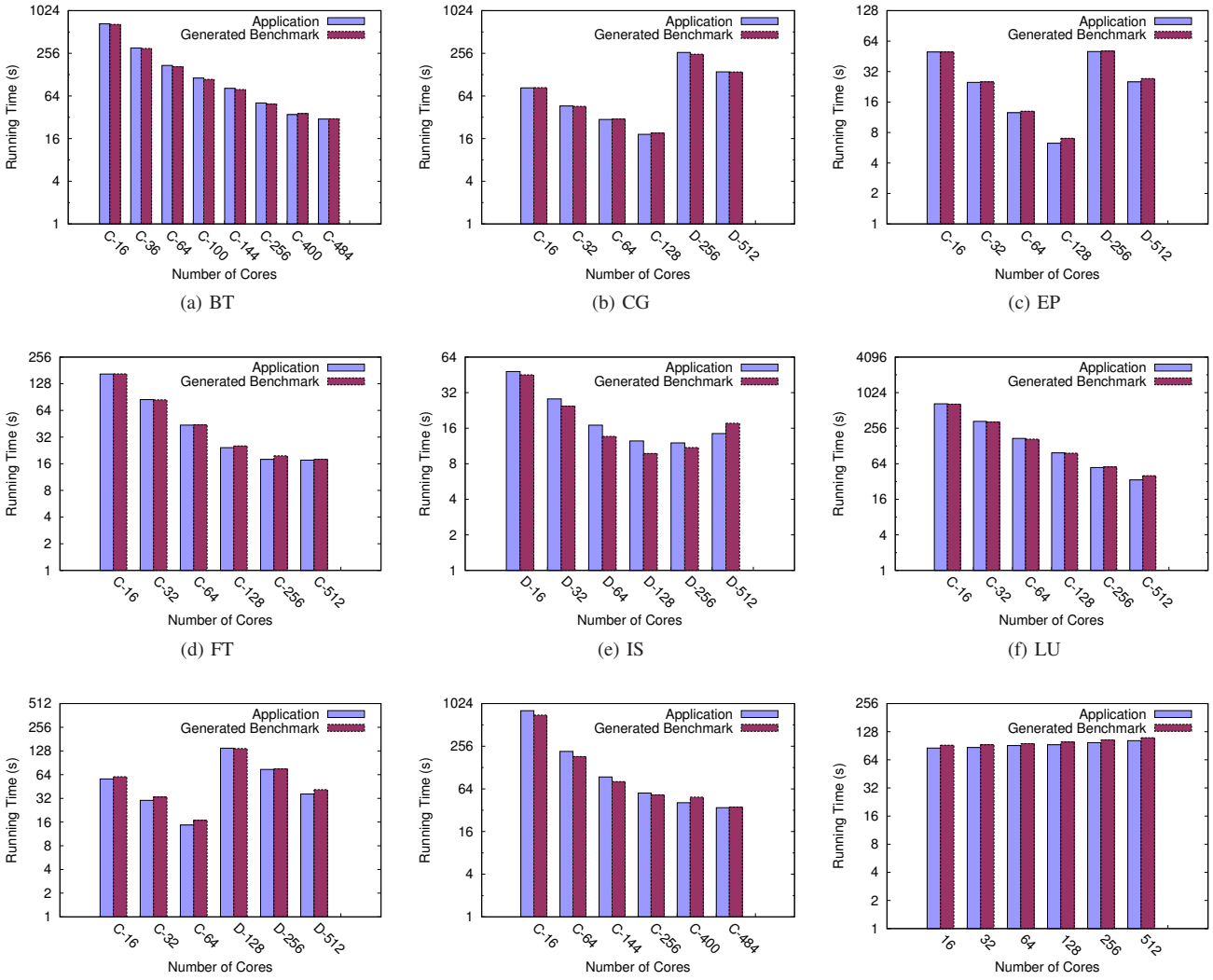
$$100\% \times |(T_{\text{gen}} - T_{\text{app}})/T_{\text{app}}|$$

across all the graphs is only 6.4%. Across all the benchmarks, IS is the benchmark for which the generated code has the lowest timing accuracy. The mean percentage error over the six node sizes of IS is 15.3% with deviations of 21.6% and 22.1% observed for 128 and 512 MPI processes, respectively. IS performs a distributed bucket sort algorithm on a set of randomly generated integers. It utilizes MPI_Alltoallv to exchange data across nodes. Because the total number of integers exchanged is a constant and the buckets are of similar sizes due to dynamical load balancing, the code generator uses MPI_Alltoall to simulate the communication pattern of MPI_Alltoallv so that the generated code is concise and more readable. As a result of this trade-off between trace compression and precision, the slightly diverging timing behavior across the set of nodes is missing in the generated code so that timing accuracy is compromised. This effect is further amplified by the fact that network contention increases with increasing numbers of MPI tasks as the computational phases become shorter due to strong scaling. As a result, we observe worse timing accuracy at large node sizes.

C. Cross Platform Results

We obtained cross platform results by running the benchmarks generated from IS and MG on ARC and Jaguar. The results are depicted in Figures 9 and 10.

Figure 9 shows that, in case of the IS benchmark, the difference between the execution times of benchmark from ARC and the original application on Jaguar reduces as the number of processors increases. This is because the computation is split across a larger number of processors reducing the per-processor computation to communication ratio and thus reducing the effect of higher processing capacity of Jaguar. Also, for the IS benchmark, the lowest time in the 16-512 processor range is obtained for 64 processors on the



(a) BT (b) CG (c) EP

(d) FT (e) IS (f) LU

(g) MG (h) SP (i) Sweep3D

Figure 8: Graphs for Timing Accuracy of NAS PB Codes for Input Class C/D and Sweep3D.

ARC cluster resembling the actual application behavior. The same benchmark with delta times from ARC but executed on Jaguar resulted in the lowest time for 256 processors on the latter platform. This is also matching the lowest runtime (at 256 processors) of the original application on Jaguar (within the 15-512 processor range).

Figure 10 shows that the execution time of the MG benchmark obtained on ARC is close to that of the original application on Jaguar, whereas the execution time for the MG benchmark obtained on Jaguar itself very closely resembles it. The difference is due to diverging CPU speeds between ARC and Jaguar. Since Jaguar has a higher processor frequency than ARC, it finishes the computation earlier than indicated by the delta time for sleeps obtained by tracing on ARC.

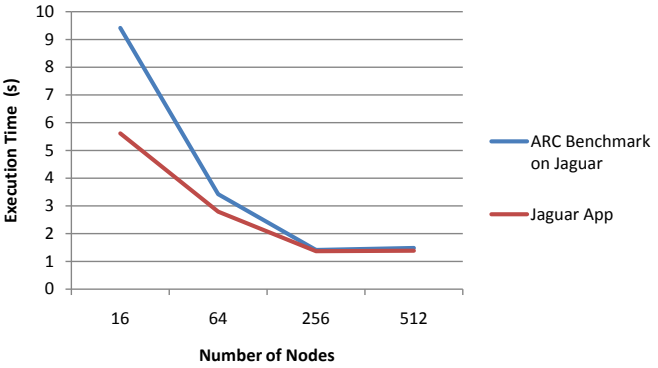


Figure 9: Cross-Platform Timing Results of IS

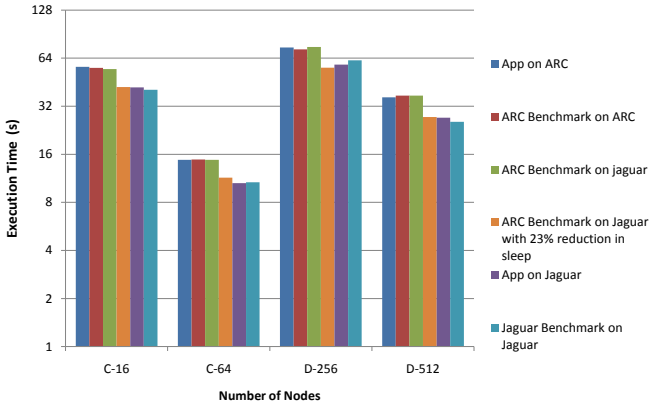


Figure 10: Cross-Platform Timing Results of MG

To verify the speedup of Jaguar over ARC, we executed a computational kernel that performs matrix multiplication on a single processor for square matrices of size 100x100 with iterations ranging from 3000 to 9000. Execution times are given in the Table II. The CPU speedup of Jaguar over ARC is around 23%, which conforms our observations from traces.

Table II: Execution Times [sec] for Matrix Multiplication on ARC and Jaguar

# Iterations	Time (ARC)	Time (Jaguar)	Speedup(%)
3000	44.168	34.259	22.43479442
6000	88.314	68.565	22.36225287
9000	132.443	102.614	22.5221416

We then reduced the sleeps in the MG benchmark obtained on ARC by 23% by proportionally shortening the delta times in the traces from ARC. The resulting MG run on Jaguar (see Figure 10, 4th bar) shows that the execution time then matches very closely to that the actual MG application on Jaguar.

Such performance experiments performed with the benchmarks generated by our tool could help in gauging different performance aspects related to communication on HPC systems with increasing complexities without actually porting the real applications to those platforms.

D. Lines of Code

We also measured the number of lines of code (LOC) in the generated code to gauge the conciseness of the generated code. The results are reported in Table III. In this table, the *App LOC* column depicts the lines of code of the native applications and the *Benchmark LOC* column reflects the lines of code of the generated benchmarks. We report the results for node sizes from 16 to 512 and the input size of Class C for the NPB codes. For those benchmarks whose code sizes become larger as the number of nodes is increased, we report the lines of code as a range to show their scalability. The *Change* column is calculated from the data in the previous two columns. A negative percentage number indicates that the generated code is shorter than the original application source code.

As shown in Table III, the number of lines of the generated code are generally lower than that of the native application. As indicated in the *Scalability* column, the generated benchmarks can be divided into two categories: Constant and Sub-linear. BT, EP, FT, IS, LU, SP, and Sweep3D belong to the first category. For these applications, the lines of the generated code does not increase with the node size, which is non-trivial for any trace-based approaches. Among these applications, only EP has a smaller code size than its generated counterpart. But given that the communication skeleton — the meaningful part of the generated code — exists only in *main.c*, if we exclude the utility functions implemented in *util.c* and *util.h*, the lines of the generated EP code is only 71, which is much smaller than that of the native application. CG and MG lead to increasing number of lines of code for the generated benchmarks because their complicated communication patterns cause imperfect trace compression. In CG, each node communicates with an increasing number of neighbors as the topology size grows with the total number of MPI processes [13]. In MG, nodes exhibit a doubly nested 7-point stencil communication pattern and diverging per-node program behavior (non-SPMD) [11]. Nonetheless, we still observe a sub-linear trend with respect to the total number of nodes.

Overall, the generated benchmarks are concise and manageable in size. Applications with more complicated communication patterns and non-SPMD program behavior should be handled with more aggressive compression techniques in ScalaTrace.

VI. Related Work

We utilize ScalaTrace to collect the communication trace of parallel applications. With a set of sophisticated domain-specific trace-compression techniques, ScalaTrace is able to generate traces that preserve the original source-code structure while ensuring scalability in trace size. Other tools for acquiring communication traces, such as Vampir [14], Extrae/Paraver [15] and tools based on the Open Trace Format [16], do not have structure-aware compression. This results in trace file sizes that grow at least linearly with the number of MPI calls and the number of MPI processes. This also increases the size of any benchmark generated from such a trace, making it not only inconvenient for processing long-running applications executing on large-scale machines but also losing the ability to resemble the original loop structure of an application. This lack of scalability is addressed in part by call-graph compression techniques [17] but still falls short of the structural compression of ScalaTrace, which extends to any event parameters. Casas et al. utilize

Table III: Comparison of Number Lines of Code

	App LOC (A)	Benchmark LOC (B)	Change = $(B-A)/A*100\%$	Scalability
BT	9217	705	-92.4%	Constant
CG	1796	1057 $\sim$ 2815	-41.1% $\sim$ 56.7%	Sub-linear
EP	325	538	65.5%	Constant
FT	2165	552	-74.5%	Constant
IS	1141	571	-50.0%	Constant
LU	5937	2504	-57.8%	Constant
MG	2580	2603 $\sim$ 9507	0.9% $\sim$ 268.5%	Sub-linear
SP	4922	689	-86.0%	Constant
Sweep3D	2096	1110	-47.0%	Constant
Average	-	-	-42.5% $\sim$ -6.6%	-

techniques of signal processing to detect internal structures of Paraver traces and extract meaningful parts of the trace files [18]. This facilitates trace analysis in a compressed manner but does not allow one to capture full information and becomes lossy and thus is not suitable for benchmark generation.

Xu et al.'s work on constructing coordinated *performance skeletons* to estimate application execution time in new hardware environments [19], [20] exhibits some similarities with our work. However, a key aspect of performance skeletons is that they filter out "local" communication (communication outside the dominant pattern). As a result, the generated code does not fully reflect the original application, which may cause subtle but important performance characteristics to be overlooked. Because our benchmark generation framework is based on lossless application traces, it is able to generate benchmarks with identical communication behavior to the original application.

Program slicing, statically reducing a program to a minimal form that preserves key properties of the original one, offers an alternate approach to generating benchmarks from application traces. Ertvelde et al. utilize program slicing to generate benchmarks that preserve application performance characteristics while hiding its functional semantics [21]. This work focuses on resembling the branch and memory access behavior for sequential applications and may therefore complement our benchmark generator for parallel applications. Shao et al. designed a compiler framework to identify communication patterns for MPI-based parallel applications through static analysis [22], and Zhai et al. built program slices that contain only the variables and code sections related to MPI events and subsequently executed these program slices to acquire communication traces [23]. Program slicing and static benchmark generation in general have a number of shortcomings relative to our run-time, trace-based approach: (a) Their reliance on inter-procedural analysis requires that *all* source code be available. This includes complete source code of an application along with the source codes of all its dependencies, such as libraries, which is often unrealistic. (b) They lack execution time information. (c) They cannot accurately handle loops with data-dependent trip counts ("**while not** converged **do**…"). (d) They produce benchmarks that are neither human-readable nor editable.

Wu et al.'s work of generating the Conceptual benchmark [24] is related to our work. ScalaTrace is used to collect the traces from application in their work. A trace traversal framework, which is similar to our traversal framework, is used to generate the source code in Conceptual, a domain specific language [25]. This language focuses on generating networking/communication benchmarks. This work does not generate all MPI calls but maps the MPI events from the trace to the corresponding combination of communication routines. The Conceptual language does not have the concept of "communicators" as in MPI. Thus, it cannot form the subsets of ranks based on a communicator. Since our work generates C code with MPI calls, it can translate all MPI events captured in the trace accurately. The Conceptual language does not have provisions like wildcard receives, thus generated code needs to be resolved for the source in the send and receive communication calls. This eliminates the non-determinism present in the source code but changes runtime behavior (and semantics) relative to internal MPI queues, which are used to buffer the receives until matching sends are encountered. In our work, we reproduce the non-determinism present in the original application, thus accurately preserving the behavior of the application. Our work generates lossless, accurate and human readable MPI communication calls in C source code from a single trace file obtained from ScalaTrace, which is easily portable to any platform, as opposed to Conceptual with the need to interpret Conceptual code, which more closely resembles trace replay.

Benchmaker [26] is a framework for synthetic benchmark generation combining microkernels based on performance metrics where weights can be used to control the "mix" of program characteristics during benchmark synthesis. BenchMaker1/2 [27] is a tool to synthetically generate bench-

marks that comes in two different versions. One version recursively expands the control-flow templates for loops and conditional execution and fills in their blocks with selected statements. Another version synthesizes existing microkernels in a weighted, compositional manner to create a larger benchmark. HBench [28], [29] is a synthetic benchmark generator for Java applications that combines microkernels based on observed application-side runtime metrics. Many other approaches to benchmark synthesis exist, ranging from generic models [30], [31] to numerous memory models (stack, reference, temporal/spatial density, memory reuse distance, locality space) [32], [33]. In contrast, our work focuses on auto-generation of benchmarks from real application instead of generating synthetic benchmarks.

VII. Conclusion

We have designed and implemented a novel communication benchmark code generator that generates benchmark code in C with MPI calls from communication traces. These traces are generated by ScalaTrace, a lossless and scalable framework to extract communication, I/O operations and execution time while abstracting away the computations. These benchmarks are human readable, compact, easy to generate and port. They also preserve the behavior of the original application in terms of execution time, communication volume and ordering of events. Furthermore, application code is obfuscated by our benchmark generation process, which allows auto-generated benchmarks of otherwise restricted / distribution-controlled applications to be released to the public. And such benchmarks can be generated and released more frequently due to the automated generation process so that benchmark releases can keep up more closely with rapid development cycles of full-scale applications.

Experimental results demonstrate the ability of our code generator to generate the communication benchmarks from codes of the NAS Parallel Benchmark Suite and Sweep3D. The obtained results show that the benchmarks accurately preserve not only application semantics but also overall execution time. We demonstrated cross-platform validation of our generated benchmarks by adjusting for different CPU speeds. We also showed that the lines of code were, on average, reduced within auto-generated benchmarks relative to the corresponding NAS codes and Sweep3D. We expect that for full-scale benchmarks, this reduction in lines of code is even more significant as our test codes were benchmarks themselves.

Overall, our benchmark generator can benefit application developers, communication researchers and HPC system designers. It may assist in performance analysis of software and hardware and can also ease migration of applications across different platforms.

References

[1] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sep. 1996.

[2] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski, "Scalable compression and replay of communication traces in massively parallel environments," in *International Parallel and Distributed Processing Symposium*, Apr. 2007.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *The International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, Fall 1991. [Online]. Available: citeseer.ist.psu.edu/article/bailey94nas.html

[4] H. Wasserman, A. Hoisie, and O. Lubeck, "Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications," *The International Journal of High Performance Computing Applications*, vol. 14, pp. 330–346, 2000.

[5] "MPI-2: Extensions to the message passing interface," July 1997.

[6] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of mpi applications with umpire," in *Supercomputing*, 2000, p. 51.

[7] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski, "Scalatrace: Scalable compression and replay of communication traces in high performance computing," *Journal of Parallel Distributed Computing*, vol. 69, no. 8, pp. 969–710, Aug. 2009.

[8] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz, "Preserving time in large-scale communication traces," in *International Conference on Supercomputing*, Jun. 2008, pp. 46–55.

[9] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 350–360, Jul. 1991.

[10] J. Marathe and F. Mueller, "Detecting memory performance bottlenecks via binary rewriting," in *Workshop on Binary Translation*, Sep. 2002.

[11] X. Wu, K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Probabilistic communication and i/o tracing with deterministic replay at scale," in *ICPP*, 2011.

[12] J. Vetter and M. McCracken, "Statistical scalability analysis of communication operations in distributed applications," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2001.

[13] X. Wu and F. Mueller, "ScalaExtrap: Trace-based communication extrapolation for SPMD programs," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2011.

[14] H. Brunst, H.-C. Hoppe, W. E. Nagel, and M. Winkler, "Performance optimization for large scale computing: The scalable VAMPIR approach." in *International Conference on Computational Science (2)*, 2001, pp. 751–760.

[15] V. Pillet, V. Pillet, J. Labarta, T. Cortes, and S. Girona, "PARAVER: A tool to visualize and analyze parallel code," in *Proceedings of the 18th Technical Meeting of WoTUG-18: Transputer and Occam Developments*, ser. Transputer and Occam Engineering, P. Nixon, Ed., vol. 44. Manchester, United Kingdom: IOS Press, Apr. 9–12, 1995, pp. 17–31.

[16] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the Open Trace Format (OTF)," in *International Conference on Computational Science*, May 2006, pp. 526–533.

[17] A. Knupfer, "Construction and compression of complete call graphs for post-mortem program trace analysis," in *International Conference on Parallel Processing*, 2005, pp. 165–172.

[18] M. Casas, R. Badia, and J. Labarta, "Automatic structure extraction from mpi applications tracefiles," in *Euro-Par Conference*, Aug. 2007.

[19] Q. Xu, R. Prithivathi, J. Subhlok, and R. Zheng, "Logicalization of MPI communication traces," Dept. of Computer Science, University of Houston, Tech. Rep. UH-CS-08-07, 2008.

[20] Q. Xu and J. Subhlok, "Construction and evaluation of coordinated performance skeletons," in *International Conference on High Performance Computing*, 2008, pp. 73–86.

[21] L. V. Ertvelde and L. Eeckhout, "Dispersing proprietary applications as benchmarks through code mutation," in *Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 201–210.

[22] S. Shao, A. Jones, and R. Melhem, "A compiler-based communication analysis approach for multiprocessor systems," in *In International Parallel and Distributed Processing Symposium*, 2006.

[23] J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng, "FACT: Fast communication trace collection for parallel applications through program slicing," in *Proceedings of SC'09*, 2009, pp. 1–12.

[24] X. Wu, F. Mueller, and S. Pakin, "Automatic generation of executable communication specifications from parallel applications," in *ICS*, 2011, pp. 12–21.

[25] S. Pakin, "The design and implementation of a domain-specific language for network performance testing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 10, pp. 1436–1449, Oct. 2007.

[26] A. M. Joshi, L. Eeckhout, and L. K. John, "The return of synthetic benchmarks," in *SPEC Benchmark Workshop*, Jan. 2008.

[27] J. Dujmović, "Automatic generation of benchmark and test workloads," in *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, ser. WOSP/SIPEW '10, 2010, pp. 263–274.

[28] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang, "The case for application-specific benchmarking," in *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, 1999, pp. 102–.

[29] X. Zhang and M. Seltzer, "Hbench:java: an application-specific benchmarking framework for java virtual machines," in *Proceedings of the ACM 2000 conference on Java Grande*, ser. JAVA '00, 2000, pp. 62–70.

[30] D. Ferrari, "On the foundations of artificial workload design," in *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1984, pp. 8–14.

[31] H. J. Curnow and B. A. Wichmann, "A synthetic benchmark," *Comput. J.*, vol. 19, no. 1, pp. 43–49, 1976.

[32] T. M. Conte and W. mei W. Hwu, "Benchmark characterization," *IEEE Computer*, vol. 24, no. 1, pp. 48–56, 1991.

[33] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2003.