Using Link-Time Optimization to Improve the Performance of MPI Programs*

Gregory R. Andrews Saumya K. Debray Benjamin W. Schwarz Matthew P. Legendre
Department of Computer Science
University of Arizona
Tucson, AZ 85721
{greg, debray, bschwarz, legendre}@cs.arizona.edu

Keywords: MPI Library, Parallel Computing, Performance, Link-Time Code Optimization

1 Introduction

A compiler translates a source program written in a high-level language into a relocatable object program for a specific machine. A linker combines object programs—from separately compiled parts of an application and from libraries—resolves symbolic references, and produces an executable binary program. A binary rewriting system (BRS) is a software system that transforms a binary program into a different but functionally equivalent binary program. One use of binary rewriting is to modify a program in order to improve some aspect of its performance, such as execution time.

A *link-time optimizer* is a BRS that optimizes object programs. A significant virtue of link-time optimization is that it can perform whole-program optimizations that cannot be done by compilers—e.g., those based on values of addresses, which are not known until link time—or that usually are not done by compilers—e.g., because library source code is not available or because interprocedural optimization is hard. In addition, a BRS is independent of the languages in which source programs are written.

This paper describes the design, implementation, and performance of PLTO, a link-time optimizer for Intel Pentium processors.[1] Our goal is *automatically* to improve the performance of parallel scientific programs that use an MPI library for communication and that execute on clusters of Pentium workstations (so-called Beowulf machines [2]). The main techniques that PLTO employs are (1) specializing the code in MPI library routines to account for exactly how they are used by an application program, and (2) inlining such routines into the application so as to avoid function call overhead and to eliminate redundant loads and stores. PLTO uses several additional kinds of optimizations—including constant propagation, dead-code elimination, and code layout—to achieve further performance improvements. These techniques are general—they can be used for any application, any implementation of the MPI library, or any other communication library.

Before developing PLTO, we examined characteristics of MPI programs and of the MPICH [14] and Lam [13] implementations of MPI to determine optimization opportunities. Section 2 summarizes what we learned. Section 3 describes the design and implementation of PLTO, our link-time optimizer. Section 4 describes the current status and performance of PLTO. The "alpha" version of PLTO is able to reduce execution time by over 10% for some applications. We expect to see better and more consistent performance soon as we refine and tune the implementation and add additional kinds of optimizations. Section 5 summarizes related work. Section 6 gives concluding remarks.

2 Characteristics and Performance of MPI Programs

To determine how much we might be able to improve the performance of MPI programs, we have written several applications—including multigrid, N-body, and LU decomposition—experimented with various MPI primitives, and implemented a hand-optimized version of the MPICH implementation of the MPI library. We have also examined and tested all the NAS benchmarks [17].

Parallel programs that use MPI have several attributes that lead to optimization opportunities. First, a program typically uses only a few different MPI functions. Second, many MPI function calls are inside loops that are executed

*This work was supported in part by the National Science Foundation under grants ASC-9720738, CDA-9500991, and EIA-0080123.

[1] PLTO is an acronym for Pentium Link-Time Optimizer. We pronounce it "Pluto."

```
static void Worker(int myid, int numIters, ... ) {
  double **grid, **temp;  /* allocate and initialize local grids */
  ...
  MPI_Barrier(MPI_COMM_WORLD);  /* wait for all workers to initialize */
  start_time = times();         /* read clock for start time */
  for (iters = 1; iters <= numIters; iters++) {
    if (below != -1)     /* exchange boundaries with neighbors */
        MPI_Send(&grid[stripSize][1], gridSize, MPI_DOUBLE,
                                   below, 0, MPI_COMM_WORLD);
    ...  /* conditionally do a second Send
    if (above != -1)
        MPI_Recv(&grid[0][1], gridSize, MPI_DOUBLE,
                             above, 0, MPI_COMM_WORLD, &status);
    ...  /* conditionally do a second Recv */
    ...  /* update local grid points */
  }
  MPI_Barrier(MPI_COMM_WORLD);   /* wait for all workers to finish */
  end_time = times();            /* read clock for end time */
  ...     /* calculate residuals and print results */
}
```

Figure 1: Code structure for communication using MPI libraries

hundreds of times. Third, many MPI function calls have the same arguments, and they are constant at run time. Fourth, the MPI library has a hierarchical organization with several levels of functions. Finally, many of the functions in the MPI library contain conditional and switch statements that depend on the (run-time constant) values of application-provided arguments.

As a simple example, consider an iterative relaxation method for a two-dimensional grid. Assume that the grid is divided into one-dimensional strips (blocks), that there is one worker process per processor, and that each worker is responsible for updating each strip of points. For this type of application, each worker process executes the type of code shown in Figure 1.[2] The values of grid points continually change, but the arguments to each call of `MPI_Send` and `MPI_Recv` are constant within each worker process. Hence, the conditional statements in the MPI library that access these arguments always take the same actions.

We can improve the communication performance of a program such as the one in Figure 1 by (1) using different communication primitives that permit communication and computation to be overlapped, thus masking communication delays, and (2) specializing the code in the MPI library to account for the exact way in which a program uses the library. The first change is under the programmer's control, but the second one is at best painful and at worst impossible for an application programmer to do.

The communication code in Figure 1 uses the simplest pair of MPI communication primitives: `MPI_Send` and `MPI_Recv`. The `MPI_Send` function is usually nonblocking; the `MPI_Recv` function is blocking. A more sophisticated combination—which is widely used in the NAS benchmarks—uses nonblocking send, `MPI_Isend`, and nonblocking receive, `MPI_IRecv`, and later waits for communication to be completed (`MPI_Wait` or `MPI_Waitall`). Using these functions avoids excess copying between the application and the MPI library and allows communication and computation to be overlapped, thus reducing the amount of time that an application is blocked waiting for messages to arrive. The MPI library also supports persistent communication, which allows the programmer to pre-allocate communication handles for sends and receives.

Whatever the choice of communication primitives, once we have a specific application program we can create a customized version of the MPI library that is specialized to handle the specific MPI function calls in the application. Consider again the sample program in Figure 1. The values of grid points continually change, but the arguments of *every* call of `MPI_Send` and `MPI_Recv` are the same. (Some of these are compile-time constants; some are not known until run-time but are constant for each call.) Consequently, we can simplify the MPI library code that gets executed as a result of each call. In particular, we can eliminate many conditional statements, such as those that check arguments

[2] We have not shown the main routine, which would initialize the MPI library before calling `Worker`. Timing of the main computational loop is done as in the NAS benchmarks by using barrier synchronization and then reading a system clock.

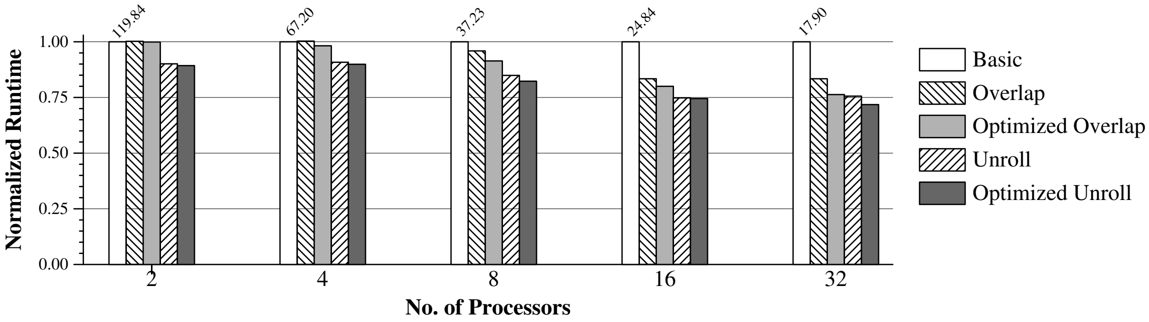


Figure 2: Performance Results for MPI Programs

for validity, and collapse switch statements. We can also determine the exact collection of internal library functions that get called—i.e., the call path through the library—and we can then inline all these functions into the top-level send and receive functions. This eliminates function-call overhead.

Figure 2 reports representative results for five different programs for Jacobi iteration.[3] The programs are (1) a basic program that uses `MPI_Send` and `MPI_Recv`, (2) a program that uses asynchronous send and overlaps communication and computation, (3) program 2 with a hand-optimized version of MPI, (4) an MPI program that unrolls the main computational loop and overlaps communication and computation, and (5) program 4 with a hand-optimized version of MPI. The programs were executed on various size clusters of 200 MHz Pentiums connected by a 100 Mb/sec Ethernet. The programs all use a grid size of 256 by 256 and 10,000 iterations of the main loop. The programs and both versions of the MPI library were compiled by the `gcc` compiler with the highest level of optimization "`-O3`."

The results in Figure 2 are normalized to 1.0 so that is easier to see the performance gains on the larger cluster sizes. The number at the top of each basic program column gives the total execution time in seconds for that program. As can be seen, the more sophisticated MPI programs take less time, with the unrolled-loop program leading to significant performance gains due to using two, two-dimensional arrays rather than one, three-dimensional array.[4] Moreover, using a version of MPI that was hand-customized as described above reduces the execution time by up to 10% relative to using the standard MPI library.

As shown in Figure 2, using an optimized version of the MPI library leads to noticeable performance improvements. However, one is obviously not going to hand-modify the library for every application program! Consequently, we are building PLTO—a Pentium Link-Time Optimizer—to automate the optimization process. PLTO is able to specialize library functions and inline function-call chains, as we did by hand. PLTO is also able to perform additional optimizations that are not possible to do by hand, such as register allocation, redundant load/store elimination, instruction scheduling, and code layout to improve instruction cache behavior.

3 PLTO: A Link-Time Optimizer

This section gives an overview of link-time optimization, summarizes how PLTO is implemented, and describes one of the key optimization techniques—value-based code specialization. The full paper will give more details.

3.1 Overview of Link-Time Optimization

Link-time optimization refers to the manipulation of object files resulting from compilation. These may range from an intermediate representation of the program that is optimized and then translated to executable code by the optimizing linker [6, 9], to conventional binaries augmented with special compiler annotations [24], to ordinary executables [5, 16, 22]. A link-time optimizer may also use *execution profiles*—i.e., summary information about the execution behavior of the program on sample inputs—to guide its optimization decisions. PLTO manipulates ordinary, statically-linked executables and uses profile information as described below.

Processing an object file involves reading it, decompiling it, constructing an internal representation of the program (e.g., in the form of a control flow graph), analyzing and transforming this internal representation to effect various

[3] We used a simple iterative scheme, because our purpose was to measure communication performance. Better iterative schemes, such as multigrid, are much more complex, but they use MPI in much the same way.

[4] Wang, et al. [25] report a similar improvement in a large Fortran program after converting matrices to vectors.

```
for (i = 0; i<n; i++) {
   straight-line code A;
   call foo(arg1, arg2);
   straight-line code B;
}
straight-line code C;
```

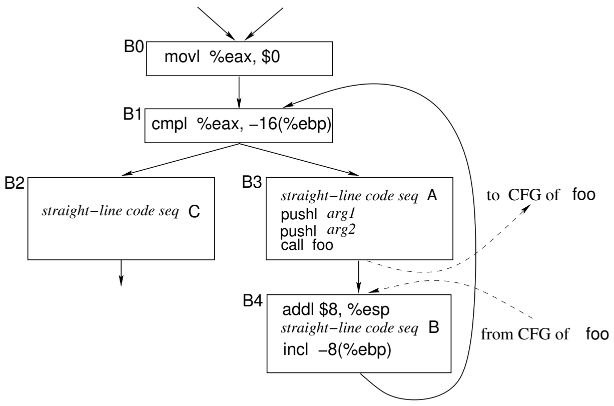


Figure 3: Simple loop and associated control flow graph

optimizations, and finally writing it back out as an object file. Much of this work is machine-dependent—e.g., object file formats, instruction representations and semantics, calling conventions, and register usage.

The overall optimization process for an object file *a.out* consists of three stages:

1. *Gathering low-level execution frequency profiles*. PLTO adds instrumentation code to the object file *a.out*. The resulting executable is then executed using representative inputs. This results in the creation of an execution profile for the program, which consists of execution counts associated with various program components. These counts are used by PLTO to identify hot spots in the code—i.e., the most frequently executed basic blocks (straight-line code fragments), loops, and functions.

2. *Gathering value profiles*. The second stage takes the input file *a.out* and the execution profile obtained from the previous step and adds instrumentation code to gather *value profiles*—i.e., information about the distribution of the values taken on by a register or variable at chosen program points. Gathering value profiles is expensive, so PLTO uses cost/benefit analysis and a machine cost model to determine what would be profitable to profile [26]. The resulting instrumented executable is then executed on (the same) representative inputs to obtain a set of value profiles.

3. *Code optimization*. The third stage of PLTO takes the input file *a.out*, together with its execution frequency and value profiles, and carries out a variety of optimizing transformations. PLTO uses the basic block and edge profiles to focus its optimization efforts and uses the value profiles to specialize the code for commonly occurring values.

3.2 Implementation of PLTO

The first step in manipulating a program is to read it in and disassemble it. The main technical problems here are distinguishing code from data and addresses from non-address constants. PLTO uses relocation information, supplied by the linker, to distinguish addresses from data.[5] Relocation information is also used to recover from disassembly errors arising from data embedded unexpectedly in the code sequence.

The main internal data structure in the implementation of PLTO is a *control flow graph* (CFG), which is constructed after the program has been disassembled. Figure 3 illustrates the CFG for a simple loop that contains a function call. Each node in the graph is a *basic block*—a sequence of regular instructions (e.g., load, store, arithmetic), possibly followed by by single branch instruction (e.g., jump, conditional branch, call, return).[6] The edges in a CFG connect basic blocks to reflect possible control transfers during execution. There is an edge from block B to block B' if execution can leave B and immediately enter B'. In Figure 3, for example, basic block B1 has two successors, reflecting the

[5] This is done by invoking the linker with additional options that instruct it to retain relocation information in the executable file.

[6] The CFGs manipulated by PLTO are a low-level representation of the program, and each instruction in a basic block is a machine instruction; Figure 3 uses instructions for the Intel x86 architecture. By contrast, compilers generally use higher-level CFGs based on intermediate-code instructions.

fact that control goes from B1 to block B3 if the conditional branch is taken, and to B2 otherwise. The dashed edge from block `B3` goes to the entry node of the CFG for function `foo` and represents the interprocedural control transfer resulting from the call to that function, while the dashed edge into block `B4` represents the interprocedural control transfer occurring due to the return from `foo`.

Once the CFG has been constructed, it is manipulated by PLTO in various ways. For the first two stages described in Section 3.1, PLTO inserts instrumentation code at appropriate points in the CFG; for the third stage, PLTO carries out a variety of program analyses and optimizations on the CFG. Finally, PLTO traverses the CFG one last time to produce a new, functionally equivalent object program.

The code optimization stage of PLTO proceeds as follows:

1. UNREACHABLE CODE ELIMINATION. Code that is guaranteed to be unreachable—arising from unreferenced library routines included by the linker—is eliminated.

2. STACK ANALYSIS. Because of the relatively small number of program-level registers in the Intel x86 architecture, code generated by compilers typically uses the runtime stack extensively, both for temporary storage of values and for passing arguments to functions. *Stack analysis* collects information about the stack usage of each function in the program. This information is subsequently used for optimizations such as inlining.

3. CONSTANT PROPAGATION. Constant values are propagated through the CFG, and any code simplifications that are enabled as a result are carried out. The constants that are propagated during link-time optimization by PLTO arise from two sources: constant addresses that are not known until after the program has been linked together, and constants that are propagated into library routines whose code is not available during compilation.

4. INLINING. A function is inlined into a particular call site if this is likely to yield reasonable performance improvements. This can occur either because the call site is executed frequently or because there are constant arguments at the call site that can be used to optimize the called routine. Inlining is implemented by cloning a copy of the function—i.e., a replica of the CFG for the body of the function—and inserting it at the call site. After inlining, the stack analysis is repeated, followed by another round of constant propagation.

5. LIVENESS ANALYSIS AND DEAD CODE ELIMINATION. After inlining has been done, PLTO does a liveness analysis of the registers, stack locations, and PSW (processor status word) bits. Based on the results of this analysis, instructions that are *dead*—i.e., whose results are not used—are eliminated. This allows us to eliminate, for example, `push` instructions corresponding to actual parameters after the values of those arguments have been propagated into an inlined function body.

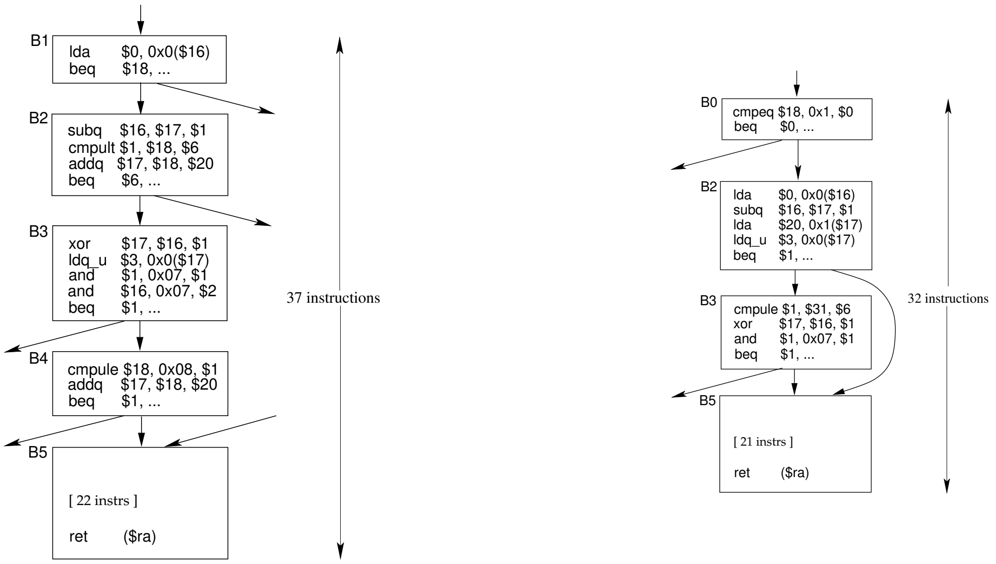
6. CODE LAYOUT AND INSTRUCTION SELECTION. After dead code elimination, PLTO traverses the CFG of the program to generate the final optimized executable. First, we use profile information to determine the actual layout of the various basic blocks, in a way that maximizes instruction cache utilization [19]; unconditional branches are inserted where appropriate at this stage to maintain correct control flow behavior. Once the layout has been determined, the offsets between different program points are known. The instruction sequence is then traversed to select the actual op-codes for branch instructions based on these offsets; for example, a branch to a nearby location may use a different op-code than one to a distant location. After this instruction selection step, the resulting program is written out.

3.3 Value-Based Code Specialization

A value profile for a variable at a program point is a (partial) description of the relative frequency of the different values it takes on whenever control reaches that point. Value profiles serve to identify situations where a variable takes on a particular value "most of the time." This information can then be used, in conjunction with a low-level cost model for the target processor, to decide whether to create a version of the code specialized for that value. The amount of code that is specialized can be a single basic block, a group of basic blocks, a loop, or a function. The decision depends on structural properties of the CFG. The details of this specialization process are discussed elsewhere [15].

Figure 4 shows an example illustrating value-based optimization for the function *memmove()* from the SPEC-95 benchmark *perl*.[7] The frequently executed portion of its control flow graph is shown in Figure 4(a). The third argument to this function is passed in register `$18`. The value profile for that register (based on the SPEC-95 training inputs)

[7] The assembly code shown here is for the Compaq Alpha architecture.



(a) Before specialization (b) After specialization

Figure 4: Value-based specialization: the function *memmove()* from the SPEC-95 benchmark *perl*

indicates that it has the value 1 over 70% of the time. Our specializer determines, based on the number of and type of instructions using the value of this register, that the benefits of specializing this code for the value 1 outweigh the cost of testing register `$18` for this value. The resulting specialized code is shown in Figure 4(b); the initial test of `$18` against the value 1 occurs in basic block B0. The overall effect of this optimization, in this case, is to reduce the length of the critical path through this function from 37 instructions to 32 instructions, a reduction of 13.5%; the speed of the application (on the SPEC-95 reference inputs) improves by about 5% as a result [15]. Conventional compile-time techniques cannot effect this optimization for two reasons: first, because the value of register `$18` cannot be guaranteed to be a constant; and second, because *memmove()* is a library routine that is not available for compile-time optimization.

As this example suggests, specialization based on commonly occurring values can significantly reduce the execution cost along the critical path of a program. Parallel programs that use MPI have these attributes, because they repeatedly invoke library routines with constant, or mostly constant, arguments. We just begun integrating value-profile-based specialization into PLTO. We have also generalized the notion of value profiling to allow the profiling of arbitrary expressions, not just program variables. This allows us to determine whether two pointers `x` and `y` are unaliased most of the time, i.e., point to different memory addresses, by profiling the value of the expression $x \neq y$. This is a significant advantage over the traditional compiler approach of relying on expensive and potentially imprecise alias analyses.

4 Current Status and Performance

Important Note to the Program Committee. Below we describe the current status and performance of PLTO. These results are preliminary and do not yet, to our chagrin, demonstrate consistent performance improvements. However, we just got PLTO up and running on full programs. We are submitting this extended abstract now because (1) we think the SC 2001 conference attendees will be interested in this work, (2) we know from the results in Section 2 that there is plenty of room for improving performance, (3) we know from earlier work on *alto* [16] that this technology works, and (4) we believe that we will have good results by the time the full paper is due.

As of mid-April we have a working version of PLTO. It implements all the optimizations described in Section 3.2, and is able to generate correct binary output for our own test programs as well as the NAS benchmarks. However, PLTO at present uses only edge profiles. We are currently integrating value profile generation and use into the system. The major difficulty doing so relative to our earlier work with *alto* on the Alpha architecture is that arguments to functions are passed on the stack in the Pentium architecture. Consequently, we have to be able to track the values of

Grid Size	*Procs*	*Execution Time* (sec)		*Speedup* (%)
		Optimized	Unoptimized	
256	2	17.03	19.22	11.39
	4	9.62	10.07	4.46
	8	7.56	8.69	13.00
512	2	13.27	13.96	4.94
	4	7.16	7.63	6.56
	8	NA	NA	NA
1024	2	5.14	5.09	−0.97
	4	2.64	2.62	−0.76
	8	1.37	1.35	−1.48

Figure 5: Execution time improvements due to PLTO

addresses as well as registers in order to conclude what stack locations are constant throughout a call chain. Finally, we have not yet had time to tune the implementation, especially the cost/benefit analysis that is critical to doing a good job of specializing the code.

The main changes that PLTO currently makes to a binary program are (1) propagate constant addresses and constant values in registers into and within functions, (2) inline frequently called functions, as determined by the edge profiles, (3) eliminate code for functions that are not called, and (4) move the most frequently executed blocks—again as determined by the edge profiles—to the front of the text segment so as to minimize instruction cache loads and conflicts.

Figure 5 presents results for a Jacobi iteration program that uses `MPI_ISend` and `MPI_Recv`, and that overlaps communication and computation by computing new values for interior points and then new values for edges. The programs were run on a cluster of 600 MHz Pentium II processors connected by a 1 Gigabit/second Ethernet. Each program was run five times using the MPICH implementation of the MPI library. The execution times reported in Figure 5 are the average of the middle three times. We do not have consistent results for 8 processors on the 512 by 512 grid, hence that entry is marked NA for not available.

In Figure 5, the column labeled Unoptimized reports total execution time for the program for various size grids and numbers of processors. The column labeled Optimized reports the time for the same program after it has been processed by PLTO. The column labeled Speedup gives the percentage difference between the execution time of the optimized program and the execution time of the original program.

As can be seen, the PLTO-optimized program is significantly faster when there is lots of communication relative to the amount of computation. In particular, the biggest improvement occurs for the 256 by 256 grid on 8 processors, which is the most communication-bound of the test programs. We do not yet know why the improvement is not consistent and smooth. As can also be seen, the PLTO “optimized” program actually ran slightly slower for the large grid sizes. Again, we do not yet know why, but the total execution time is small, so that is undoubtedly part of the explanation. Moreover, we do know from the output of a machine-level performance measurement tool called *rabbit* [20] that the PLTO-optimized binaries executed fewer load operations and had fewer cache misses and fewer branch mispredicts than the unoptimized binaries. This *should* be producing lower overall execution times.

In the next two months we will be measuring and analyzing numerous programs to gain better understanding of exactly what is happening, especially at the machine level. We will, of course, then tune PLTO based on what we learn. As noted above, we will also be integrating the value-based specialization into PLTO, which should lead to significant improvements based on our experience using that technique in the *alto* project.

5 Related Work (Summary)

We are not aware of any work that uses binary rewriting to optimize communication software for a specific application. Some researchers have investigated optimizing MPI communication libraries themselves, but this work is very different from ours. Kielmann *et al.* describe optimizations for wide-area systems [12]; Tang *et al.* use a thread-based implementation to reduce the overhead of MPI code [23].

A number of authors have discussed ways to utilize semantic information about the behavior of an application to reduce the amount of data communication that is performed (see, for example, [1, 4, 11]). These works are complementary to ours, in that they reduce the amount of data communication, whereas we optimize the communication libraries and reduce the overhead of communication.

There is a considerable body of work on link-time code optimization [5, 6, 9, 22, 24]. Much of this work relies on compiler support, whereas PLTO does not. Link-time optimizers that do not require such special compiler support include Spike [5] and OM [22]. None of the above systems carries out value-based code specialization.

Pioneering work on value profiling and value-profile-based code optimization is due to Calder *et al.* [3]. Some implementations of object-oriented languages also use a limited form of value-profile-based specialization in an attempt to mitigate the high cost of dynamically dispatched calls.

6 Conclusions

Computational scientists write a parallel program in order to solve a problem faster or to solve a larger problem in the same amount of time. If someone offered them a 10% performance improvement at the cost of doing a profiling run, they would no doubt jump at the offer. We have shown that there is room for this kind of improvement in programs that use the MPI library for communication. We have described a technology, link-time optimization, that can deliver it, and we have described an instance of this technology, PLTO, that we are building. Results to date indicate that the technology is promising, but they also raise questions for which we do not yet have the answers. Our immediate tasks are to find the answers and to improve PLTO so that it delivers the performance improvements that we know are possible based on our analysis of MPI programs and libraries and on our experience with *alto*.

References

[1] S. Amarasinghe and M. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 1993, pp. 126–138.

[2] The Beowulf Project. URL: `http://www.beowulf.org`.

[3] B. Calder, P. Feller, and A. Eustace. Value Profiling and Optimization. *Journal of Instruction-Level Parallelism* 1 (1999), 1–6.

[4] S. Chakrabarti, M. Gupta, and J.-D. Choi. Global Communication Analysis and Optimization. *Proc. SIGPLAN '96 Conference on Programing Language Design and Implementation*, June 1996, pp. 68–78.

[5] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin. Optimizing Alpha Executables on Windows NT with Spike. *Digital Technical Journal*, vol. 9 no. 4, 1997, pp. 3–20.

[6] C. S. Collberg. *Flexible Encapsulation*. Ph.D. Thesis, Lund University, 1992.

[7] D. R. Engler and T. A. Proebsting. DCG: An Efficient, Retargettable Dynamic Code Generation System. *Proc. Sixth Int. Conf. on Architectural Support for Prog. Langs. and Operating Systems* (ASPLOS-VI), November 1994, 263-272.

[8] T. Fahringer and E. Mehofer. Buffer-safe and cost-driven communication optimization. *Journal of Parallel and Distributed Computing*, vo. 57 (1999), 33-63.

[9] M. F. Fernández. Simple and Effective Link-Time Optimization of Modula-3 Programs. *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 103–115.

[10] D. Grunwald and R. Neves. Whole-Program Optimization for Time and Space Efficient Threads. *Proc. Seventh Int. Symp. on Arch. Support for Prog. Langs. and Operating Systems* (ASPLOS-VII), September 1996, 50-59.

[11] M. Kandemir, P. Bannerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. A Global Communication Optimization Technique Based on Data-Flow Analysis and Linear Algebra. *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 6, Nov. 1999, pp. 1251–1297.

[12] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MAGPIE: MPI's Collective Communication Operations for Clustered Operations for Clustered Wide Area Systems. *Proc. 1999 SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP'99), May 1999, pp. 131–140.

[13] LAM/MPI Parallel Computing. URL: `http://www.mpi.nd.edu/lam/`.

[14] MPICH—A Portable Implementation of MPI.
URL: `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[15] R. Muth, S. Watterson, and S. K. Debray. Code Specialization based on Value Profiles. *Proc. 7th. International Static Analysis Symposium* (SAS 2000), June 2000, pp. 340–359. Springer LNCS, vol. 1824.

[16] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere. `alto` : A Link-Time Optimizer for the DEC Alpha. *Software—Practice and Experience* **31**:67–101, Jan. 2001.

[17] The NAS Parallel Benchmarks. URL: `http://www.nas.nasa.gov/Software/NPB/`.

[18] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Trans. on Computer Systems*, vol. 18, no. 1, February 2000, 37-66.

[19] K. Pettis and R. C. Hansen, "Profile-Guided Code Positioning", *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990, pp. 16–27.

[20] Rabbit: A Performance Counters Library for Intel/AMD Processors and Linux. Scalable Computing Laboratory, Ames Laboratory, U.S. D.O.E., Iowa State University. Oct. 2000.
URL: `http://www.scl.ameslab.gov/Projects/Rabbit`.

[21] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary Translation. *Communications of the ACM*, vol. 36, no. 2, Feb. 1993, pp. 69–81.

[22] A. Srivastava and D. W. Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Languages*, pp. 1–18, March 1993.

[23] H. Tang, K. Shen, and T. Yang. Program Transformation and Runtime Support for Threaded MPI Execution on Shared-Memory Machines. *ACM Transactions on Programming Languages and Systems*, vo. 22, no. 4, July 2000, pp 673–700.

[24] D. W. Wall. Global Register Allocation at Link Time. *Proc. SIGPLAN '86 Symposium on Compiler Construction*, July 1986, pp. 264–275.

[25] P. Wang, D. Katz, and Y. Chao. Optimization of a parallel ocean general circulation model. *Proc. Supercomputing '97*, November 1997.

[26] S. A. Watterson and S. K. Debray, Goal-Directed Value Profiling. *Proc. 2001 International Conference on Compiler Construction* (CC 2001), April 2001 (to appear).