Automatically Adapting Programs for Mixed-Precision Floating-Point Computation

Michael O. Lam, Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland, College Park
Email: {lam, hollings}@cs.umd.edu

Bronis R. de Supinski, Matthew P. LeGendre
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Email: {bronis, legendre1}@llnl.gov

ABSTRACT

As scientific computation continues to scale, efficient use of floating-point arithmetic processors is critical. Lower precision allows streaming architectures to perform more operations per second and can reduce memory bandwidth pressure on all architectures. However, using a precision that is too low for a given algorithm and data set leads to inaccurate results. In this paper, we present a framework that uses binary instrumentation and modification to build mixed-precision configurations of existing binaries that were originally developed to use only double-precision. This framework allows developers to explore mixed-precision configurations without modifying their source code, and it enables automatic analysis of floating-point precision. We include a simple search algorithm to automate identification of code regions that can use lower precision. Our results for several benchmarks show that our framework incurs low overhead (less than 10X in most cases). In addition, we demonstrate that our tool can replicate manual conversions and suggest further optimization; in one case, we achieve a speedup of 2X.

1. INTRODUCTION

"Floating-point" is a method of representing real numbers in a finite binary format. It stores a number in a fixed-width field with three segments: 1) a sign bit (b), 2) an exponent field (e), and 3) a significand (or mantissa) (s). The stored value is $(-1)^b \cdot s \cdot 2^e$. Floating-point was first used in computers in the early 1940's, and was standardized by IEEE in 1985, with the latest revision approved in 2008 [26]. The IEEE standard provides for different levels of precision by varying the field width, with the most common widths being 32 bits ("single" precision) and 64 bits ("double" precision). Figure 1 graphically represents these formats.

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.

Double-precision arithmetic generally results in more accurate computations, but with several costs. The main cost is the higher memory bandwidth and storage requirement, which are twice that of single precision. Another cost is the reduced opportunity for parallelization, such as. on the x86 architecture, where packed 128-bit XMM registers can only hold and operate on two double-precision numbers simultaneously compared to four numbers with single precision. Finally, some architectures even impose a higher cycle count (and thus also energy cost) for each arithmetic operation in double precision. In practice, it has been reported that single-precision calculations can be 2.5 times faster than corresponding double-precision calculations, because of the various factors described above [19].

As high-performance computing continues to scale, these concerns regarding precision, memory bandwidth, and energy usage will become increasingly important [15]. Thus, application developers have powerful incentives to use a lower precision wherever possible without compromising overall accuracy. Unfortunately, the numerical accuracy degradation of switching to single precision can be fatal in some cases, and many developers simply choose the "safe option" of sticking with double precision in all cases.

Many computational scientists are investigating another option: "mixed-precision" configurations. With these schemes, certain parts of the program use double-precision operations to preserve numerical accuracy while the rest of the program uses single-precision operations to increase speed and save memory bandwidth. Thus, the developer can still reap some of the benefits of single precision while maintaining the desired level of accuracy. However, building these mixed-precision configurations can be difficult. This task either requires extensive knowledge of numerical analysis and the problem domain or many trial-and-error experiments.

In this paper, we present a framework that uses automatic binary instrumentation and modification to build mixed-precision versions of existing binaries that were originally developed to use double precision only. Our framework allows developers to experiment with mixed-precision configurations easily, without modifying their source code. We also present a simple search algorithm to find a mixed-precision variation of the program automatically.

Our results for several benchmarks show that our framework is effective and incurs low overhead. In most cases, the

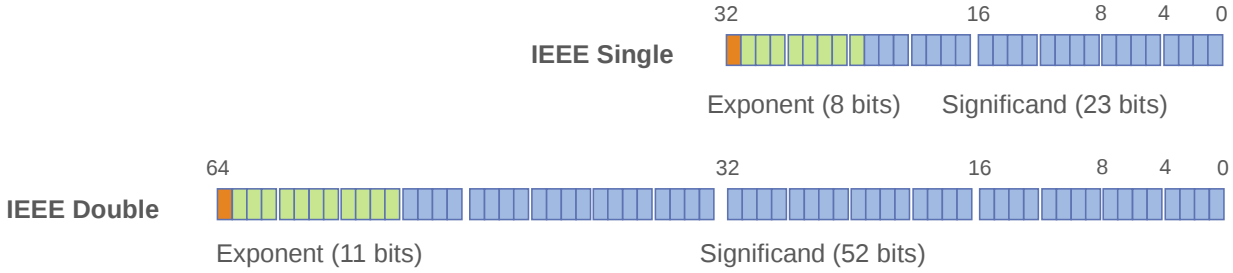

Figure 1: IEEE standard formats

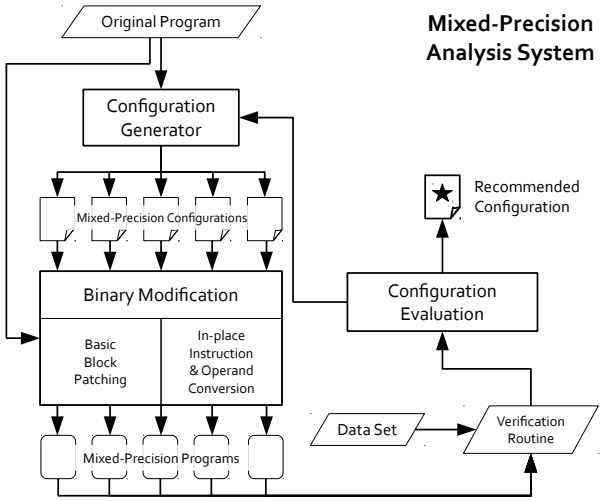

Figure 2: Mixed-precision analysis system overview

overhead is less than 10X, which is several orders of magnitude lower that existing floating-point analysis tools. For several of these benchmarks, the tool found that over 20% of all floating-point operations could be replaced with their single-precision equivalents. We also tested the AMG microkernel and found that the entire kernel could be replaced with single-precision arithmetic, resulting in a speedup of nearly 2X. In addition, our tool was able to replicate a manual conversion of SuperLU from double precision to single precision, and we show that loosening the error bounds can increase the amount of the program that can be replaced with single precision arithmetic.

The rest of the paper is arranged as follows. First, we describe our approach and implementation in Section 2. Second, we relate our evaluation and results in Section 3. Finally, we briefly enumerate related work in Section 4.

2. OUR APPROACH

We introduce a comprehensive system for analyzing an application's precision requirements. Given a representative data set and a verification routine, this system builds multiple mixed-precision configurations of the application and evaluates them, choosing the one that promises the greatest benefit in terms of speedup and easing of memory bandwidth pressure. Figure 2 shows an overview of this system.

A key system component is our framework for automatic binary-level mixed-precision configuration of floating-point programs. Section 2.1 describes our format for storing floating-point configurations and our GUI editor for building and visualizing the resulting program. This framework is based on the Dyninst library [8] for binary code patching and uses XED from the Intel Pin library [37] for instruction parsing. Section 2.2 describes the current, simple search algorithm, which consists of a breadth-first search through the program structure to find the coarsest granularity at which each part of the program can be replaced by single precision while still passing a user-provided verification routine. We describe in Section 2.3 our novel technique that inserts single-precision operations to replace double-precision operations. Our technique modifies the 64 bits that hold the double-precision representation to simulate single precision, storing a flag in the extra bits to indicate the presence of a modified number. This approach does not fully realize the benefits of using single precision format but allows us to identify when single precision preserves sufficient accuracy. The source code can then be transformed to realize the full benefits for situations that we identify as safe. Finally, as we describe in Section 2.4, we use basic block patching to extract the original double-precision floating-point instructions and replace them with new code, contained in binary "snippets."

2.1 Configuration representations

Our "precision configurations." provide a method of communicating which parts of a program should be executed in single precision and which parts should be executed in double precision. A configuration is a series of mappings:

$$p \rightarrow \{single, double, ignore\}$$

The mappings involve all $p \in P_d$, where P_d is the set of all double-precision instructions in program P. Since there are natural parent-child containment aggregations in program structure (i.e., instructions are contained within basic blocks, which are contained within functions), the configuration allows decisions to be made about these aggregate structures, overriding any decisions about child members. The configuration controls the analysis engine as follows:

- If the mapping for p_i is *single*, then the opcode of p_i will be replaced with the corresponding single-precision

```
    FUNC01: main()
      BBLK01
s       INSN01: 0x6f45ce "addsd %xmm1, %xmm0"
d       INSN02: 0x6f45d7 "mulsd %xmm2, %xmm1"
s       INSN03: 0x6f45da "subsd %xmm1, %xmm0"
d       INSN04: 0x6f45e8 "mulsd %xmm2, %xmm1"

    FUNC02: solve()
      BBLK02
s       INSN05: 0x6f7abe "addsd %xmm1, %xmm0"
s       INSN06: 0x6f7ac6 "addsd %xmm1, %xmm0"
s       INSN07: 0x6f7aca "addsd %xmm1, %xmm0"
d       INSN08: 0x6f7ad3 "mulsd %xmm1, %xmm2"
s       INSN09: 0x6f7ada "addsd %xmm1, %xmm0"
      BBLK03
d       INSN10: 0x6f7aee "mulsd %xmm1, %xmm2"
d       INSN11: 0x6f7af4 "subsd %xmm1, %xmm0"
d       INSN12: 0x6f7af9 "mulsd %xmm1, %xmm2"

s   FUNC03: split()
      BBLK04
s       INSN13: 0x6f8248 "subsd %xmm1, %xmm0"
d       INSN14: 0x6f824c "divsd %xmm2, %xmm1"
d       INSN15: 0x6f824f "divsd %xmm2, %xmm1"
```

Figure 3: Example replacement analysis configuration file

opcode, the inputs will be cast to single precision before the operation, and the result will be stored as a replaced double-precision number with a flag.

- If the mapping for p_i is *double*, then the opcode of p_i will not be replaced, the inputs will be cast to double precision before the operation, and the result will be stored as a regular double-precision number.
- If the mapping for p_i is *ignore*, then instruction p_i will be ignored entirely, which is useful for flagging unusual constructs like random number generation routines.

We use a simple, human-readable exchange file format (Figure 3 shows an example) to store these configurations. The file is plain text and lists the program's functions, basic blocks, and instructions, using indentation to improve readability. The initial list of these structures is easily generated using a simple static analysis that traverses the program's control flow graph. The first column contains flags such as "d" (double precision), "s" (single precision), or "i" (ignore) that control the precision of the code structures during instrumentation. The format supports simple toggling of larger aggregate structures like functions. If an aggregate entry has a flag in the first column, it overrides any flags specified for its children; if the aggregate entry has no flag, each child requires a flag individually. Our automated search system automatically generates new configuration files, which we then pass to our instrumentation system.

In the example configuration of Figure 3, certain instructions from each function have been selected for replacement with single precision. In addition, the function `split()` has a single-precision replacement flag, overriding the individual flags of all instructions in that function.

We also built a GUI (shown in Figure 4) that displays a tree corresponding to the program structure, allowing a developer to adjust a configuration manually without having to edit a lengthy text file. The GUI also allows the developer to visualize the results of our automatic search to understand what part of the code can be changed to single precision. If debug information is available, the GUI can also present a view that shows the corresponding source code location for a particular instruction. This capability aids in the conversion of particular code regions to single precision.

2.2 Automatic breadth-first search

We developed a simple automatic search system that attempts to replace as much of the program as possible using a breadth-first search through the entire program's configuration space. There are 2^n total possible configurations to test, where n is the number of floating-point instructions in the program. Since evaluating each test configuration requires a full program run, exhaustively testing every configuration is not feasible. Our bread-first search strategy represents our initial attempt to exploit some structure in these configurations for a faster search. Improving the search strategy is an ongoing area of research.

Our search maintains a work queue of possible configurations, testing them one by one and adding to the final configuration any individual configurations that pass the application-defined verification process. This process is highly parallelizable, and the system can launch many independent tests if cores are available. The search first generates configurations for each module in the program. Each configuration replaces the entire corresponding module with single precision, leaving the rest of the program in double precision. If any of these module-level configurations fail to pass verification, the routine begins to descend recursively through the program structure, testing function-level configurations before continuing to basic blocks and finally individual instructions as necessary. The recursion terminates when any structure (module, function, or block) is replaced and passes verification, or when the search tests an individual instruction (which cannot be further subdivided). The search can also be configured to stop at basic blocks or functions, allowing for faster convergence with coarser results.

Our system includes two simple optimizations that enhance the search speed and help the search converge faster. The first optimization involves using a binary search to break up large functions and basic blocks into two equally-sized intermediate partitions, rather than adding them all immediately as individual configurations when the parent configuration fails. This reduces the amount of configurations that must be tested when there are a large number of replaceable sections sprinkled with a few non-replaceable sections. The second optimization prioritizes configurations based on an initial profiling run; the configurations that replace the most frequently executed instructions are tested first. This allows the search to rule out large replacements more quickly and to provide faster preliminary results to the user.

After performing a brute-force breadth-first search through the program's structure, our tool will have found the coarsest granularity at which each part of the program can successfully be replaced by single precision. The routine then assembles a "final" configuration by taking the union of all previously-found successful individual configurations. This configuration is also tested automatically, although it may not pass verification without further changes because the precision levels of various instructions are not independent. In other words, decreasing precision in one part of a program may impact the sensitivity of other portions. However, the final configuration serves as an interesting starting point for the developer to investigate since it represents a rough indicator of how much of the original program can be individually replaced by single precision while maintaining the original desired level of accuracy.

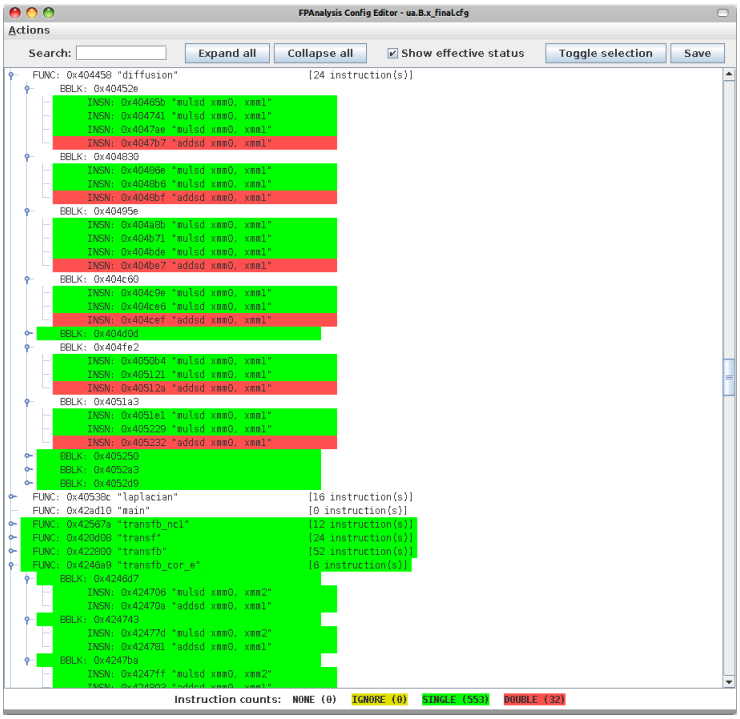

Figure 4: Graphical configuration editor, viewing a configuration for one of the NAS benchmarks

2.3 Binary modification

Our strategy for implementing mixed-precision configurations in existing binaries is to replace some double-precision instructions selectively with their single-precision equivalents, and to replace their double-precision operands with their single-precision equivalents in memory. Of course, after our system produces a final configuration, a programmer would then need to convert these operations to single precision in the original code. Recall that the purpose of our system is to identify regions of a program that can be computed in single precision, not to perform the code transformation in the original source language.

These narrowing conversions (double precision to single precision) allow our analysis to store the new (lower-precision) value in the same location as the old value. Thus, the 32 bits of the new single-precision value are stored in the lower 32 bits of the original 64-bit double-precision register or memory location. The remaining high 32 bits are set to a specific bit pattern (`0x7FF4DEAD`) [1] to indicate to the analysis that this value has been replaced. Figure 5 illustrates this process. This technique works for single values as well as "packed" floating-point values in 128-bit XMM registers.

To implement a replaced instruction, our framework can insert a streamlined "binary blob" snippet of machine code instructions. This snippet checks the operands (replacing them if necessary) and runs the original instruction in the desired precision. The desired precision is dictated for every floating-point instruction in the original program by a configuration file, described in Section 2.1. These new machine code instructions are generated by a simple mini-compiler, which implements routines for building flag tests and rewriting instruction opcodes in lower precisions. Figure 6 shows the template for these snippets in the case where we perform the instruction in single precision.

Since most of the snippet operations are integer instructions, the snippets impose a minimal overhead, and the downcast operation is performed only when the input has not already been replaced. In order to avoid hard-to-find synchronization bugs or writing to unwritable memory, the analysis copies any memory operands into a temporary register, and modifies the replaced instruction to use only register operands. In addition, once we replace any instruction with its single-precision equivalent, we must replace all floating-point instructions with our snippets, even the ones that are to be performed in double-precision. This change is necessary even if we do not replace a particular instruction with its single-precision equivalent, because we must add a check and possible upcast in case any of the incoming operands were replaced with single precision by an earlier operation. This requirement provides the benefit that anything that our analysis misses causes a crash, which is much easier to debug than mis-rounded operations.

2.4 Basic block patching

To modify the binary and insert our code snippets, we use Dyninst's CFG-patching API. This API allows us to split the original program's basic blocks at arbitrary points and rearrange the edges between blocks. To insert our code in the place of an instruction, we first split the basic block that contains the instruction into three blocks: 1) any instructions before the original instruction, 2) the original instruction,

[1] We choose this value for two reasons. The first four hex digits (`0x7FF4`) encode a NaN, ensuring that the program never silently propagates incorrect values. The second four hex digits (`0xDEAD`) form a common human-readable value that is easy to spot in a hex dump.

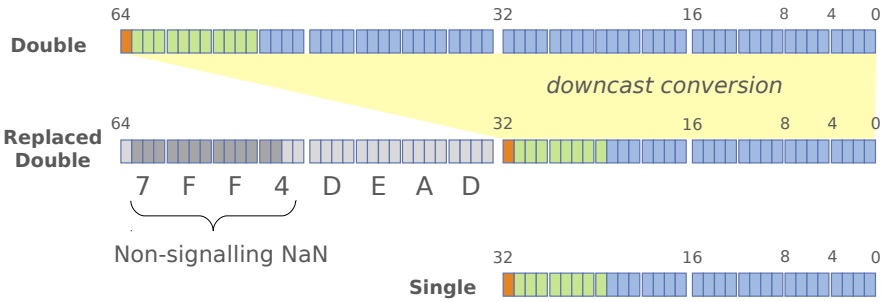

Figure 5: In-place downcast conversion and replacement

```
  push %rax
  push %rbx

  <for each input operand>
    <copy input into %rax>
    mov %rbx, 0xffffffff00000000
    and %rax, %rbx                # extract high word
    mov %rbx, 0x7ff4dead00000000
    test %rax, %rbx               # check for flag
    je next                       # skip if replaced
    <copy input into %rax>
    cvtsd2ss %rax, %rax           # down-cast value
    or %rax, %rbx                 # set flag
    <copy %rax back into input>

next:
  <next operand>
  pop %rbx
  pop %rax
  <replaced operand>      # e.g. addsd => addss

  <fix flags in any packed outputs>
```

Figure 6: Single-precision replacement template

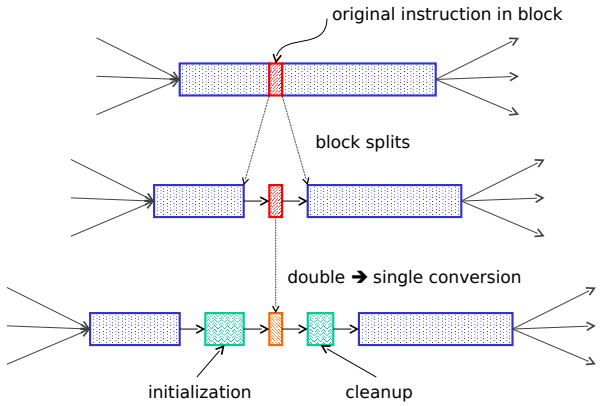

Figure 7: Basic block patching

and 3) any instructions after the original instruction. This allows us to insert our own code and re-arrange the edges from the surrounding parts of the original basic block to point to our new code instead of the original instruction. Figure 7 illustrates this process.

After performing the patching process, we use Dyninst's binary rewriter to create a new executable with the replaced code. The rewriter can also output modified shared libraries, allowing us to instrument and to modify functions in external dependencies. Thus, we can analyze third-party libraries even if the source code is not available.

2.5 Future optimizations

We have several avenues to improve our current system. First, we could reduce the runtime overhead by streamlining the machine code that is emitted, in order to produce more compact and efficient snippets. Second, we observe that in many cases the implementations of transcendental functions like sine, cosine, and logarithms contain lookup routines or bitwise manipulation for speed. Adding special handling for these functions improves performance and increases the fraction of the instructions in the original program that can be replaced with single precision. Third, static data flow analysis could improve overheads by detecting instructions that never encounter replaced double-precision numbers under a given configuration, and thus would not need to be replaced with a double-precision snippet. Finally, we could streamline the search algorithm to converge more quickly on the optimal configuration by adapting more conventional search heuristics rather than doing a simple breadth-first search.

3. RESULTS

In this section, we present results using our analysis technique. First, we show that the runtime overhead on several standard benchmarks is feasible for real workloads. Second, we show how our technique can inform developers about the floating-point behavior of their program and its sensitivity to roundoff error. Third, we show how our analysis can inform floating-point transformations for runtime speedup. Since most aspects of identifying mixed precision code regions amount to a single-node optimization, we confined our experiments to single nodes.

3.1 Correctness and runtime overheads

We first verified the correctness of our replacement on several NAS benchmarks [5] by manually converting the codes to use single precision and comparing the outputs to that of the instrumented version. The final results were identical, bit-for-bit, indicating that the instrumented versions were performing the exact same operations as the manually-converted versions of the original programs.

As a part of this verification, we developed a small script that attempted an automatic translation of Fortran source code to use single precision instead of double precision. How-

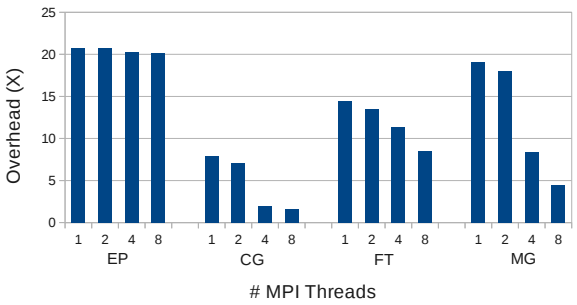

Figure 8: NAS MPI scaling results

Benchmark	Overhead
ep.A	3.4X
ep.C	5.5X
cg.A	3.4X
cg.C	4.5X
ft.A	4.2X
ft.C	7.0X
mg.A	5.8X
mg.C	14.7X

Figure 9: NAS benchmark overhead results

ever, we still had to tweak some files by hand, since some source files were generated by auto-configuration scripts. In some cases, while examining the results of the comparison, we found errors in our manual conversion, which needed to be corrected before the results matched. This process demonstrates that even by itself, the whole-program replacement routine is valuable as a complete automation of a error-prone manual (or semi-automated) process.

To examine the overhead of our tool, we looked at intra-node scaling by replacing all instructions with double-precision snippets. This transformation does not affect the semantics or results of the program, but shows how much overhead our inserted code causes in the base case in which it makes no conversions. Figure 8 shows the runtime overhead results from these experiments with Class A versions of the benchmarks. In general, the overall overhead decreases as the number of threads on a single core increases. Figure 9 shows specific runtime overheads from individual configurations for Class A and C inputs.

All benchmarks were MPI versions compiled by the Intel Fortran compiler with -O2 optimization enabled. Tests were performed on a distributed cluster, each node with twelve Intel Xeon 2.8GHz cores and 24 GB memory running 64-bit Linux. Each run used eight cores. The overhead was calculated as the ratio between the instrumented and original execution user CPU times as reported by the "time" command. In most cases, these overheads are under 20X, making this technique viable for test and trial runs on real data. In particular, these overheads are two orders of magnitude below those reported by the runtime cancellation detection tool [6] mentioned in the related work section, which range from 160X to over 1000X.

We also ran our automatic mixed-configuration search on these NAS benchmarks. For simplicity and speed of execution, we used the single-threaded versions of the benchmarks in this experiment. For each benchmark, we tested two input set sizes: class W and class A. All benchmarks were compiled by the Intel Fortran compiler with optimization enabled, and the tests were performed on the same cluster as the overhead experiments.

The benchmarks provided a wide range of replacement results. The percentages in Figure 10 indicate how sensitive the benchmark is to the precision level used. The columns contain the number of instructions that were candidates for replacement, the total number of configurations tested, the percentage of instructions replaced with single precision (measured statically), the percentage of instruction executions replaced (measured dynamically at runtime), and the verification result of the final composed configuration.

In nearly all cases, the system tested fewer configurations than there were candidates, showing that our techniques for pruning the search space are effective.[2] Some benchmarks (such as CG and FT) seem to be highly sensitive, since only a very low percentage of instruction executions can be replaced. Others seem to hold more promise for building a mixed-precision version. The ones that fail the final verification illustrate that using the simple union of all individually passing instructions does not automatically guarantee a passing configuration. This observation suggests that a second search phase may be useful, to determine the largest subset of individually-passing instruction replacements that may be composed to create a passing final configuration.

3.2 AMG Microkernel

To conduct an end-to-end test of our tool and a subsequent code modification by a programmer, we looked for a program that could run entirely in single precision, in order to simplify the manual conversion process. It was also necessary that the program include a verification routine that could be analyzed by our tool. The Algebraic Multi-Grid microkernel [1], which performs the critical sections of a multigrid solver, met our needs. For our experiments, we used 5,000 iterations on eight cores.

As expected, our system verified that the kernel could be replaced with single precision. The overhead of our analysis was only 1.2X for this benchmark, with all instructions replaced by single precision. We verified the results by manually converting the entire program to single precision and re-compiling. This conversion includes changing the verification routine to single precision, but in some circumstances this change is acceptable. In this instance, the adaptive nature of the multigrid method corrects for numerical inaccuracy by iterating to increasingly accurate results. For the microkernel, we observed a user CPU time decrease from 175.48s for the double-precision version to 95.25s for the single-precision version, a nearly 2X speedup.

3.3 SuperLU Linear Solver

To evaluate our tool further, we use a test program that is already implemented in both single- and double-precision versions, the SuperLU [14] general purpose linear solver. LU

[2]The one exception (SP) had many instructions in the final configuration that sequentially alternated between single and double precision, causing our search engine to spend an inordinate amount of time searching individual instructions rather than aggregated structures, leading to tests of more configurations than actual candidates.

Benchmark	Candidates	Configurations Tested	Instructions Replaced Static	Instructions Replaced Dynamic	Final Verification
bt.W	6,647	3,854	76.2%	85.7%	fail
bt.A	6,682	3,832	75.9%	81.6%	pass
cg.W	940	270	93.7%	6.4%	pass
cg.A	934	229	94.7%	5.3%	pass
ep.W	397	112	93.7%	30.7%	pass
ep.A	397	113	93.1%	23.9%	pass
ft.W	422	72	84.4%	0.3%	pass
ft.A	422	73	93.6%	0.2%	pass
lu.W	5,957	3,769	73.7%	65.5%	fail
lu.A	5,929	2,814	80.4%	69.4%	pass
mg.W	1,351	458	84.4%	28.0%	pass
mg.A	1,351	456	84.1%	24.4%	pass
sp.W	4,772	5,729	36.9%	45.8%	fail
sp.A	4,821	5,044	51.9%	43.0%	fail

Figure 10: NAS benchmark results, showing the benchmark name, number of candidates for replacement, number of configurations tested, percentage of instructions replaced with single precision (measured statically), percentage of instruction executions replaced (measured dynamically at runtime), and the verification result of the final composed configuration

Threshold	Instructions Replaced Static	Instructions Replaced Dynamic	Final Error
1.0e-03	99.1%	99.9%	1.59e-04
1.0e-04	94.1%	87.3%	4.42e-05
7.5e-05	91.3%	52.5%	4.40e-05
5.0e-05	87.9%	45.2%	3.00e-05
2.5e-05	80.3%	26.6%	1.69e-05
1.0e-05	75.4%	1.6%	7.15e-07
1.0e-06	72.6%	1.6%	4.77e-07

Figure 11: SuperLU linear solver memplus results

decomposition and linear solving comprise some of the most computationally expensive portions of larger scientific codes. The SuperLU library performs such operations, and it includes a single-core linear solver example program that can be compiled to use either single- or double-precision (but not mixed-precision). The program also reports an error metric that is useful in comparing the sensitivity of various mixed-precision configurations.

For these experiments we used the "memplus" memory circuit design data set from the Matrix Market [7], which contains nearly 18K rows. The linear solver for this data set runs for around three seconds on our machine, which allows us to test many configurations. The single-precision manually recompiled version achieves a 1.16X speedup over the double-precision version, which is equivalent to a 150 MFlops improvement. The reported error for the double-precision version of the solver is 2.16e-12, and the reported error for the single-precision version is 5.86e-04.

To run an automated search on the linear solver program, we wrote a driver script that ran the program and compared the reported error against a predefined threshold error bound. Using an error bound just above the error returned by the single-precision version, our search tool found that 99.1% of the double-precision solver's floating-point instructions could be replaced by single-precision. This equated to 99.9% of all runtime floating-point operations. That percentage matches the precision profile of the single-precision version of the program, and shows that our tool can find all replacements inserted manually by an expert.

Figure 11 shows the results of running our search tool using various error thresholds. The general trend is that when the error threshold is stricter, the search finds fewer static or dynamic instructions that can be replaced. For this application, the error of the final run (using the union of all passing configurations) tends to be much lower than the threshold used during the search. Thus, our tool can help identify the areas of a program that are sensitive to roundoff error.

4. RELATED WORK

4.1 Backwards & forwards error analysis

The analysis efforts regarding floating-point representation and its accompanying roundoff error initially focused on manual backward and forward error analysis. This field was active as early as 1959 [11], with Wilkinson's seminal work in the area being published in 1964 [44]. This research was continued by others throughout the following decades up to the present time [28, 33, 29, 31, 36] and recently summarized by Goldberg and Higham [21, 24].

Forward error analysis begins at the input and examines how errors are magnified by each operation. Backward error analysis is a complementary approach that starts with the computed answer and determines the exact floating-point input that would produce it. Both techniques provide an indication of how sensitive the computation is, and how incorrect the computed answer might be. Computations that are highly sensitive are called *ill-conditioned*.

Higham [24] describes examples of these analyses for a variety of different numerical analysis problems. Unfortunately, the results of these analyses are difficult for a programmer to understand or to apply without extensive training or error analysis background.

4.2 Interval & affine arithmetic

Researchers have attempted to model the behavior of a floating-point program using a technique called "interval arith-

metic" [29, 30, 41], which represents every number x in a program using a range $\bar{x} = [x.lo, x.hi]$ instead of a fixed value. Arithmetic operations operate on these intervals, usually resulting in a wider interval in the result:

$$\bar{x} + \bar{y} = [x.lo + y.lo, x.hi + y.hi]$$

$$\bar{x} - \bar{y} = [x.lo - y.hi, x.hi - y.lo]$$

Unfortunately, regular interval arithmetic is not always useful due to the quick compounding of errors [2], and the difficulty of handling intervals containing zero [24]. In the worst case, division by zero will produce an invalid interval, or the interval will eventually expand to $(-\inf, +\inf)$, a result that is trivially correct but practically useless. Even in less extreme circumstances, however, the average-case error is rarely as bad as the worst-case, and so interval analysis by itself is usually of little value to programmers who are merely interested in the practical behavior.

Interval arithmetic was later improved by Andrade and others [2] with the concept of "affine arithmetic," replacing the ranges of interval arithmetic with a linear combination of error factors. In this scheme, a number x is represented as a first-degree polynomial $\hat{x}$:

$$\hat{x} = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + \cdots + x_n\epsilon_n$$

Affine representation preserves information about error independence, and allows some errors to cancel out others. This sharing indicates that the error came from the same input and will cancel out in the sum. Thus, the bounds for the result are tighter than those that would be obtained in standard interval arithmetic.

More recently, Goubault and Martel and others [13, 16, 17, 22, 35, 38, 39] have built abstract semantics and static analyses using affine arithmetic. These techniques, like any static analysis, are *a priori* and give conservative error estimates. In addition, recent work by Martel and others [40] describes a system that can perform program transformations to increase accuracy. These transformations involve rearranging operations according to well-known rules of floating-point arithmetic, rather than by adjusting the precision.

Unfortunately, the fact that they are static analyses also means that they are not dataset-sensitive, and still give conservative estimates that may not be useful. In addition, they require some manual tuning by the programmer, particularly with regards to the extent that loops are unrolled: more unrolling produces better answers but requires more lengthy analyses. These techniques also only work for a subset of language features (often excluding HPC-specific interests like MPI communication), and are usually limited to C programs. Finally, these techniques merely make observations about error, and do not aid the programmer in building mixed-precision variants of their code.

4.3 Manual mixed-precision applications

More recently, many researchers [4, 9, 10, 25, 34, 42, 43] have demonstrated that mixed precision can achieve the same results as using purely double-precision arithmetic, while being much faster and memory-efficient. They usually present linear solvers (particularly sparse solvers) as examples, showing how the majority of operations can be performed in single precision. These solvers have been applied

```
 1: LU ← PA
 2: solve Ly = Pb
 3: solve Ux_0 = y
 4: for k = 1, 2, ... do
 5:     r_k ← b − Ax_{k−1}  (*)
 6:     solve Ly = Pr_k
 7:     solve Uz_k = y
 8:     x_k ← x_{k−1} + z_k  (*)
 9:     check for convergence
10: end for
```

Figure 12: Mixed precision algorithm

to a wide range of problems, including fluid dynamics [3], lattice quantum chromodynamics [12], finite element methods [20], and Stokes flow problems [18]. Often, GPUs are cited as the target of these optimizations because of their streaming capabilities [3, 12, 19].

In the iterative algorithm shown in Figure 12, for example, only the steps marked with asterisks (lines 5 and 8) must be executed in double precision. The authors note that all $O(n^3)$ steps can be performed in single precision, while the double-precision steps are only $O(n^2)$. Thus, using mixed precision can yield significant performance and memory bandwidth savings. On the streaming Cell processor, for instance, the mixed-precision version performed up to eleven times faster than the original double-precision version. Even on non-streaming processors, they obtained a performance improvement between 50% and 80%.

Researchers in the field of computer graphics have also found that mixed-precision algorithms can improve performance [23]. By varying the number of bits used for graphics computations, they report speedups of up to 4X or 5X, with little or no apparent image degradation. They use fixed-point arithmetic, but the mixed-precision concepts are similar to floating-point. Unfortunately, these techniques do not automatically generalize to other domains because they involve very specialized vertex and lighting calculations.

In recent work, Jenkins and others [27] describe a novel scheme for reorganizing data structures by numerical significance. Their technique splits up a floating-point number into several chunks on byte boundaries. All pieces of corresponding significance are stored consecutively in memory for storage and I/O, and the original numbers are re-assembled only when needed for calculation. Thus, the developer can vary the precision of floating-point data during data movement by truncating the lower-precision blocks. In their experiments, they found that some applications can use as few as three bytes (24 bits) of floating-point data and retain an acceptable level of accuracy. This work focused on the I/O implications, however, and did not address the possibility of single-precision arithmetic. Their system also incurs overhead during data re-assembly.

4.4 Dynamic cancellation detection

In previous work, we developed a dynamic runtime analysis for detecting numerical cancellation [32], which is a loss of significant bits due to the subtraction of nearly identical numbers. Cancellation can be harmful if one of the numbers was subject to roundoff error from previous calculations. Our analysis can detect all cancellations above a given threshold, reporting aggregate results for each instruction as well as a sampling of detailed information about indi-

vidual cancellations. These results can inform the developer about the floating-point behavior of their program. However, since cancellation does not necessarily always indicate that harmful roundoff error has occurred, the results can be difficult to interpret and to apply. Other authors have built on this work and developed more heavyweight techniques that quantify the "badness" of individual cancellations [6] in an attempt to be more helpful to the developer, but the overhead of their techniques make them prohibitively expensive for full-scale data sets. None of these efforts directly contribute towards mixed-precision computation.

5. CONCLUSION

We have shown that automated techniques can build mixed-precision configurations of existing binaries that use double-precision arithmetic, and that such techniques can be used to search an application automatically for portions that can be replaced with single-precision arithmetic. We have presented benchmark overhead results that show the technique's feasibility, and have described two experiments that demonstrated a speedup gained by a conversion guided by our analysis. In the future, we plan to optimize our implementation further by streamlining the inserted code to reduce overhead and by improving the search algorithm to converge more quickly. This work improves the ability of high-performance floating-point application developers to make informed decisions regarding the behavior of their code and the necessary precision levels for various parts of their programs.

Acknowledgments

We wish to acknowledge and thank Andrew R. Bernat at the University of Wisconsin, Madison for his contributions to the code patching library contained in Dyninst, as well as Osni Marques at the Laurence Berkeley National Laboratory for supporting our use of the SuperLU library. This work is supported in part by DOE grants DOEN.DESC0002351 and DOEN.DESC0002616.

This article has been authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. (LLNL-CONF-635248).

6. REFERENCES

[1] ASC Sequoia Benchmark Codes. https://asc.llnl.gov/sequoia/benchmarks/. Accessed 21 September 2011.

[2] M. V. A. Andrade, J. a. L. D. Comba, and J. Stolfi. Affine Arithmetic, 1994.

[3] H. Anzt, B. Rocker, and V. Heuveline. Energy efficiency of mixed precision iterative refinement methods using hybrid hardware platforms. *Computer Science Research and Development*, 25(3-4):141–148, 2010.

[4] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating Scientific Computations with Mixed Precision Algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2008.

[5] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0, 1995.

[6] F. Benz, S. Hack, and A. Hildebrandt. A Dynamic Program Analysis to find Floating-Point Accuracy Problems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[7] R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra. The Matrix Market: A web resource for test matrix collections. In R. F. Boisvert, editor, *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137. Chapman & Hall, London, 1997.

[8] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14:317–329, 2000.

[9] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. Exploiting Mixed Precision Floating Point Hardware for Scientific Computation. In *High Performance Computing and Grids in Action*. IOS Press, 2008.

[10] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy. 2007.

[11] J. W. Carr. Error analysis in floating point arithmetic. *Communications of the ACM*, 2(5):10–16, May 1959.

[12] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi. Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. *Computer Physics Communications*, 181(9):30, 2010.

[13] F. De Dinechin, C. Q. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. *Proceedings of the 2006 ACM symposium on Applied computing SAC 06*, page 1318, 2006.

[14] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[15] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J. C. Andre, D. Barkai, J. Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick. The International Exascale Software Project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[16] S. P. Eric Goubault Matthieu Martel. Asserting the

Precision of Floating-Point Computations: A Simple Abstract Interpreter. *Programming Languages and Systems*, pages 287–306, 2002.

[17] C. F. Fang, R. A. Rutenbar, M. Püschel, and T. Chen. Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling. *Proceedings of the 40th conference on Design automation DAC 03*, page 496, 2003.

[18] M. Furuichi, D. a. May, and P. J. Tackley. Development of a Stokes flow solver robust to large viscosity jumps using a Schur complement approach with mixed precision arithmetic. *Journal of Computational Physics*, 230(24):8835–8851, Oct. 2011.

[19] D. Göddeke and R. Strzodka. Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):22–32, 2011.

[20] D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):221–256, Aug. 2007.

[21] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991.

[22] E. Goubault. Static Analyses of the Precision of Floating-Point Operations. *Static Analysis*, pages 234–259, 2001.

[23] X. Hao and A. Varshney. Variable-precision rendering. *Proceedings of the 2001 symposium on Interactive 3D graphics SI3D 01*, http:149–158, 2001.

[24] N. J. Higham. *Accuracy and Stability of Numerical Algorithms, Second Edition*. SIAM Philadelphia, 2002.

[25] J. D. Hogg and J. A. Scott. A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 37(2):1–24, 2010.

[26] IEEE. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, New York, Aug. 2008.

[27] J. Jenkins, E. R. Schendel, S. Lakshminarasimhan, D. A. Boyuka II, T. Rogers, S. Ethier, R. Ross, S. Klasky, and N. F. Samatova. Byte-precision level of detail processing for variable precision analytics. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 48:1—-48:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[28] T. Kaneko and B. Liu. On Local Roundoff Errors in Floating-Point Arithmetic. *J. ACM*, 20(3):391–398, 1973.

[29] W. Kraemer. A Priori Worst Case Error Bounds for Floating-Point Computations. *IEEE transactions on computers*, 47(7):750–756, 1998.

[30] W. Krämer and A. Bantle. Automatic Forward Error Analysis for Floating Point Algorithms. *Reliable Computing*, 7(4):321–340, Aug. 2001.

[31] T. I. Laakso and L. B. Jackson. Bounds for floating-point roundoff noise. *IEEE transactions on circuits and systems*, 41(6):424–426, 1994.

[32] M. O. Lam, J. K. Hollingsworth, and G. W. Stewart. Dynamic Floating-Point Cancellation Detection. In *WHIST '11*, 2011.

[33] J. L. Larson, M. E. Pasternak, and J. A. Wisniewski. Algorithm 594: Software for Relative Error Analysis. *ACM Transactions on Mathematical Software*, 9(1):125–130, Mar. 1983.

[34] X. S. Li, M. C. Martin, B. J. Thompson, T. Tung, D. J. Yoo, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, and A. Kapur. Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software*, 28(2):152–205, June 2002.

[35] M. D. Linderman, M. Ho, D. L. Dill, T. H. Meng, and G. P. Nolan. Towards program optimization through automated analysis of numerical precision. *Proceedings of the 8th annual IEEE ACM international symposium on Code generation and optimization CGO 10*, page 230, 2010.

[36] B. Liu and T. Kaneko. Error analysis of digital filters realized with floating-point arithmetic. *Proceedings of the IEEE*, 57(10):1735–1747, 1969.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[38] M. Martel. Propagation of Roundoff Errors in Finite Precision Computations: A Semantics Approach. *Programming Languages and Systems*, pages 159–186, 2002.

[39] M. Martel. Semantics-Based Transformation of Arithmetic Expressions. *Static Analysis*, pages 298–314, 2007.

[40] M. Martel. Program transformation for numerical precision. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 101–110, New York, NY, USA, Jan. 2009. ACM Press.

[41] P. L. Richman. Automatic error analysis for determining precision. *Communications of the ACM*, 15(9):813–820, Sept. 1972.

[42] Robert Strzodka and Dominik Goddeke. Mixed Precision Methods for Convergent Iterative Schemes. In *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures, May 2006*, 2006.

[43] Robert Strzodka and Dominik Goddeke. Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components. *IEEE Proceedings on Field-Programmable Custom Computing*, 2006.

[44] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Inc., 1964.