Tool Integration for Source-Level Mixed Precision

Michael O. Lam*†, Tristan Vanderbruggen†, Harshitha Menon†, Markus Schordan†
*James Madison University
lam2mo@jmu.edu
†Lawrence Livermore National Laboratory
{lam26,vanderbrugge1,harshitha,schordan1}@llnl.gov

ABSTRACT

Mixed-precision computing offers potential data size reduction and performance improvement at the cost of accuracy, a tradeoff that many practitioners in high-performance computing and related fields are becoming more interested in as workloads become increasingly communication-bound. However, it can be difficult to build valid mixed-precision configurations and navigate the performance/accuracy space without the help of automated tools. We present FloatSmith, an open-source, end-to-end source-level mixed-precision tuner that incorporates several software tools (CRAFT, TypeForge, and ADAPT) into an integrated tool chain.

I. INTRODUCTION

High-performance computing (HPC) applications extensively use floating point arithmetic operations, so using floating-point arithmetic operations efficiently is critical to achieve good performance. Modern computer architectures usually support multiple levels of precision as defined by the IEEE standard: 32 bits (single-precision), 64 bits (double-precision), 128 bits (quad-precision, usually implemented in software), as well as 16 bits (half-precision, increasingly common in accelerators). Higher precision may improve the accuracy of simulation results, but it usually results in an increase in application run time, energy consumption, and memory pressure. However, not all applications require higher precision. In order to take advantage of the performance gains and energy savings, applications should use lower precision when possible while maintaining the required accuracy. One approach is called *mixed-precision arithmetic*, which uses multiple levels of precision within the same application.

Manually identifying the variables that can be in lower precision and generating a mixed-precision version of the application is challenging. There have been many efforts to automate this process. Various static analysis tools [1], [2] have been proposed that use interval and affine arithmetic or Taylor series approximations to provide rigorous bounds on precision errors. However, they do not scale very well and thus have been applied to only very small benchmarks. There are also several dynamic search-based approaches [3], [4], [5], which evaluate different mixed precision configurations of the program to choose the best configuration that gives the most performance gains while satisfying some error-related criteria. The main disadvantage of these approaches is that the state space to explore is exponential in the number of variables, which makes these search-based approaches very time intensive.

We propose FloatSmith, an integrated tool chain, to automatically identify mixed-precision configurations that provide the most performance gains within the specified error threshold. FloatSmith is an integration of three different tools: 1) CRAFT, a testing-based search tool, 2) TypeForge, a compiler-based static analysis and code transformation tool, and 3) ADAPT, an instrumentation-based automatic differentiation tool for mixed precision error analysis. FloatSmith produces a source-level mixed precision version of the program, enabling programmers to analyze the required mixed-precision changes and also makes it easy to maintain different code versions. Our tool combines search-based dynamic approaches with compiler-based static analysis techniques and rigorous precision analysis to speed up the dynamic search process.

II. METHODS

FloatSmith integrates analysis and transformation tools into a tool chain of three tools to achieve mixed precision through source-to-source transformations:

1) Configurable Runtime Analysis for Floating-point Tuning (CRAFT[1]) [6], [3], [7] provides a general framework for floating-point program analysis. We leveraged the existing testing-based search framework to implement source-level tuning.
2) TypeForge is a tool that uses the ROSE compiler framework[2] to perform type substitution and code instrumentation on source code. It can change the types of variables, data members, and aggregate type variables. We use the type conversion to convert variables to lower precision, and we use the instrumentation capability to insert ADAPT function calls. We also use TypeForge to generate compiler-based type dependency information to (optionally) narrow the CRAFT search space to groups of variables that must be converted together.
3) Algorithmic Differentiation Applied to Precision Tuning (ADAPT[3]) [8] is a wrapper for the CoDiPack library for algorithmic differentiation [9] that adds floating-point precision tuning analysis. We use ADAPT to (optionally) narrow the CRAFT search space to variables that can be replaced according to the differentiation results.

[1] `http://github.com/crafthpc/craft`
[2] `http://www.rosecompiler.org`
[3] `http://github.com/LLNL/adapt-fp`

Figure 1 shows an overview of the integrated tool chain. We use TypeForge to extract a list of tunable variables (along with clusters or groups based on static type dependency analysis described below) and optionally to insert ADAPT instrumentation. If ADAPT is used, the user must also insert some pragma-based annotations to describe the program outputs and allowed error thresholds. The ADAPT-instrumented program runs and produces a mixed-precision recommendation that is guaranteed not to exceed the user-provided error threshold; however, it might not speed up the program.

Finally, CRAFT performs a variable configuration space search (optionally starting from the ADAPT results or the TypeForge variable clusters/groups) and attempts to find a mixed-precision configuration that both passes a user-defined representative workload and verification routine and achieves a speedup. If such a configuration is found, CRAFT reports the fastest one and provides the modified source code. Each configuration is built using TypeForge for type conversion, and the search process can parallelize naturally if multiple cores or nodes are available.

Currently, our framework only fully supports applications compatible with C++11, but that is due to a limitation in an ADAPT dependency. The rest of the framework should be compatible with any language for which there is a Rose compiler front end.

The remainder of this section discusses changes to the various components of our system that were required to enable the entire tool chain. These changes not only improved each individual tool but also enabled "more than the sum of all parts" emergent benefits when using all components.

A. CRAFT

CRAFT [6], [3], [7] is a general framework for floating-point program analysis with support for a variety of different kinds of analysis. We use the mixed-precision search system, which is implemented in Ruby and was originally designed to find double-precision machine code instructions that could be replaced by their single-precision equivalents. The primary change required for the tool chain was to add the notion of a program variable as a first-class object for tuning. We implemented this and added a new "variable mode" activated using a command-line switch ("`-V`").

One benefit of doing the search at the variable level with a source-to-source tool is that speedups can be verified empirically by compiling and running a mixed-precision configuration. Previously, the instrumentation overhead involved in building the configuration far outweighed any benefit of the precision replacement. In addition, the compiler did not have a chance to perform optimizations like vectorization that would have exposed performance improvements at lower precision. With source-level transformations neither of these problems manifest, and CRAFT can directly search for a speedup.

Thus, we pursued two metrics for measuring the quality of a configuration: 1) the total number of variables replaced (which is a metric that was used previously for instructions) and 2) the actual speedup achieved by the configuration.

A secondary change to CRAFT was the addition of several new search strategies. Because variables do not have as deep of a structural hierarchy as instructions ("function $\rightarrow$ variable" rather than "module $\rightarrow$ function $\rightarrow$ basic block $\rightarrow$ instruction"), the old hierarchical strategy was less useful. Thus, we added (or updated) new search strategies:

1) **Combinational** - This is a brute-force strategy that simply tries all potential combinations of variables. This strategy is not viable for more than a few variables because it will test $2^n - 1$ configurations given n variables (it does not need to try the configuration where zero variables are replaced). However, it is useful for establishing a baseline for comparison and for finding a global maximum replacement count and speedup for small numbers of variables.
2) **Compositional** - This strategy tries replacing every variable individually and then attempts to build better configurations using compositions of already-passing configurations. It does this by taking every passing configuration with k replacements and building new configurations by merging with previously-passing k-cardinality and 1-cardinality configurations (see Algorithm 1). This approach will generally find the global maximum replacement count and speedup, but it is not guaranteed to do so. However, it also does not necessarily try all possible combinations. In practice, it avoids areas of the search space dominated by non-replaceable variables and provides results that are sometimes globally optimal.
3) **Delta debugging** - This strategy is based on the algorithm described in the original Precimonious paper [4] and used for comparison in other recent work [10], [11]. It uses a binary search on the list of program variables and examines an asymptotically smaller space than either of the other approaches. However, it is also not guaranteed to find global maxima for either replacement count or speedup.
4) **Hierarchical-compositional** - This strategy uses structural hierarchy information (i.e., code modules and functions) to do a breadth-first search for individual components that can be entirely replaced with single-precision variables in isolation. This search is similar to the original CRAFT hierarchical search. Then, after replacements are found, they are combined using the compositional search strategy to find larger replacements, with the exception that only k-cardinality configurations are considered for merging.

Finally, CRAFT was also improved by adding support for grouping variables by type-dependency labels emitted by TypeForge and adding the ability to run configurations using a job scheduler for better parallelism across a cluster.

B. TypeForge

TypeForge is a tool to facilitate various type refactoring operations. It provides a set of primitive transformations affecting the base-type of various elements of code. These

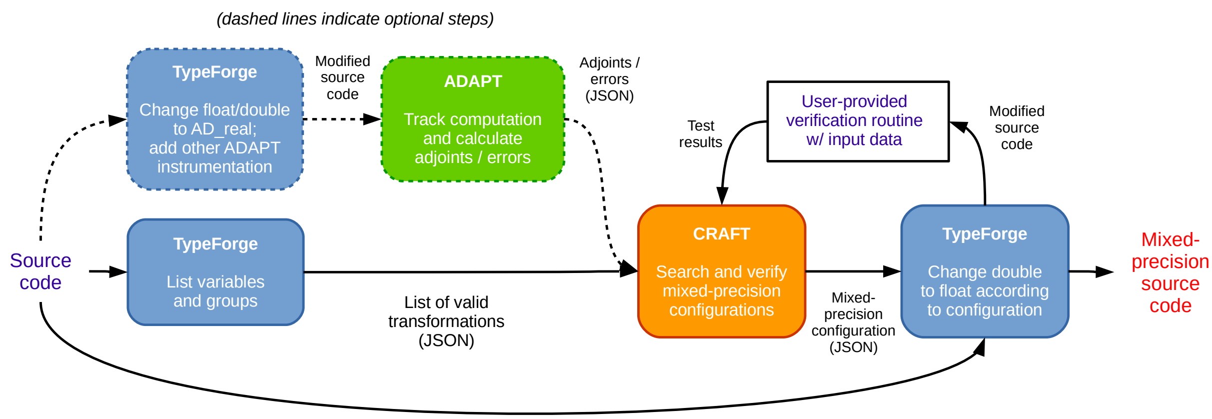

Fig. 1. Integrated system tool chain overview

Data: R: set of all possible individual replacements
Result: P: set of passing configurations
$Q = \{\{r\} \mid r \in R\}$
$P = \emptyset$
while $Q \neq \emptyset$ **do**
  $c = choose(Q)$
  $Q = Q - \{c\}$
  if *test(c)* **then**
    $Q = Q \cup \{c \cup p \mid p \in P,\ c \cap p = \emptyset,\ (|p| = |c|) \vee (|p| = 1)\}$
    $P = P \cup \{c\}$
  end
end

Algorithm 1: Compositional search

elements include the types of variables, function parameters, and function return types. In addition, a function's call site can be transformed to handle template functions where the return type depends on a template argument.

TypeForge recognizes native compound types such as "`const double`", "`double *`", and "`double[3]`" as well as instantiated containers from the C++ Standard Template Library (STL) such as "`std::vector<double const &> const &`". When replacing "`double`" by "`float`", the four examples above respectively become: "`const float`", "`float *`", "`float[3]`", and "`std::vector<float const &> const &`". Because of the flexible nature of the construction of types in ROSE, we can handle any composition of native types such as "`float * const * x[10]`".

In FloatSmith, TypeForge is used for three tasks: (a) replace/change the base type of all floating-point variables, function parameters, or function return-type with ADAPT's differentiable type "`AD_real`", (b) list all variables, function parameters, or function return-type whose type is based on "`double`" but could be changed to "`float`", and (c) replace/change base type of selected variables, function parameters, or function return-type by "`float`". TypeForge performs these tasks using its three main capabilities: (1) enumerating possible transformations, (2) clustering dependent transformations, and (3) generating source code with transformed base types.

```
1 double a0[8];
2 double a1[8];
3 double * ptr = a0;
4 *ptr = 2.;
5 ptr = a1 + 4;
```

Listing 1. Example of dependent type changes. Line 3: typeof(ptr) must be compatible with typeof(a0). Line 5: typeof(ptr) must be compatible with typeof(a1).

ADAPT's transformation (a) is performed only using capabilities (1) and (3) because all transformations are applied together. For the search phase with CRAFT we also use the clustering capability (2), because the task (b) that lists all variables, function parameters, and function return-types that can be changed yields a large number of potential transformations; however, some transformations can yield incorrect code if applied by themselves. For example, Listing 1 demonstrates a situation where changing the base type of any of three variables (`a0`, `a1`, or `ptr`) requires that all three variables must be changed. Our clustering method (2) solves this issue by grouping together transformations depending on each other. This leads to a reduction of the size of the search space, sometimes by a factor of two (see the LULESH results in Table III).

```
struct Domain {
  std::vector<double> v;

  Domain() : v(N, 0.) {}

  double & get(int i) {
    return v[i];
  }

  void extract(int off, int n, double * p) {
    for (int i = off; i < off + n; ++i) {
      *(p + i) = v[i];
    }
  }
```

```
15 };
16
17 int main() {
18   Domain D(2*N);
19   double R;
20
21   for (int i = 0; i < 2*N; ++i) {
22     double & v = D.get(i);
23     R += v;
24   }
25
26   double * arr = alloc<double>(N);
27   D.extract(3, N, arr);
28
29   return 0;
30 }
```

Listing 2. Demonstrating code features found in LULESH. Line 2, 7, & 12: Vector from the STL is used to store floating-point values. Line 6 & 22: Accessor returns reference to floating-point values. Line 10 & 27: Pointer to floating-point values passed to a function. Line 26: Call to templated function where the return type depends on the template argument.

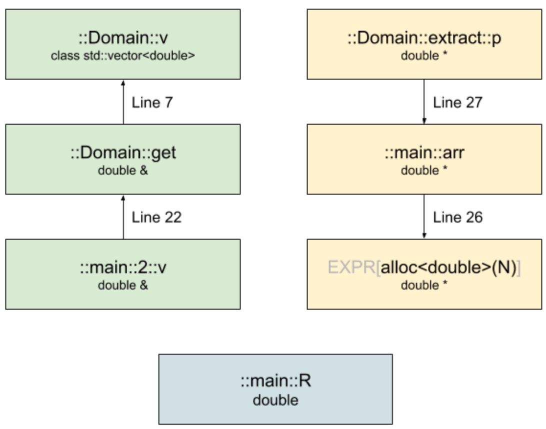

Fig. 2. In this graph, the nodes corresponds to objects that can be transformed by TypeForge (Table I) while the edges correspond to the dependencies (Table II). Each node shows the unique name of the object and its original type. The edges are annotated with the line that introduces the corresponding dependency. Nodes are colored depending on their respective clusters.

TypeForge works directly on ROSE's Abstract Syntax Tree (AST). It proceeds in the following steps:

1) TypeForge identifies objects in the AST for which it can apply type transformations. In Table I, we show the list of objects for Listing 2. While we can change the type of variables, parameters, and function return-types, we also had to permit call-site transformations. This was driven by the templated functions where the return-type depends on the template parameters (for instance, LULESH allocates floating-point arrays this way).
2) Objects are filtered based on their types. Only objects whose base type is `float` or `double` can be transformed (such as: `float const *`, `double &`, or `std::vector<double const *>`). In Table I, the type of selected nodes is shown in bold.
3) All expressions in the application are analyzed looking for type dependencies. In Table II, we describe the dependencies introduced by the expressions in Listing 2.
4) The potential transformations and the dependencies between them form a directed graph. We use a standard clustering algorithm to group together transformations that depend on each other. Figure 2 shows the graph generated for Listing 2 and the clusters are reported in the last column of Table I.

C. ADAPT

ADAPT [8] uses algorithmic differentiation to estimate the error that would be introduced by changing each variable's type. Algorithmic differentiation (AD) uses the chain rule of differentiation to evaluate numerically the derivative of a computer program. ADAPT uses derivatives obtained from AD in conjunction with the first-order Taylor series approximation to construct a model that would estimate the error introduced as a result of the change in precision of variables. ADAPT uses the error model to perform a greedy allocation and recommends variables that should be replaced to maximize the amount of conversions under a provided total error threshold. The AD analysis is done by a header-only wrapper around the CoDiPack [12] library, which uses C++ expression templates to record computation and calculate derivatives for a given program.

Originally, the ADAPT instrumentation was inserted manually: the developer had to replace all floating-point variables with a differentiable type (`AD_real`) and insert some other calls to provide information about execution to the ADAPT library. With the integration of TypeForge, much of this process is now automated, except for marking the beginning and end of the computation of interest as well as indicating which variables should be considered outputs along with their allowable error threshold. Listing 3 shows an example of what these annotations look like.

```
1  int main()
2  {
3      Domain D;
4      double R;
5      initialize_data(&D);
6  #   pragma adapt begin
7      R = perform_computation(&D);
8  #   pragma adapt output R 1e-8
9  #   pragma adapt end
10     return 0;
11 }
```

Listing 3. Sample ADAPT instrumentation (lines 6, 8, and 9).

Additionally, the ADAPT analysis originally could not guarantee a speedup because it only considered whether a replacement was valid according to the error criteria. However, the ADAPT output is now used during the CRAFT search to narrow the search space (usually by restricting the search to ADAPT-recommended variables, or by using the ADAPT error to sort the variables before searching), and the mixed-precision configurations are actually tested in order to determine automatically whether any of them yield a speedup.

The ADAPT tool required relatively few changes for the tool chain project. The only significant new features added were support for the new JSON output format (see section

Object	Line	Kind	Type	Cluster
::Domain::v	2	field	**class ::std::vector<double>**	#1
::Domain::Domain()	4	return type	N/A	
::Domain::get(int)	6	return type	**double &**	#1
::Domain::get(int)::i	6	local var.	int	
::Domain::extract(int,int,double*)	10	return type	void	
::Domain::extract(int,int,double*)::off	10	parameter	int	
::Domain::extract(int,int,double*)::n	10	parameter	int	
::Domain::extract(int,int,double*)::p	10	parameter	**double ***	#2
::Domain::extract(int,int,double*)::0::i	11	local var.	int	
::main()	17	return type	void	
::main()::D	18	local var.	class ::Domain	
::main()::R	19	local var.	**double**	#3
::main()::2::i	21	local var.	int	
::main()::2::v	22	local var.	**double &**	#1
::main()::arr	26	local var.	**double ***	#2
EXPR[alloc<double>(N)]	26	call expression	**double ***	#2

Table I. This table shows all objects detected by TypeForge when analyzing the code from Listing 2. There are three categories of objects: variables (local, global, field, or function parameter), return-type (method or function), and call-expression. Objects with types based on double are candidates for transformation. Selected transformations can require each other leading to independent clusters of transformations.

Line	Code	Effect
7	return v[i]	base type of the return type of ::Domain::get must be same as the base type of ::Domain::v
12	*(p+i) = v[i]	no-effect: value assignment
22	v = D.get(i)	the base type of ::main::2::v must be same as base type of the return type of ::Domain::get
23	R += v	no-effect: value accumulation
26	arr = alloc<double>(N)	type of ::main::arr depends on the call-expression, meaning that it depends on the template argument
27	D.extract(3, N, arr)	base type of ::Domain::extract::p must be the same as the base type of ::main::arr

Table II. This table details the effects of the expressions in Listing 2. In many cases, expressions are evaluating scalar values which are assigned to scalar variables, written at an address, or stored through references. In these cases, the expression does not imply dependencies between the type of the objects (because floating point scalar are cast-compatible). Dependencies arise in the case of pointer arithmetic and when retrieving references to variables.

below for more discussion of this) and support for multiple dependent (output) variables.

D. FloatSmith Coordination

Finally, we developed a new software tool chain called FloatSmith that provides an interactive interface for running the various pieces of the entire system without requiring the user to be an expert in using any of them. This tool asks the user several questions and walks them through creating the various scripts necessary for the rest of the system. It then runs the various pieces of the system, describing them as they run. The actual run scripts are saved so that the user can re-run various stages over again if they wish (or if there is a problem that they need to fix).

For experienced users (or for repeatable experiments) we also provide a batch mode that can be invoked with the `-B` option. This accepts all default options unless overridden using command-line parameters. In this mode, the only required command-line parameter is the `--run` parameter, which specifies how to execute the program. It will assume a standard `make` command can build the project and that the user wants the output to remain identical to the original. For instance, the AXPY example can be run from its folder in the FloatSmith repository using the following simple command:

```
floatsmith -B --run "./axpy"
```

This level of automation is unique to our approach and provides a remarkably low barrier-to-entry, especially with the container image that we also provide to avoid having to install all of the prerequisite tools manually. Of course, the user will likely wish to tweak the search based on the results (e.g., to be more or less strict in the verification), and we provide several ways to do so (changing the search strategy, verifying via regular expression or custom script, etc.).

To enable all three tools to inter-operate cleanly, we designed a new JSON-based data interchange format. This format encodes information about tunable variables, clusters of variables, differentiation results, and mixed-precision configurations. All three tools emit some form of data in this format, and all but ADAPT also read data in this format (ADAPT input information is embedded in source code annotations).

III. Results

We have successfully applied the whole system to several examples and benchmarks. In this section we present our preliminary results. Extending these experiments to larger applications and doing a comprehensive comparison with similar tools is future work. All of these results were run on a cluster with Intel Xeon E5-2630 CPUs and 32GB of RAM, and all performance experiments were repeated ten times with the minimum runtime recorded. All benchmarks except for LULESH (which was multi-threaded with OpenMP) were single-threaded, but the cluster nodes were used to run multiple configurations simultaneously to reduce the overall search time (because the search is naturally parallel).

Table III shows results comparing the various search strategies described above. Clearly, the combinational approach does not scale, and the compositional approach has a similar

	AXPY	SUM2PI	ARCLEN	DFT	LULESH
Candidate variables/clusters	3	8	10	11	710 / 376
Approximate time (in seconds) to run each configuration	4	8	4	4	74
Configurations tested					
Combinational	7	255	1023	2047	>1e110
Compositional	4	128	14	513	571
Delta debugging	6	26	38	42	1772
Hierarch-comp	6	14	15	68	782
Highest number of replacements					
Combinational	2	7	3	8	-
Compositional	2	7	3	8	11
Delta debugging	2	7	1	5	24
Hierarch-comp	2	7	2	6	7
Best speedup					
Combinational	80%	-	-	2%	-
Compositional	80%	-	-	2%	2%
Delta debugging	80%	-	-	-	2%
Hierarch-comp	80%	-	-	-	2%

Table III. Search strategy comparison results (dashed lines generally indicate no speedup; also, LULESH was not run in combinational mode due to an infeasible number of configurations)

	FFT	EP	CG	MG
Candidate variables	25	64	86	115
Approximate time (in seconds) to run each configuration	4	19	10	18
Configurations tested				
Combinational	>1e7	>1e19	>1e25	>1e34
Delta debugging	96	366	382	456
Highest number of replacements				
Delta debugging	24	4	22	4
Best speedup				
Delta debugging	-	-	4%	1%

Table IV. Mid-sized program search results (dashed lines indicate no speedup)

problem, especially when many individual variables can be replaced. Delta debugging is often more efficient and converges with fewer tested configurations, but does not always find all possible replacements. Hierarchical-compositional often converges relatively quickly and sometimes finds more passing configurations than delta debugging.

The first example is AXPY, a synthetic benchmark created to demonstrate the potential benefit of mixed precision. It does some simple vectorizable operations on two very large arrays, one of which can be stored in single precision without compromising the final results. As expected, all of the searches we ran found this replacement (along with one other scalar replacement), and the performance improvement was around 80% on our test machine. The original maximum resident set size was 1,563,564 kbytes (1.5 GB) while the mixed-precision maximum resident set size was 1,172,940 (1.1 GB), and page faults dropped from 584,483 to 1,505. This search was entirely automated; all FloatSmith had to be told was how to run the program (see the command in the previous section).

The SUM2PI and ARCLEN examples are from CRAFT and Precimonious, respectively, and the DFT example is a simple discrete Fourier transform implementation used in other prior mixed-precision work [13]. All three perform iterative calculations where certain variables can be stored in single precision. Again, all of our FloatSmith searches found replacements in all of these examples; however, these examples do not demonstrate a significant speedup because the conversion does not enable any new optimizations except for a very minor (2%) speedup in DFT.

Table IV shows results from a few other mid-sized benchmarks. These were only run with the delta-debugging strategy due to time constraints. The FFT benchmark is from the GNU Scientific Library [4], and the EP, CG, and MG benchmarks are from the NAS Parallel Benchmark suite [14]. These benchmarks represent a significant increase in complexity, as the number of variables rise into the dozens and the source code is split among multiple files (the total number of lines ranges from ≈250 for FFT to over 1,500 for MG). Again, FloatSmith found valid mixed-precision replacements, even if none of them result in a significant speedup on the CPU.

Finally, the LULESH benchmark [15] is a well-known Department of Energy proxy application (≈6,600 lines) that has been used to demonstrate mixed-precision results in previous work [8], [10], with speedups of 20% or more. We tested the OpenMP version (enabling thread-based parallelism) with `-O3` optimization enabled and a problem size of $50 \times 50 \times 50$. As in another recent work [10], we check the iteration count and final origin energy, which must match the original exactly. We also check that the "TotalAbsDiff" metric is on the same order (i.e., one digit of accuracy). Because of the large number of variables, we ran our searches based on the clusters reported by TypeForge rather than individual variables.

Unfortunately, we are not currently able to replicate the 20+% speedup found in previous work [8], [10], for the

[4] `urlhttps://www.gnu.org/software/gsl/`

	w/o ADAPT	w/ ADAPT
SUM2PI		
Combinational	255	15
Compositional	128	8
Delta debugging	26	26
Hierarch-Comp	14	6
ARCLEN		
Combinational	1023	3
Compositional	14	3
Delta debugging	38	38
Hierarch-Comp	15	1

Table V. Impact of ADAPT recommendations on SUM2PI search (total configs tested)

following reasons. First, the result in Menon et al. [8] was obtained using a source transformation that involved creating multiple versions of a function and a new temporary data structure. We are currently unable to automate this level of sophistication in transformation. Second, the result in Laguna et al. [10] was based on the GPU version and also worked at the LLVM level, transforming LLVM IR and thus finding opportunities for mixed precision that do not translate well back to source-level transformations.

However, we do find valid configurations, and unlike the previous approaches our technique provides a source-level transformation automatically [5]. We anticipate that in the future more sophisticated analysis will be able to close these gaps and replicate or improve on the speedups found by prior work.

Table V shows results demonstrating the impact of ADAPT analysis on the search phase. If ADAPT info is present, the combinational and compositional search strategies will use it to narrow the field of valid candidates for replacement and consider only those that ADAPT recommends replacing. As the results show, this can significantly reduce the number of configurations that must be tested. If ADAPT info is present, the delta debugging search strategy will sort the variables by ADAPT-reported error, potentially improving the search convergence by grouping low-error variables together. However, we did not observe this effect in our initial experiments.

IV. Future Work

A. Generalization of Results

There are limitations inherent to using a testing-based approach to find mixed-precision configurations. Our techniques use a developer-provided testing routine to invoke potential configurations with a representative data set and to verify that the output has an acceptable level of accuracy. Strictly speaking, any results could be only applicable to the given input data set. However, in practice the results usually generalize to some extent, and this sort of dynamic analysis is a pragmatic approach taken in many domains, including performance optimization and software testing. The nature of our analysis allows us to analyze whole programs on a scale approaching HPC applications, which is generally impossible for more conservative and rigorous approaches. In the future, however, we hope to mitigate this limitation using various techniques such as input fuzzing (a testing technique that provides random or invalid data) and automatic detection of pathological inputs.

B. Performance Prediction

Currently, the search strategy has no reliable way to determine which configuration(s) will have the best performance. This is determined by trial-and-error. A performance model that could accurately predict the performance of a configuration without actually building and running it would make the search converge much quicker. One possible performance model involves detecting the number of floating-point casts introduced by a mixed-precision configuration (similar to the approach used by GPUMixer [10]). TypeForge already reports static cast information, and we plan to use it to estimate the performance impact.

C. Extension to GPUs

Speedups due to precision reduction are more significant on accelerated platforms. With ADAPT [8], for example, the authors found a 20% speedup on LULESH only with the GPU version. This is because single-precision arithmetic is often not significantly faster on CPUs; speedup must be achieved by vectorization and reduced memory pressure. Theoretically, our system can support tuning GPU code directly because ROSE has been extended to include CUDA support. Unfortunately, due to time constraints this has been relegated to future work.

D. Integration with Other Tools

We hope to be able to add more components in the future. This could include components to improve accuracy [16], bound error [17], perform cancellation or dynamic range detection [6], prototype alternative representations [18], or estimate/improve fault tolerance. Some of these (like the differentiation analysis of ADAPT and the type dependency analysis of TypeForge) could help narrow the search space.

E. Comparison to Related Work

There is prior work in building mixed-precision configurations using software tools, including CRAFT [6], [3], [7] and ADAPT [8], which we extended in this work to build an end-to-end source-level tuning system. In this sense, our work is similar to a recent effort [19] to combine other floating-point tools, although the purpose of those tools was to optimize accuracy in a statically-verifiable way rather than to inform mixed-precision implementations.

Other related work in automated mixed-precision analysis includes Precimonious [4], [20] and HiFPTuner [5], FPTuner [1], Daisy [21], [2], [17], GPUMixer [10], and AMPT-GA [11] among others (e.g., [13], [22], [23], [24], [25], [26]). All of these prior approaches build mixed-precision versions of a program in one way or another, but none of them provide an end-to-end source-level tuning framework that is as automated as our approach. A detailed comparison with these efforts is future work.

[5]The output currently has formatting differences due to the way ROSE re-emits modified code. ROSE does provide a mechanism that will allow us re-emit the code with no or minimal formatting modifications (to diff against the original code), and this feature will be investigated in future work.

V. Conclusion

We combined three program analysis components (CRAFT, TypeForge, and ADAPT) into an end-to-end precision tuning system called FloatSmith. We extended all three components to enable the integration, implemented new software to coordinate the components, and tested the system on several examples and benchmarks. We demonstrated that such analysis is feasible and applicable to small-scale HPC workloads. We also identified several ideas for extension projects, and anticipate that the system will serve as a foundation for many avenues of future work.

Acknowledgements

We acknowledge Scott Lloyd, who provided invaluable guidance in the early stages of this project. We also acknowledge Logan Moody, who provided the initial design and implementation of the data interchange format, as well as Nathan Pinnow, who helped to extend TypeForge.

We also gratefully thank Jeff Hittinger for funding this project. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, via LDRD project 17-SI-004. IM release number LLNL-CONF-787885.

References

[1] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamari, "Rigorous Floating-Point Mixed-Precision Tuning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, (New York, NY, USA), pp. 300—-315, ACM, 2017.

[2] E. Darulova, E. Horn, and S. Sharma, "Sound Mixed-Precision Optimization with Rewriting," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS'18)*, (Porto, Portugal), pp. 208–219, IEEE Press, 2018.

[3] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically Adapting Programs for Mixed-Precision Floating-Point Computation," in *Proceedings of the 27th International ACM Conference on Supercomputing (ICS '13)*, (New York, New York, USA), p. 369, ACM Press, jun 2013.

[4] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning Assistant for Floating-Point Precision," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on (SC'13)*, (New York, New York, USA), pp. 1–12, ACM Press, nov 2013.

[5] H. Guo and C. Rubio-González, "Exploiting Community Structure for Floating-Point Precision Tuning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*, no. 333, (Amsterdam, Netherlands), pp. 333–343, ACM, 2018.

[6] M. O. Lam, J. K. Hollingsworth, and G. Stewart, "Dynamic Floating-Point Cancellation Detection," *Parallel Computing*, vol. 39, pp. 146–155, mar 2013.

[7] M. O. Lam and J. K. Hollingsworth, "Fine-grained floating-point precision analysis," *The International Journal of High Performance Computing Applications*, vol. 32, no. 2, pp. 231–245, 2018.

[8] H. Menon, M. O. Lam, D. Osei-kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "ADAPT : Algorithmic Differentiation Applied to Floating-Point Precision Tuning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*, (Dallas, Texas), pp. 48:1—-48:13, IEEE Press, 2018.

[9] M. Sagebaum, T. Albring, and N. R. Gauger, "High-performance derivative computations using codipack," *arXiv preprint arXiv:1709.07229*, 2017.

[10] I. Laguna, P. C. Wood, R. Singh, and S. Bagchi, "GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications," in *High Performance Computing* (M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, eds.), (Cham), pp. 227–246, Springer International Publishing, 2019.

[11] P. V. Kotipalli, R. Singh, P. Wood, I. Laguna, and S. Bagchi, "AMPT-GA: Automatic Mixed Precision Floating Point Tuning for GPU Applications," in *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, (New York, NY, USA), pp. 160–170, ACM, 2019.

[12] M. Sagebaum, T. Albring, and N. R. Gauger, "High-Performance Derivative Computations using CoDiPack," tech. rep., 2012.

[13] A. A. Gaffar, O. Mencer, W. Luk, P. Y. K. Cheung, and N. Shirazi, "Floating-point bitwidth analysis via automatic differentiation," in *Proceedings - 2002 IEEE International Conference on FieId-Programmable Technology, FPT 2002*, 2002.

[14] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *International Journal of Supercomputer Applications*, vol. 5, pp. 63–73, sep 1991.

[15] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," tech. rep., Livermore, CA, 2013.

[16] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*, pp. 1–11, 2015.

[17] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, "Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper)," in *24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 10805 LNCS, (Thessaloniki, Greece), pp. 270–287, 2018.

[18] M. O. Lam and B. L. Rountree, "Floating-Point Shadow Value Analysis," in *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*, ESPT '16, (Piscataway, NJ, USA), pp. 18–25, IEEE Press, 2016.

[19] H. Becker, P. Panchekha, E. Darulova, and Z. Tatlock, "Combining tools for optimization and analysis of floating-point computations," in *Formal Methods* (K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, eds.), (Cham), pp. 355–363, Springer International Publishing, 2018.

[20] C. Rubio-González, D. Hough, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, and D. H. Bailey, "Floating-point precision tuning using blame analysis," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, (New York, New York, USA), pp. 1074–1085, ACM Press, 2016.

[21] E. Darulova and V. Kuncak, "Sound compilation of reals," *ACM SIGPLAN Notices*, vol. 49, pp. 235–235–248–248, jan 2014.

[22] R. Nathan, B. Anthonio, S. L. Lu, H. Naeimi, D. J. Sorin, and X. Sun, "Recycled Error Bits: Energy-Efficient Architectural Support for Floating Point Accuracy," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2014.

[23] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, "PROMISE: floating-point precision tuning with stochastic arithmetic," in *17th international symposium on Scientific Computing, Computer Arithmetic and Verified Numerics (SCAN 2016)*, (UPPSALA, Sweden), pp. 98–99, Sept. 2016.

[24] N. Ho, E. Manogaran, W. Wong, and A. Anoosheh, "Efficient floating point precision tuning for approximate computing," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 63–68, Jan 2017.

[25] R. Medhat, M. O. Lam, B. L. Rountree, B. Bonakdarpour, and S. Fischmeister, "Managing the Performance/Error Tradeoff of Floating-point Intensive Applications," in *Proceedings of the International Conference on Embedded Software (EMSOFT'17)*, ACM, 2017.

[26] Y. Chatelain, E. Petit, P. de Oliveira Castro, G. Lartigue, and D. Defour, "Automatic exploration of reduced floating-point representations in iterative methods," in *Euro-Par 2019: Parallel Processing* (R. Yahyapour, ed.), (Cham), pp. 481–494, Springer International Publishing, 2019.

VI. Artifact Description

The code for the tools described in this paper is open source and available in the following public GitHub repositories:

- FloatSmith: github.com/crafthpc/floatsmith (GPL3)
- CRAFT: github.com/crafthpc/craft (LGPL3)
- ADAPT: github.com/LLNL/adapt-fp (GPL3)
- TypeForge: github.com/rose-compiler/rose (tag 0.9.11.95) (revised BSD)

The FloatSmith repository contains documentation on how to install and use the entire tool chain (an automated installer script for a known-working dependency configuration is provided). Local installation requires a Linux-based operating system on an x86 architecture with a C/C++ compiler that supports C++11. However, results for selected examples can be reproduced in a hardware-agnostic container from Docker Hub without full installation (note that it may take several hours to rebuild these results depending on your hardware):

```
docker pull lam2mo/floatsmith
docker run -it lam2mo/floatsmith
./run_experiments.sh
```

The FloatSmith repository does not contain some examples and benchmarks, such as the NAS benchmarks (available at https://www.nas.nasa.gov/publications/npb.html) or LULESH (available at https://computing.llnl.gov/projects/co-design/lulesh). The FloatSmith repository contains information about how to rebuild and use the Docker container to analyze arbitrary programs on a local file system.

The results in the paper were generated on a 16-node Intel Xeon cluster running RHEL7 with GCC 4.9.3. Benchmark versions used include NPB 3.1 and LULESH 2.0. Benchmarks were unmodified except for adding ADAPT annotations (optional) and combining the LULESH code into a single source file for simpler analysis (also optional). No external data sets are necessary.