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

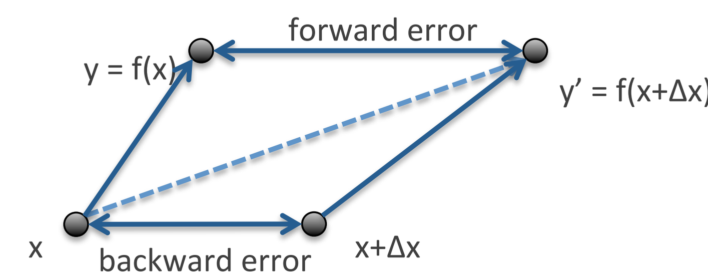
As scientific computation continues to scale, it is crucial to use floating-point arithmetic processors as efficiently as possible. Lower precision allows streaming architectures to perform more operations per second and can reduce memory bandwidth pressure on all architectures. However, using a precision that is too low for a given algorithm and data set will result in inaccurate results.

We present a framework that uses binary modification to build mixed-precision configurations of existing binaries that were originally developed to use only double precision. This allows developers to easily experiment with mixed-precision configurations without modifying their source code, and it permits auto-tuning of floating-point precision. We include a search algorithm to automatically identify which code regions can use lower precision. Initial results with the Algebraic MultiGrid kernel demonstrate a nearly 2X speedup.

Motivation

- Finite floating-point precision causes round-off error
  - Occurs when a real number cannot be exactly represented
  - Compromises certain "ill-conditioned" calculations
  - Actual problems are very hard to detect and diagnose
- Increasingly important as HPC scales
  - Double-precision data movement is a bottleneck
  - Streaming processors are faster in single-precision (~2x)
  - Need to balance speed (singles) and accuracy (doubles)

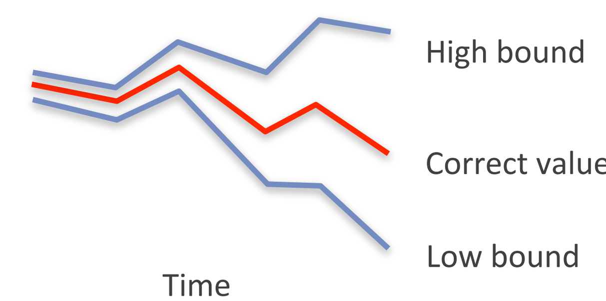
Related Work



Traditional backwards and forwards error analysis is very useful, but requires extensive numerical analysis expertise.

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

Manual mixed-precision algorithm; asterisks and red text indicate double-precision steps. Not immediately generalizable; this is what we hope to automate.



Interval arithmetic represents numbers with upper & lower bounds; it is too conservative and expensive.

Methods

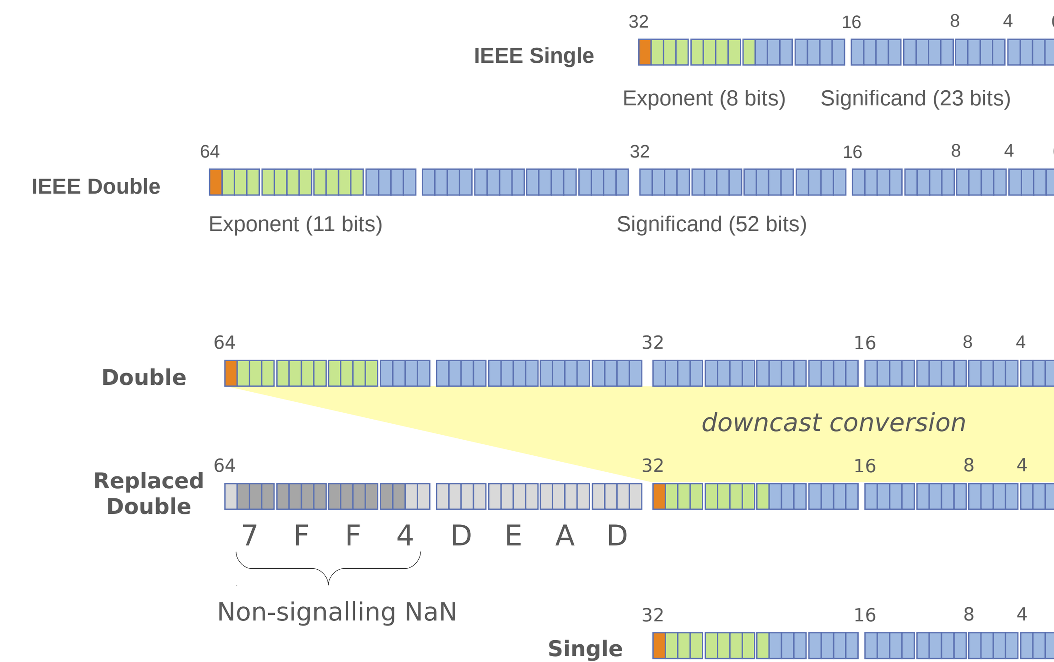
In-place Single-precision Replacement

General approach:

- Selectively replace some double-precision instructions with their single-precision equivalents
- Replace double-precision operands with their single-precision equivalents in memory

Narrowing conversions:

- 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)



Binary Modification

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.

```
  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>
```

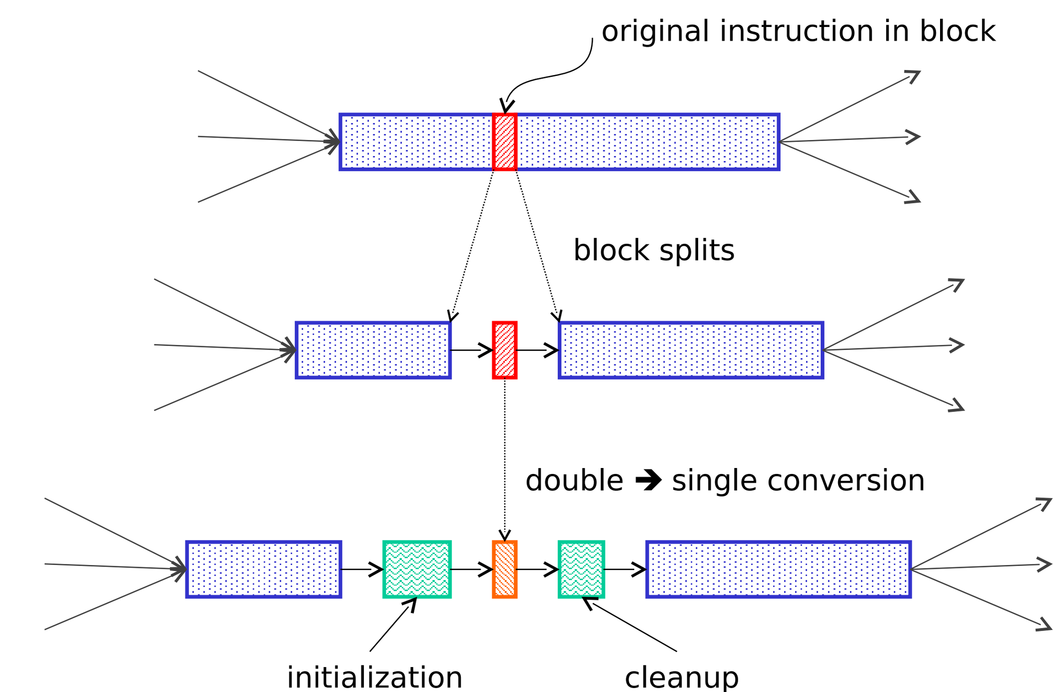
Configuration Representations

The core configuration model is a series of mappings p → {*single*, *double*, *ignore*} for all p ∈ P_d, where P_d is the set of all double-precision instructions in program P.

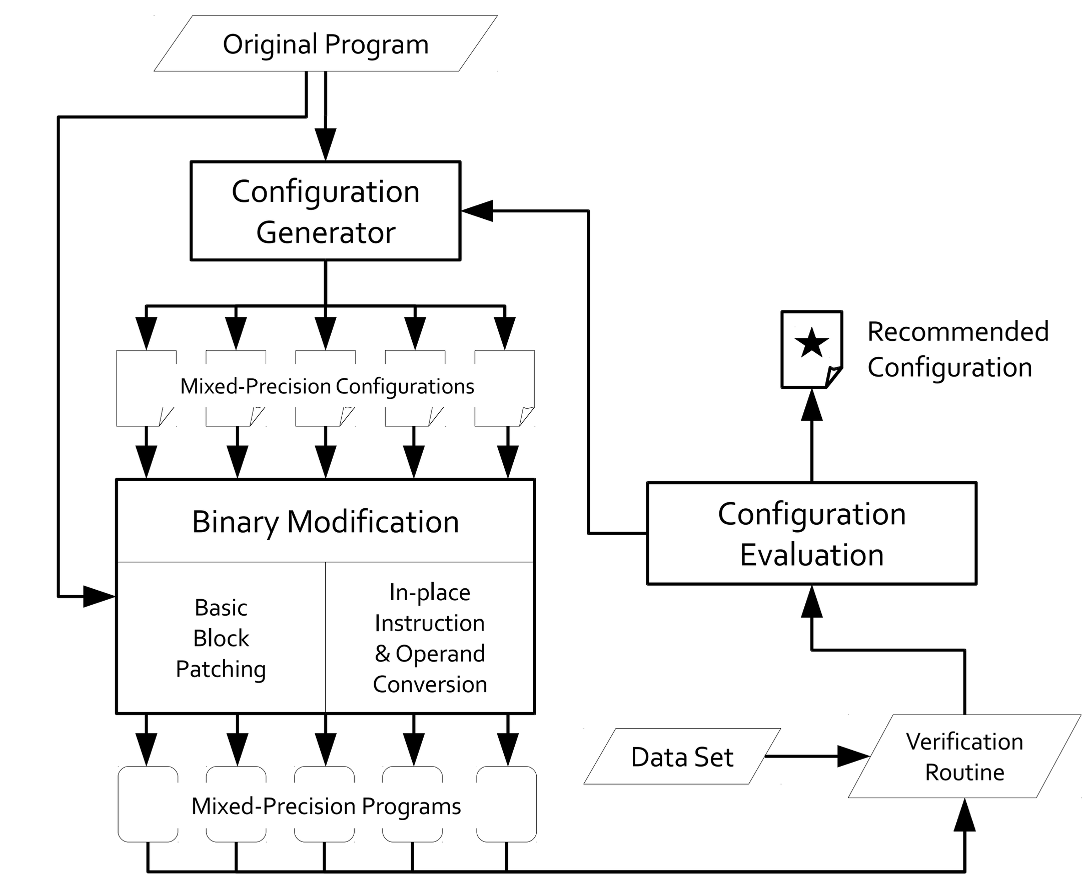
- If the mapping for p_i is *single*:
  - Replace opcode
  - Cast inputs to single precision
  - Store replaced double precision (single precision with flag)
- If the mapping for p_i is *double*:
  - Do NOT replace opcode
  - Cast inputs to double precision
  - Store regular double precision
- If the mapping for p_i is *ignore*:
  - Do not modify the instruction (useful for flagging unusual constructs; e.g., random number generators)

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 re-arrange the edges between blocks.



System Overview



We introduce a system for auto-tuning the precision level of a target application.

Given a representative data set and a verification routine, this system builds multiple mixed-precision configurations of the application and evaluates them. The system determines which configurations produce "valid" results according to a user-defined correctness test.

As output, the system produces the valid configuration that results in the highest number of single-precision instruction executions for the representative run.

Future improvements:

- New search algorithms
- Inclusion of performance data
- Inclusion of conversion costs

Overhead and Sensitivity

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

Epsilon	Static	Dynamic	Verification
1.0e-02	61.6%	36.9%	SUCCESS
1.0e-04	61.6%	36.9%	FAILED
1.0e-06	61.6%	36.9%	FAILED
1.0e-08	59.2%	13.2%	SUCCESS
1.0e-12	58.9%	11.3%	SUCCESS

Above: NAS EP replacement percentages and verification status, varying the epsilon value used for verification comparison

Left: Complete replacement overhead results from a selection of NAS benchmarks

Results

End-to-end test: We tested on the Algebraic MultiGrid microkernel, which performs the critical sections of a multigrid solver. For our experiments, we used 5,000 iterations on eight cores.

Highlights:

- The search algorithm confirmed that the entire application could be replaced with single precision.
- The average analysis overhead was only 1.2X for each configuration test run.
- Results were verified by manually converting the entire program to single precision and re-compiling.
- We observed a user CPU time decrease from 175.48s to 95.25s, a **nearly 2X speedup**!