

Optimization of Floating-point Precision using Binary Modification

Michael Lam




University of Maryland, College Park

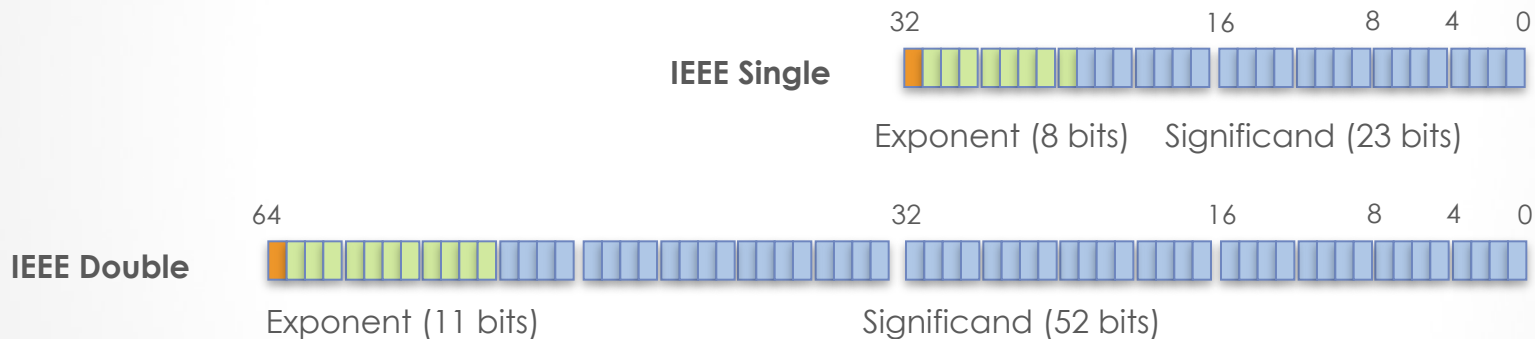
Jeff Hollingsworth, Advisor

Dyn
inst



Background

- Floating-point represents real numbers as $(\pm \text{sgnf} \times 2^{\text{exp}})$
 - Sign bit 
 - Exponent 
 - Significand (“mantissa” or “fraction”) 



- Floating-point numbers have finite precision
 - Single-precision: 24 bits (~7 decimal digits)
 - Double-precision: 53 bits (~16 decimal digits)

Example

$$1/10 \rightarrow 0.1$$

0x3DCCCCD = 00111101 11001100 11001100 11001101



Single-precision

**0x3FB999999999999A = 00111111 10111001 10011001 10011001
10011001 10011001 10011001 10011010**



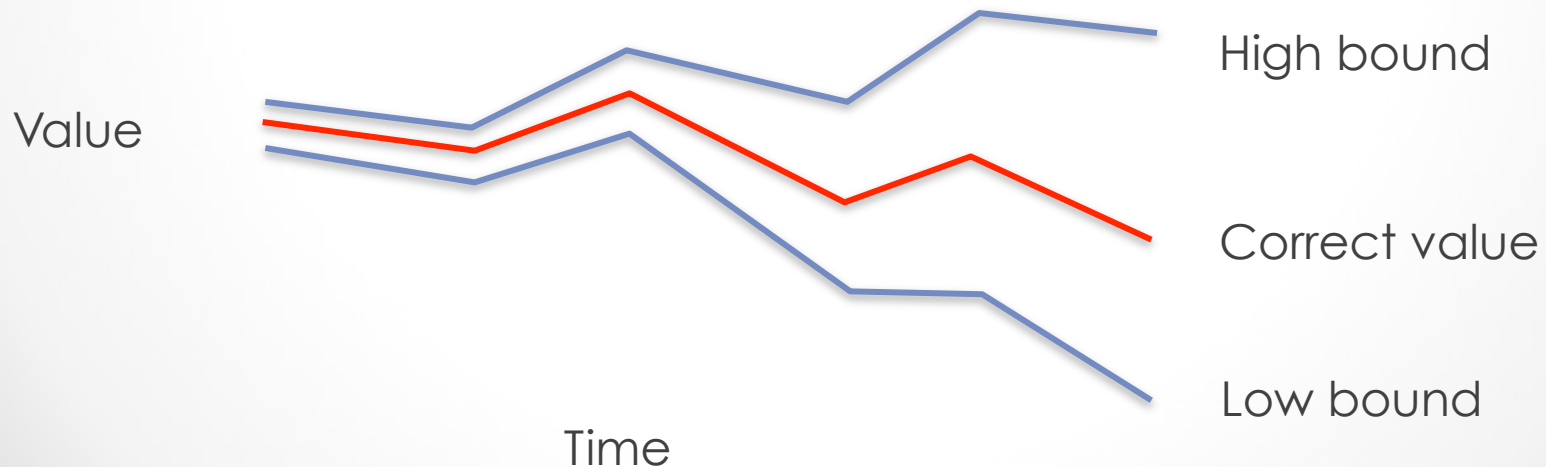
Double-precision

Motivation

- Finite precision causes round-off error
 - Compromises “ill-conditioned” calculations
 - 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)

Previous Work

- Traditional error analysis (Wilkinson 1964)
 - Forwards vs. backwards
 - Requires extensive numerical analysis expertise
- Interval/affine arithmetic (Goubault 2001)
 - Conservative static error bounds are largely unhelpful



Previous Work

- Manual mixed-precision (Dongarra 2008)

- Requires numerical expertise

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

**Mixed-precision
linear solver
algorithm**

**Red text indicates
steps performed in
double-precision (all
other steps are
single-precision)**

- Fallback: ad-hoc experiments
 - Tedious, time-consuming, and error-prone

Our Goal

Develop automated analysis techniques to **inform** developers about floating-point behavior and make **recommendations** regarding the precision level that each part of a computer program must use in order to maintain overall accuracy.

Framework

CRAFT: Configurable Runtime Analysis for Floating-point Tuning

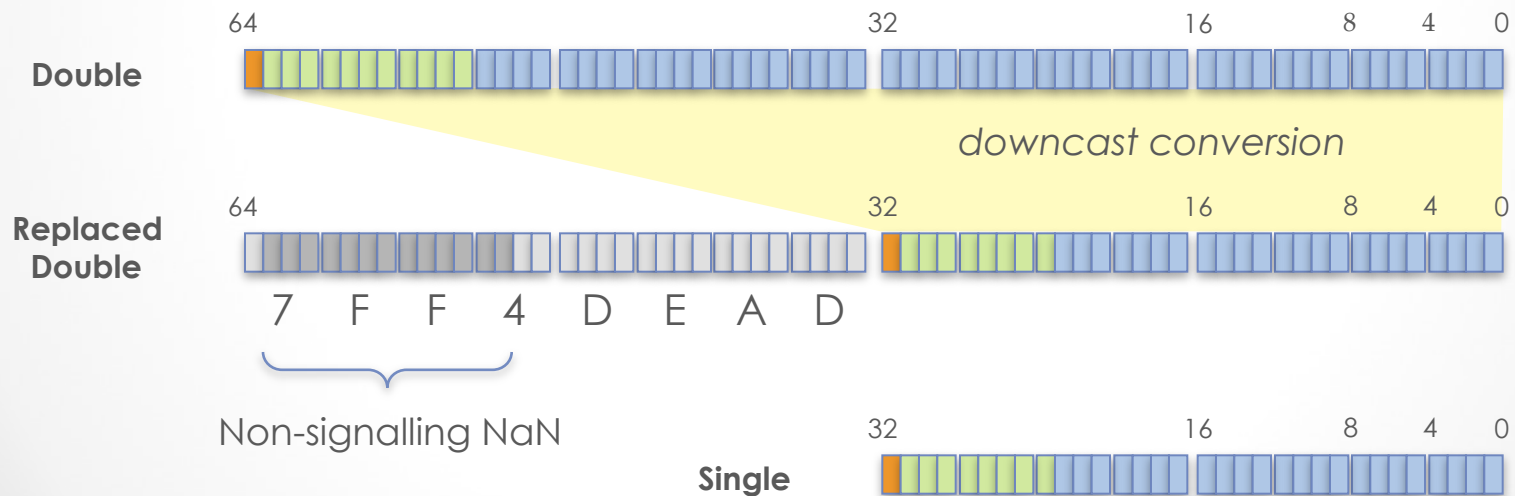
- Static binary instrumentation
 - Controlled by configuration settings
 - Replace floating-point instructions with new code
 - Re-write a modified binary
- Dynamic analysis
 - Run modified program on representative data set
 - Produces results and recommendations

Advantages

- Automated
 - Minimize developer effort
 - Ensure consistency and correctness
- Binary-level
 - Include shared libraries without source code
 - Include compiler optimizations
- Runtime
 - Dataset and communication sensitivity

Implementation

- Current approach: in-place replacement
 - Narrowed focus: doubles \rightarrow singles
 - In-place downcast conversion
 - Flag in the high bits to indicate replacement



Example

$\text{gvec}[i,j] = \text{gvec}[i,j] * \text{lvec}[3] + \text{gvar}$

1 `movsd 0x601e38(%rax, %rbx, 8) → %xmm0`

2 `mulsd -0x78(%rsp) → %xmm0`

3 `addsd -0x4f02(%rip) → %xmm0`

4 `movsd %xmm0 → 0x601e38(%rax, %rbx, 8)`

Example

$\text{gvec}[i,j] = \text{gvec}[i,j] * \text{lvec}[3] + \text{gvar}$

1 `movsd 0x601e38(%rax, %rbx, 8) → %xmm0`

check/replace -0x78(%rsp) and %xmm0

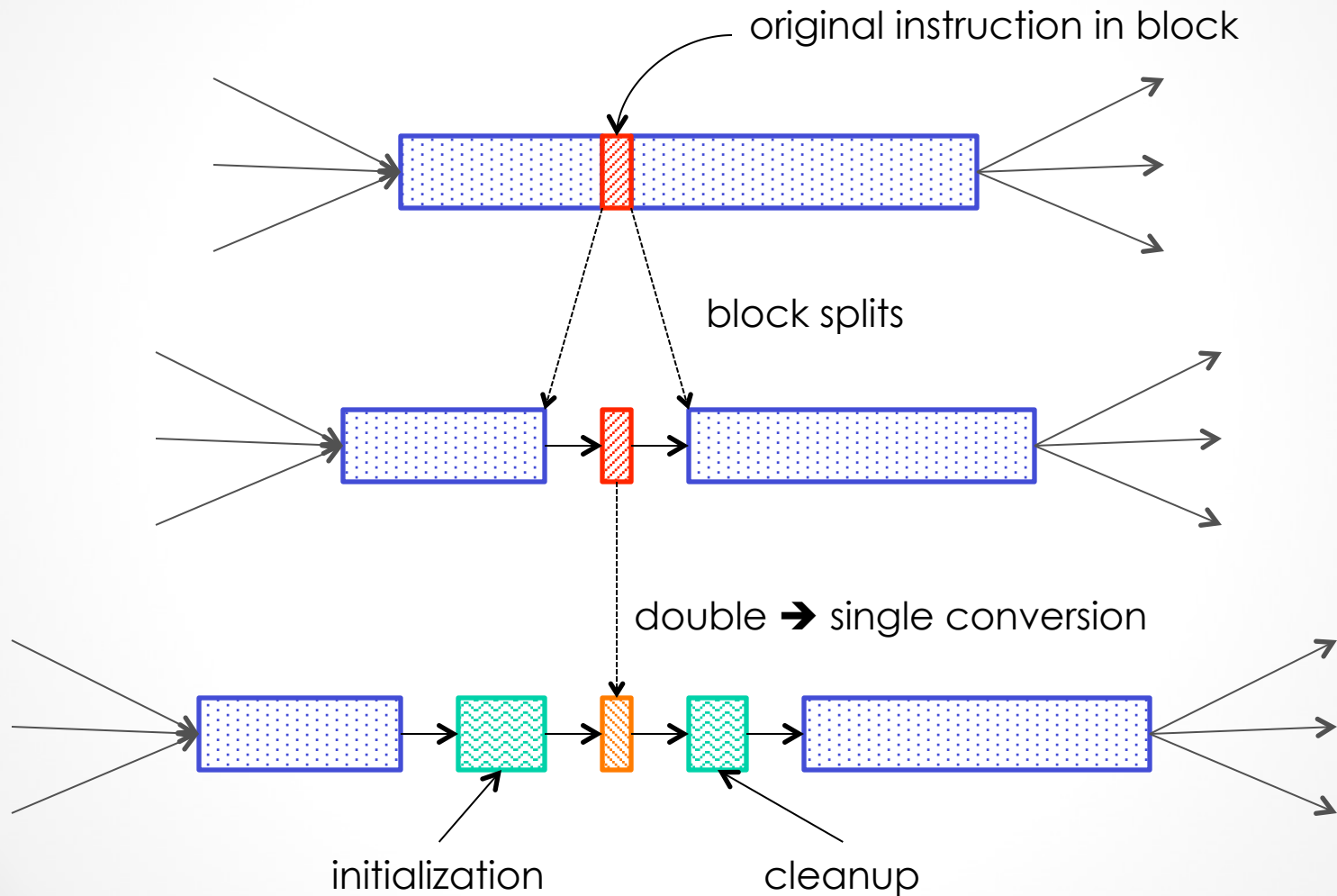
2 `mulss -0x78(%rsp) → %xmm0`

check/replace -0x4f02(%rip) and %xmm0

3 `addss -0x20dd43(%rip) → %xmm0`

4 `movsd %xmm0 → 0x601e38(%rax, %rbx, 8)`

Implementation



00400000

addsd xmm0, xmm1

00606000	nop	006060dd	rol rax, 0x20
00606001	mov qword ptr [rsp-0xb8], rax	006060e1	movlpd qword ptr [rsp-0xe8], xmm1
00606009	mov rax, 0x0	006060ea	mov rcx, 0xffffffff
00606013	lahf	006060f4	and qword ptr [rsp-0xe8], rcx
00606014	seto al	006060fc	or qword ptr [rsp-0xe8], rax
00606017	mov qword ptr [rsp-0xc0], rax	00606104	movlpd xmm1, qword ptr [rsp-0xe8]
0060601f	mov qword ptr [rsp-0xd0], rax	0060610d	mov rax, 0x605000
00606027	mov qword ptr [rsp-0xd8], rbx	00606117	mov rbx, qword ptr [rax]
0060602f	mov qword ptr [rsp-0xe0], rcx	0060611a	inc qword ptr [rbx+0xf8]
00606037	movq rax, xmm0	00606121	mov rcx, qword ptr [rsp-0xe0]
0060603c	mov rbx, 0x7fffffff00000000	00606129	mov rbx, qword ptr [rsp-0xd8]
00606046	and rax, rbx	00606131	mov rax, qword ptr [rsp-0xd0]
00606049	mov rbx, 0x7ff4dead00000000	00606139	mov rax, qword ptr [rsp-0xb8]
00606053	cmp rbx, rax		
00606056	jz 0x7fff1d923f6c	00606141	addss xmm0, xmm1
006060a2	movq rax, xmm1		
006060a7	mov rbx, 0x7fffffff00000000	00606145	mov qword ptr [rsp-0xb8], rax
006060b1	and rax, rbx	0060614d	mov qword ptr [rsp-0xd0], rax
006060b4	mov rbx, 0x7ff4dead00000000	00606155	mov qword ptr [rsp-0xd8], rcx
006060be	cmp rbx, rax	0060615d	mov eax, 0x7ff4dead
006060c1	jz 0x7fff1d923f6c	00606162	mov rcx, 0xffffffff
0060605c	cvtsd2ss xmm0, xmm0	0060616c	and rax, rcx
00606060	mov eax, 0x7ff4dead	0060616f	rol rax, 0x20
00606065	mov rcx, 0xffffffff	00606173	movlpd qword ptr [rsp-0xe0], xmm0
0060606f	and rax, rcx	0060617c	mov rcx, 0xffffffff
00606072	rol rax, 0x20	00606186	and qword ptr [rsp-0xe0], rcx
00606076	movlpd qword ptr [rsp-0xe8], xmm0	0060618e	or qword ptr [rsp-0xe0], rax
0060607f	mov rcx, 0xffffffff	00606196	movlpd xmm0, qword ptr [rsp-0xe0]
00606089	and qword ptr [rsp-0xe8], rcx	0060619f	mov rcx, qword ptr [rsp-0xd8]
00606091	or qword ptr [rsp-0xe8], rax	006061a7	mov rax, qword ptr [rsp-0xd0]
00606099	movlpd xmm0, qword ptr [rsp-0xe8]	006061af	mov rax, qword ptr [rsp-0xc0]
006060c7	cvtsd2ss xmm1, xmm1	006061b7	add al, 0x7f
006060cb	mov eax, 0x7ff4dead	006061ba	sahf
006060d0	mov rcx, 0xffffffff	006061bb	mov rax, qword ptr [rsp-0xb8]
006060da	and rax, rcx	006061c3	nop

Configuration

The screenshot displays the FPAnalysis Config Editor interface. At the top, there are search and control buttons: "Search:", "Expand all", "Collapse all", "Show effective status" (checked), "Toggle selection", and "Save". The main area shows a tree view of functions and blocks. Functions listed include "diffusion", "laplacian", "main", "transfb_nc1", "transfb", "transfb", "transfb", "transfb_cor_e", and "transfb". Each function is expanded to show its constituent Basic Blocks (BBLK) and instructions (INSN). Instructions are color-coded: green for "single" and red for "double". At the bottom, the instruction counts are summarized: NONE (0), IGNORE (0), SINGLE (553), and DOUBLE (32).

Function	Block	Instruction	Count
FUNC: 0x404458 "diffusion" [24 instruction(s)]	BBLK: 0x40452e	INSN: 0x40465b "mulsd xmm0, xmm1"	1
		INSN: 0x404741 "mulsd xmm0, xmm1"	1
		INSN: 0x40479e "mulsd xmm0, xmm1"	1
		INSN: 0x4047b7 "addsd xmm0, xmm1"	1
	BBLK: 0x404830	INSN: 0x40486e "mulsd xmm0, xmm1"	1
		INSN: 0x4048b6 "mulsd xmm0, xmm1"	1
		INSN: 0x4048bf "addsd xmm0, xmm1"	1
	BBLK: 0x40495e	INSN: 0x404a8b "mulsd xmm0, xmm1"	1
		INSN: 0x404b71 "mulsd xmm0, xmm1"	1
		INSN: 0x404bde "mulsd xmm0, xmm1"	1
		INSN: 0x404be7 "addsd xmm0, xmm1"	1
	BBLK: 0x404c60	INSN: 0x404c9e "mulsd xmm0, xmm1"	1
INSN: 0x404ce6 "mulsd xmm0, xmm1"		1	
INSN: 0x404cef "addsd xmm0, xmm1"		1	
BBLK: 0x404d0d			
BBLK: 0x404fe2	INSN: 0x4050b4 "mulsd xmm0, xmm1"	1	
	INSN: 0x405121 "mulsd xmm0, xmm1"	1	
	INSN: 0x40512a "addsd xmm0, xmm1"	1	
BBLK: 0x4051a3	INSN: 0x4051e1 "mulsd xmm0, xmm1"	1	
	INSN: 0x405229 "mulsd xmm0, xmm1"	1	
	INSN: 0x405232 "addsd xmm0, xmm1"	1	
BBLK: 0x405250			
BBLK: 0x4052a3			
BBLK: 0x4052d9			
FUNC: 0x40538c "laplacian" [16 instruction(s)]			
FUNC: 0x42ad10 "main" [0 instruction(s)]			
FUNC: 0x42567a "transfb_nc1" [12 instruction(s)]			
FUNC: 0x420d08 "transfb" [24 instruction(s)]			
FUNC: 0x422600 "transfb" [52 instruction(s)]			
FUNC: 0x4246a9 "transfb_cor_e" [6 instruction(s)]			
BBLK: 0x4246d7	INSN: 0x424706 "mulsd xmm0, xmm2"	1	
	INSN: 0x42470a "addsd xmm0, xmm1"	1	
BBLK: 0x424743	INSN: 0x42477d "mulsd xmm0, xmm2"	1	
	INSN: 0x424781 "addsd xmm0, xmm1"	1	
BBLK: 0x4247ba	INSN: 0x4247ff "mulsd xmm0, xmm2"	1	
	INSN: 0x424803 "addsd xmm0, xmm1"	1	

Instruction counts: NONE (0) IGNORE (0) SINGLE (553) DOUBLE (32)

Configuration Search

- Helper script
 - Generates and tests a variety of configurations
 - Keeps a “work queue” of untested configurations
 - Brute-force attempt to find maximal replacement
- Algorithm:
 - Initially, build individual configurations for each module and add them to the work queue
 - Retrieve the next available configuration and test it
 - If it passes, add it to the final configuration
 - If it fails, build individual configurations for any child members (functions, basic blocks, instructions) and add them to the queue
 - Build and test the final configuration

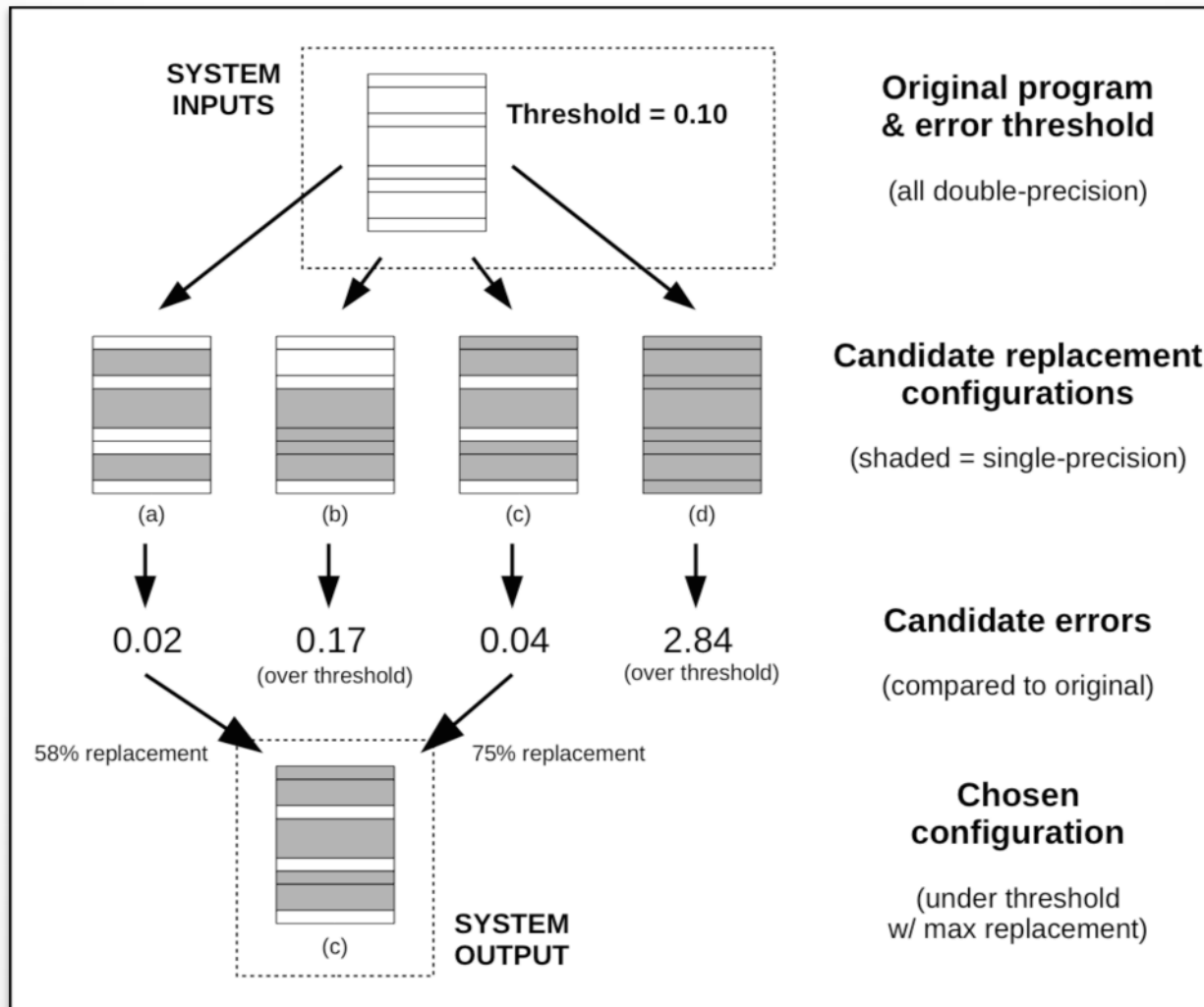
Demo

Observations

- Representation-dependent operations
 - Random number generators
 - “Libm” functions (log, exp, sin, cos, etc.)
- Accumulators
 - Multiple operations concluded with an addition
 - Requires double-precision only for the final addition
- Minimal dataset-based variation
 - Larger variation due to optimization level

Future Work

Automated configuration tuning



Conclusion

Automated instrumentation techniques can be used to implement **mixed-precision configurations** for floating-point code, and there is much opportunity in the future for **automated precision recommendations.**

Acknowledgements

Jeff Hollingsworth, University of Maryland (Advisor)

Drew Bernat, University of Wisconsin

Bronis de Supinski, LLNL

Barry Rountree, LLNL

Matt Legendre, LLNL

Greg Lee, LLNL

Dong Ahn, LLNL

<http://sourceforge.net/projects/crafthpc/>