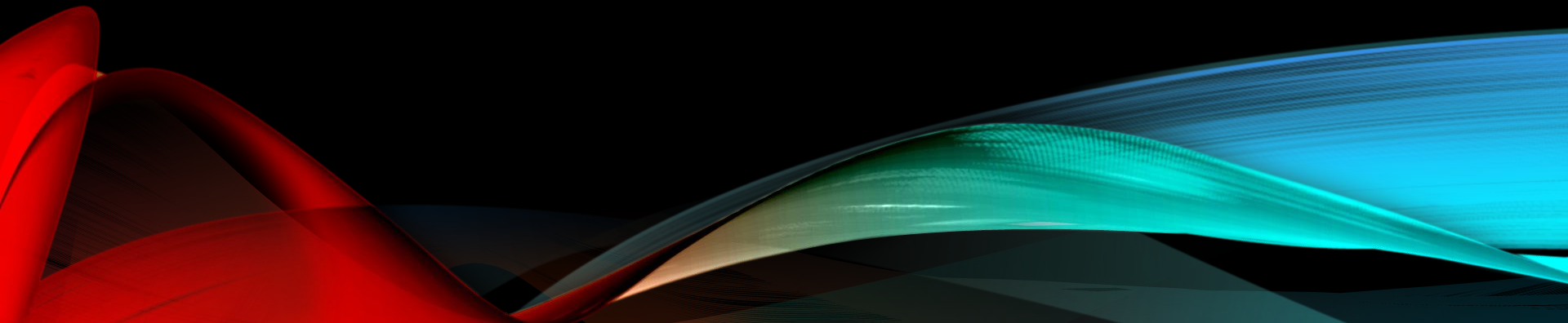
Approximate Computing

Nikolai Lenney jlenney

Charles Li cli4

18-742 S20

Load Value Approximation



Background

- Value Locality
  - Reuse of common values
  - Runtime constants and redundant real-world input data
- Load Value Prediction
  - On load, predict the value that is loaded
    - Skip fetch on next level cache/main memory and provide prediction
      - Save on energy and latency
    - Only works if exact match
    - Rollback speculative instructions on mismatch
      - Energy inefficient due to large buffers for rollback
      - Speed of rollback impacts performance
    - Floating point can be very difficult to predict correctly
      - High number of mantissa bits can lead to slightly incorrect values
      - 1.000 v 1.001 is a mispredict but is effectively the same value

Background

- Exact value comparisons lead to unnecessary rollbacks
  - Instead trade-off value integrity/accuracy for performance and energy
  - Load value approximator used to estimate memory values
- Many applications can tolerate inexactness
  - Image processing
  - Augmented Reality
  - Data mining
  - Robotics
  - Speech recognition
- Confidence window
  - How close is close enough? ±10%? ±5%?
  - Larger window gives better *coverage*
  - Performance-error tradeoff

Load Value Approximation

1. Load X misses in L1 Cache
2. Load Value Approximator generates X_Approx
3. Processor pretends X_Approx was returned on a "hit"
4. Main memory/next level cache fetches block with X_Actual (sometimes)
5. X_Actual trains Load Value Approximator

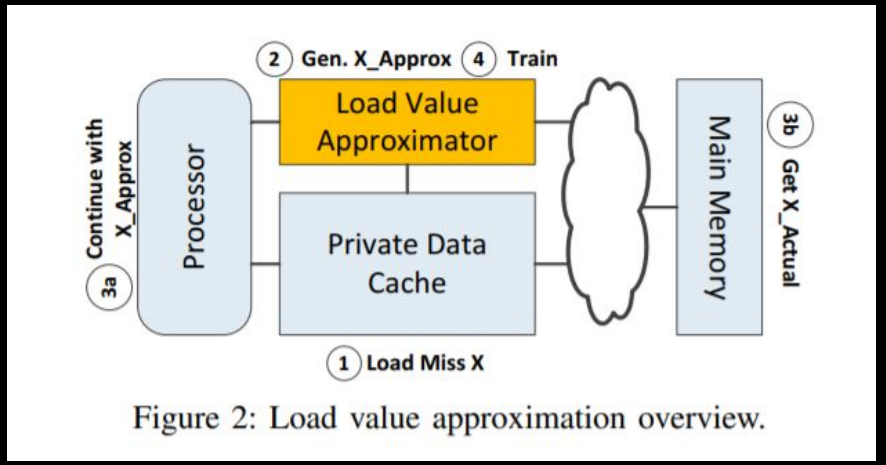


Figure 2: Load value approximation overview.

Load Value Approximator

- Global History Buffer (GHB)
  - FIFO queue storing most recently loaded values
- Approximator Table Entry
  - Accessed using hash on GHB values and Instruction Address
  - Tag
  - Saturating confidence counter
  - Degree counter
  - Local History Buffer (LHB)

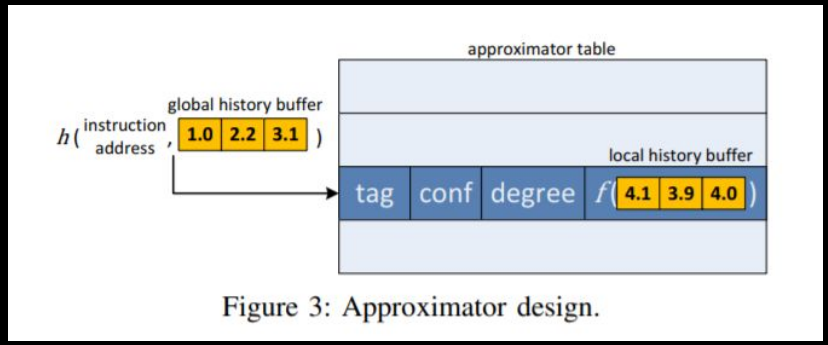


Figure 3: Approximator design.

Approximator Table

- Saturating Confidence Counter
  - Signed Counter
  - Use approximation if counter is positive
  - Increment/decrement based on accuracy of approximation
- Degree Counter
  - Number of times to reuse prediction before updating our table
  - Affects ratio of fetches to cache misses
- Local History Buffer (LHB)
  - Load values based on the global history buffer pattern & PC

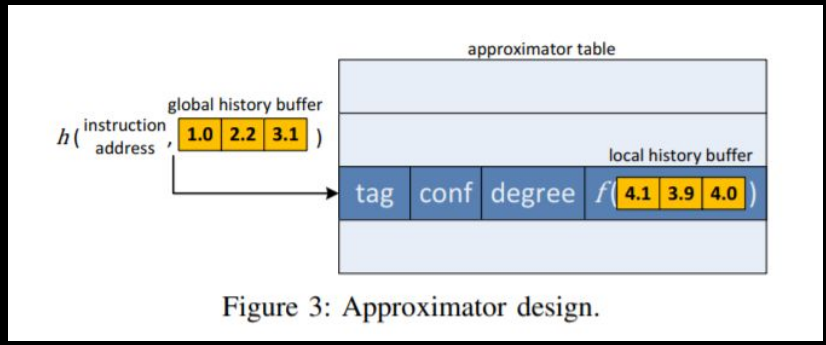


Figure 3: Approximator design.

Application

- Use ISA extensions to support load value approximation
- Programmers annotate code
- Do not use approximation for
  - Control Flow
    - Can cause incorrect behavior
    - x == 42 approximation is bad
  - Divide-by-Zero
    - Data in denominator could be approximated as 0
  - Memory Addresses
    - Can read from/write to incorrect memory addresses
    - “Catastrophic results”

Application

- Do use approximation for the Common Case
  - Expensive loops/functions
  - Corner cases likely not going to add much value
  - Programmer must profile their own code
    - Find accesses where cache misses occur
    - Find places where approximate data is usable
      - Likely only in small regions of code since approximate in one context does not imply approximate in all contexts

Evaluation Tactics

- Metrics
  - Misses-per-kilo-instructions (MPKI)
  - Blocks fetched (L1 only)
  - Output Error
- Design space exploration
  - GHB size
  - Confidence threshold
  - Value Delay
  - Approximation Degree

Benchmark	L1 MPKI	Instruction count variation
blackscholes	0.93	0.99%
bodytrack	4.93	0.05%
canneal	12.50	1.25%
ferret	3.28	0.60%
fluidanimate	1.23	0.17%
swaptions	4.92E-05	0.00%
x264	0.59	2.37%

Table I: Precise L1 MPKI and variation in dynamic instruction count when employing load value approximation.

Design Space Exploration

- GHB Size
  - Smaller GHB tends to have larger output error
  - Smaller GHB tends to have fewer MPKI
- Simple, low-overhead approximators work well

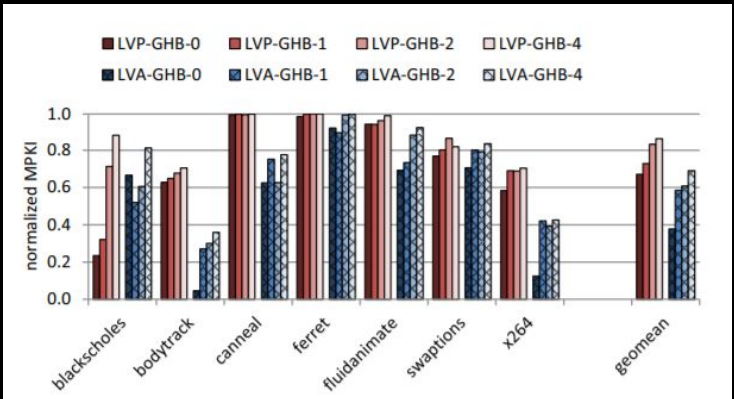


Figure 4: LVA vs. an idealized LVP for different GHB sizes.

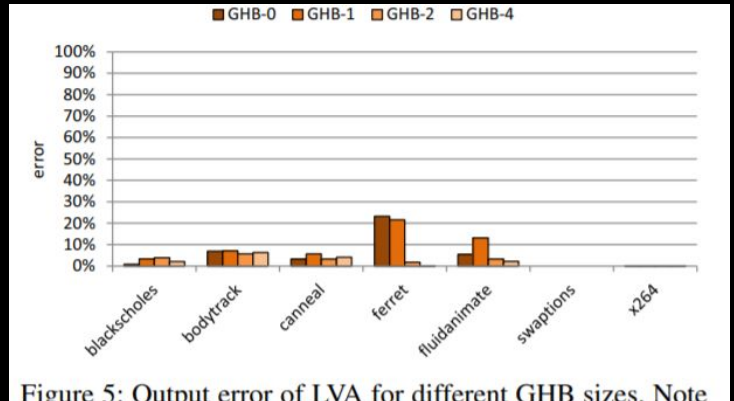
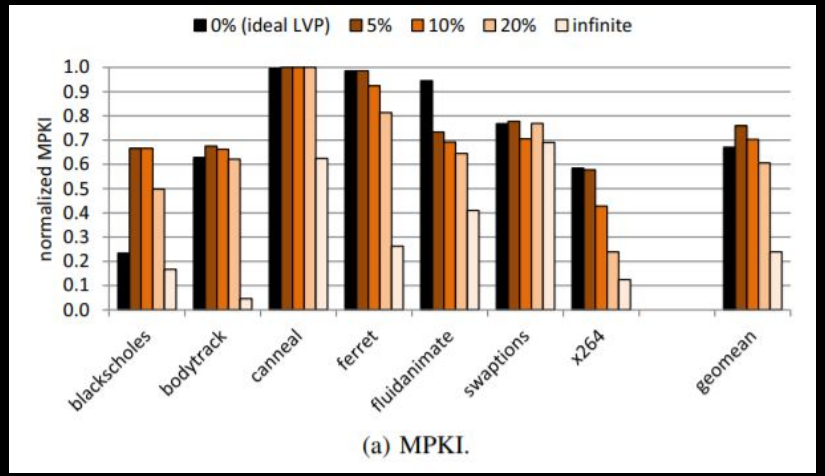


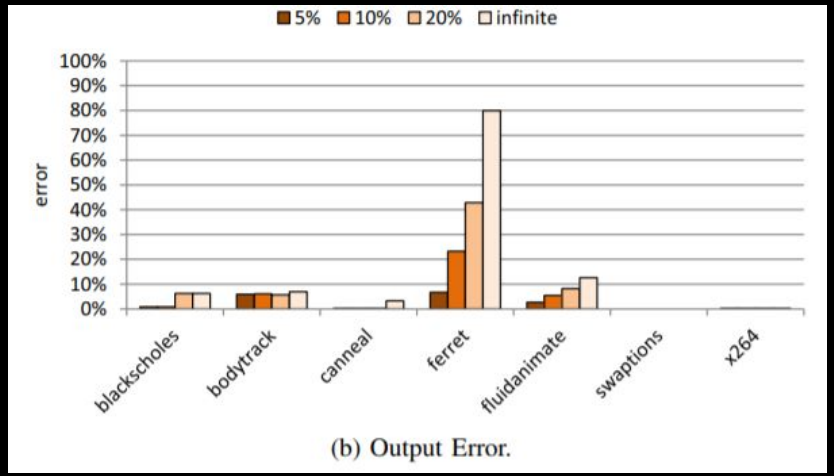
Figure 5: Output error of LVA for different GHB sizes. Note the near zero error for swaptions and x264.

Design Space Exploration

- Confidence Window
  - Larger window typically means more error
  - Larger window typically means fewer MPKI
- Integers are better for approximation than floats



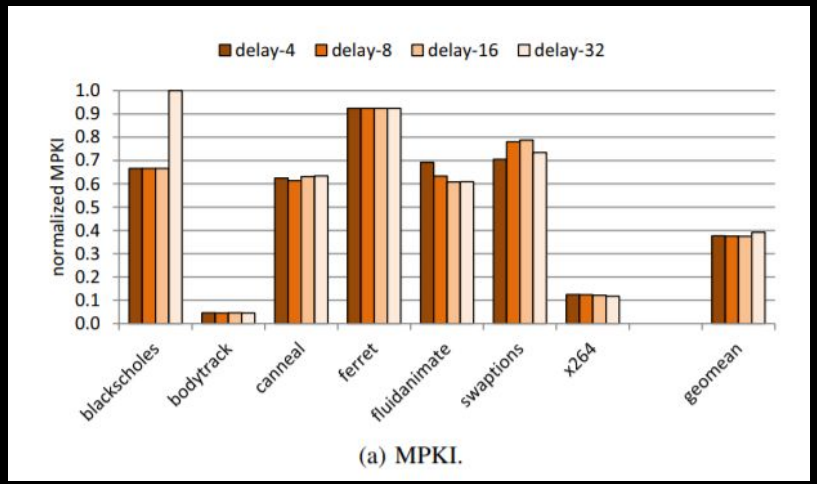
(a) MPKI.



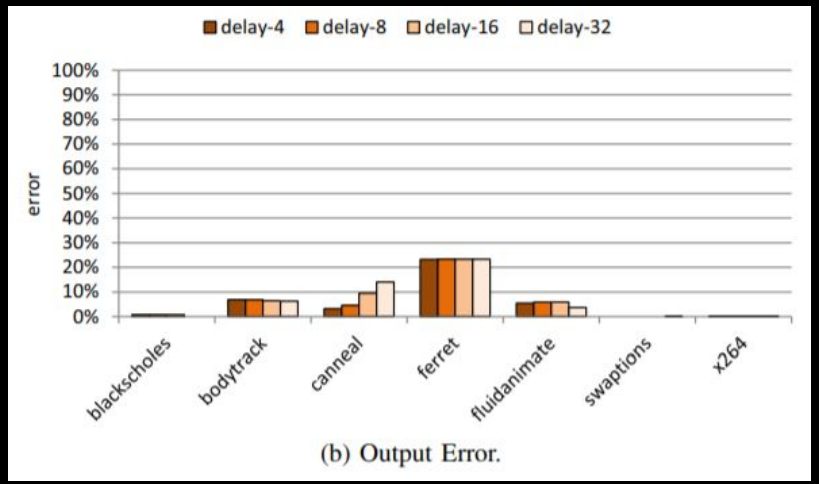
(b) Output Error.

Design Space Exploration

- Value Delay
  - LVA is highly robust with regards to value delay
  - No impact on performance since confidence is not changed
  - No impact on error due to lack of inter-dependence between data



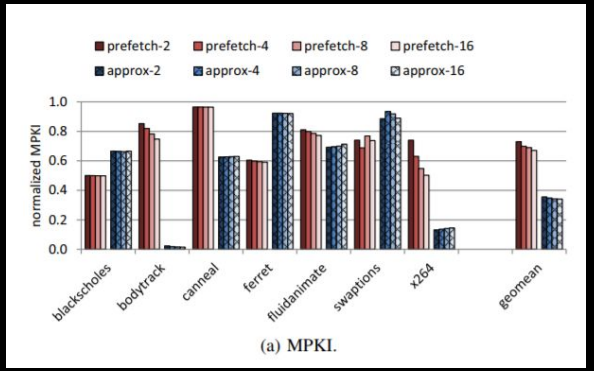
(a) MPKI.



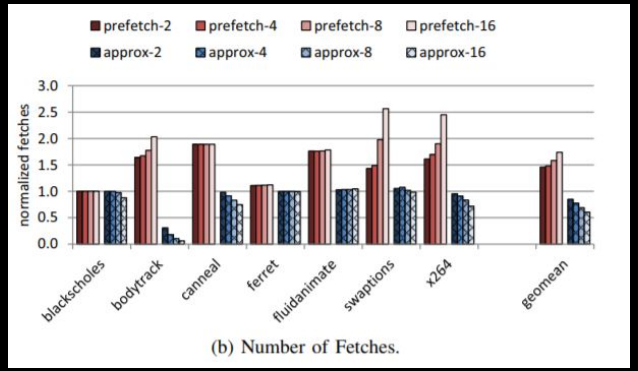
(b) Output Error.

Design Space Exploration

- Approximation Degree
  - More prefetches lowers MPKI but increases overall fetches
  - Higher approximation degree increases output error due to less training



(a) MPKI.



(b) Number of Fetches.

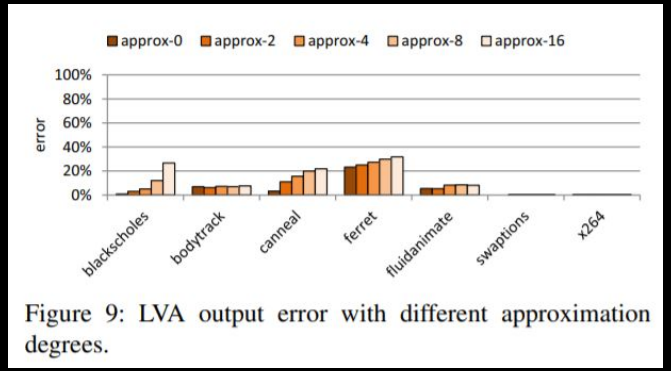
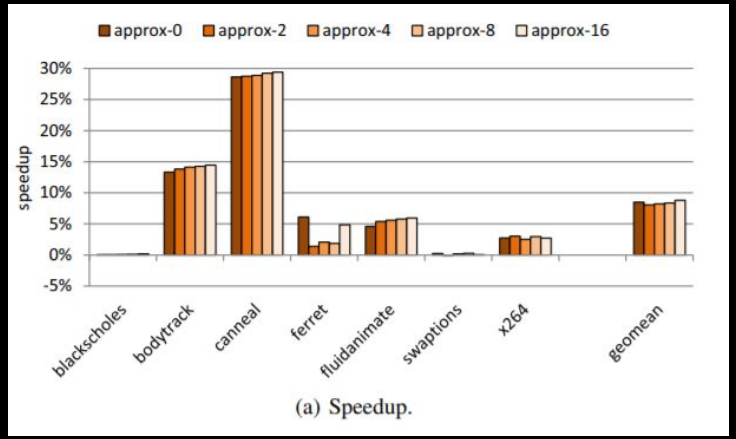


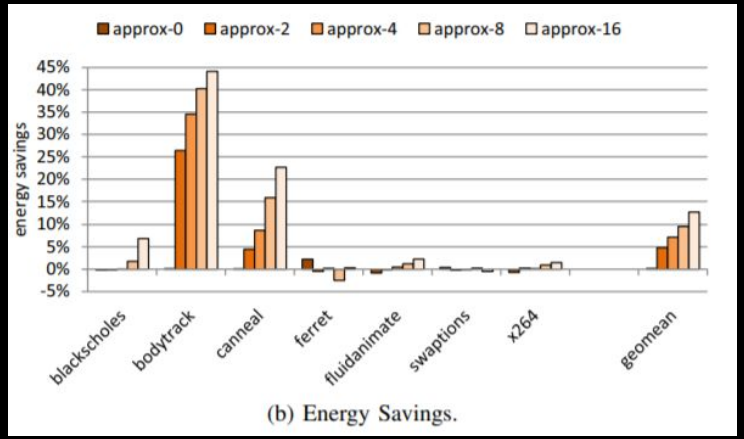
Figure 9: LVA output error with different approximation degrees.

Results

- Gives realistic value delay (~1 as opposed to presumed 4)
- Improve performance by an average of 8.5%
- Reduce L1 miss latency by 41% on average
- Reduce EDP by up to ~64% depending on approximation degree
- Energy savings ~7-12% depending on approximation degree



(a) Speedup.



(b) Energy Savings.

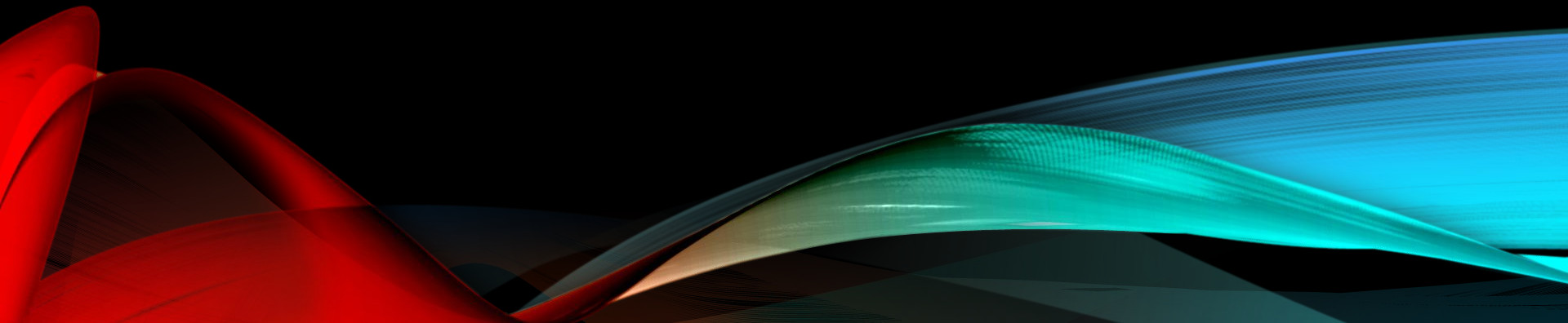
Discussion

- Overhead introduced by approximator table is ~18KB (64bit) or ~10KB(32bit)
- No approximation of application data leads to small GHB being optimal
- Approximator can use fewer mantissa bits for floating point values to improve hashing
- Memory consistency can be problematic, should not use LVA for applications with a need for memory consistency

Pros and Cons

- Pros
  - Provides for good trade-off in accuracy and energy, especially since accuracy is not needed all the time
  - Very simple design to add to a basic pipeline, with minimal ISA extensions (seems to only need to identify approximable loads)
  - Clearly identifies when this is usable and when it is not
- Cons
  - Has a very small test set and leaves many optimizations for future work
  - Can still have significant inaccuracy (see Ferret benchmark)

Neural Acceleration for General-Purpose Approximate Programs



Background

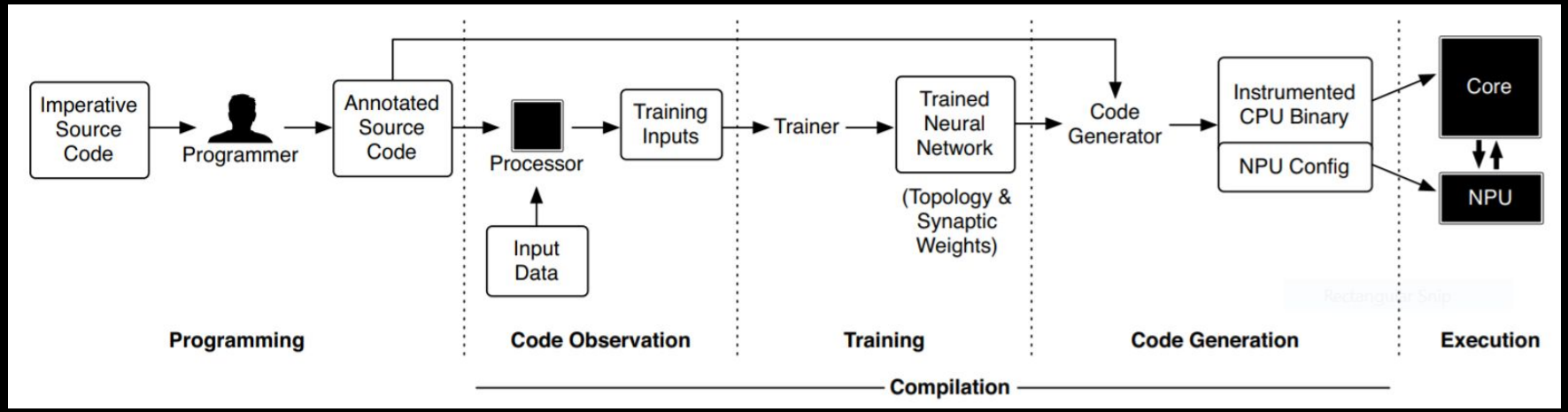
- Many applications are highly error tolerant, and can be approximated
  - Image processing, Augmented Reality, Data mining, Robotics, Speech recognition
- Neural Networks are highly effective at finding patterns in input data and correlating these to output values
  - Recall that running a Neural Network involves a series of matrix/vector operations and nonlinear functions.
- If we can approximate memory lookups, arithmetic, simple control flow, why not try to approximate entire sections of code?
- Many functions are used frequently and take a long time/a lot of energy to run, but are also very predictable with a neural network

Code Region Criteria

- Hot Code
  - Want to focus on regions of code that are frequently executed and that take up a large portion of a program's total runtime
  - Regions that are too small may suffer from overheads
- Approximability
  - Programs needs to be able to tolerate imprecision
  - Translating a region to a NN is the compiler's job, not the programmers
- Well-Defined Inputs & Outputs
  - Region must have a fixed number of inputs and outputs
- Pure
  - Must not access values from outside of the region, except for the inputs and outputs

Parrot Overview

- Programmer identifies and marks functions to be approximated
- Annotated code is run by a profiler to generate NN parameters
- Profiler gives new source code that replaces function calls with NN instantiations



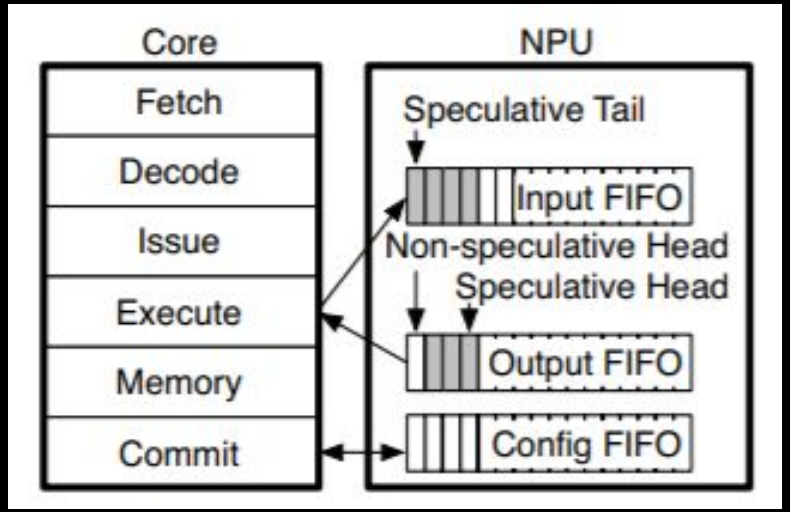
Training

1. Programmer gives profiler a set of valid application inputs for training
2. Application collects function inputs/outputs as training/testing data
3. Uses a simple search through 30 possible NN topologies guided by mean squared error
   - 1 or 2 hidden layers
   - Each layer can have 2, 4, 8, 16, or 32 hidden units
   - Choose topology with highest test accuracy and lower NPU latency, but prioritizing accuracy
4. Generate a binary that instantiates the NPU with the determined topology and weight

- Could also use online training, but this would incur high overheads at runtime.

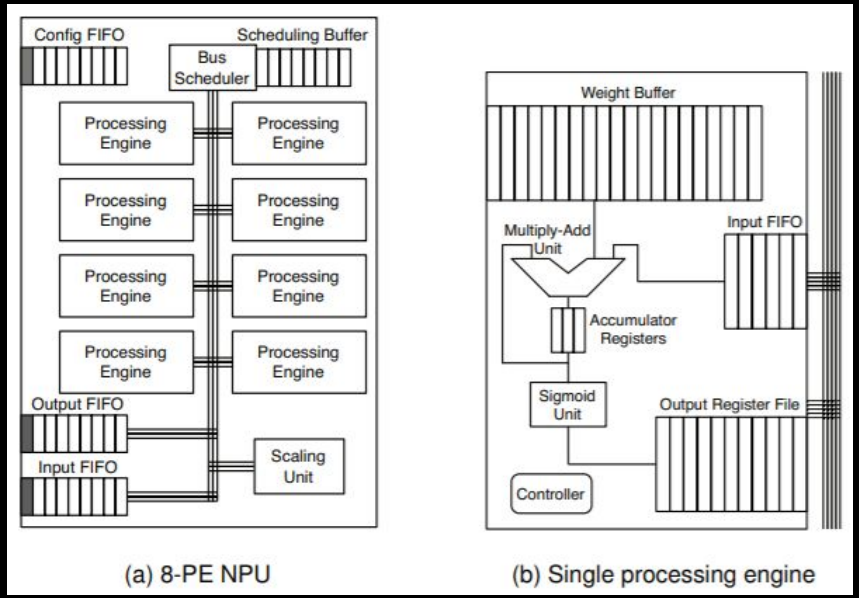
ISA

- Neural Processing Unit is tightly coupled with out of order pipeline.
- ISA includes 4 instructions for interfacing with NPU
  - **enq.c %r**: writes a value to config FIFO
  - **dec.c %r**: reads a value from config FIFO
  - **enq.d %r**: writes a value to input FIFO
  - **deq.d %r**: reads a value from output FIFO
- NPU supports speculative data reads and writes
- Can be made to work with interrupts and context switches



NPU Overview

- NPU is run by a static schedule given by the configuration
- The scheduler takes the following steps for each layer:
  - Assign each neuron to a PE
  - Assign order of multiply add ops.
  - Assign order to outputs of layer
  - Produce a bus schedule according to the assigned order of ops.



Benchmarks

	Description	Type	Evaluation Input Set	# of Function Calls	# of Loops	# of ifs/ elses	# of x86-64 Instructions	Training Input Set	Neural Network Topology	NN MSE	Error Metric	Error
fft	Radix-2 Cooley-Tukey fast Fourier	Signal Processing	2048 Random Floating Point Numbers	2	0	0	34	32768 Random Floating Point Numbers	1 -> 4 -> 4 -> 2	0.00002	Average Relative Error	7.22%
inversek2j	Inverse kinematics for 2-joint arm	Robotics	10000 (x,y) Random Coordinates	4	0	0	100	10000 (x,y) Random Coordinates	2 -> 8 -> 2	0.00563	Average Relative Error	7.50%
jmeint	Triangle intersection detection	3D Gaming	10000 Random Pairs of 3D Triangle Coordinates	32	0	23	1,079	100000 Random Pairs of 3D Triangle Coordinates	18 -> 32 -> 8 -> 2	0.00530	Miss Rate	7.32%
jpeg	JPEG encoding	Compression	220x200-Pixel Color Image	3	4	0	1,257	Three 512x512-Pixel Color Images	64 -> 16 -> 64	0.00890	Image Diff	9.56%
kmeans	K-means clustering	Machine Learning	220x200-Pixel Color Image	1	0	0	26	50000 Pairs of Random (r, g, b) Values	6 -> 8 -> 4 -> 1	0.00169	Image Diff	6.18%
sobel	Sobel edge detector	Image Processing	220x200-Pixel Color Image	3	2	1	88	One 512x512-Pixel Color Image	9 -> 8 -> 1	0.00234	Image Diff	3.44%

Error CDF

- Most applications have close to or well over 50% of their inputs hitting 5% error or less
- Every application has over 80% of their inputs hitting 10% error or less
- NN error will likely be in tolerable error range for many applications

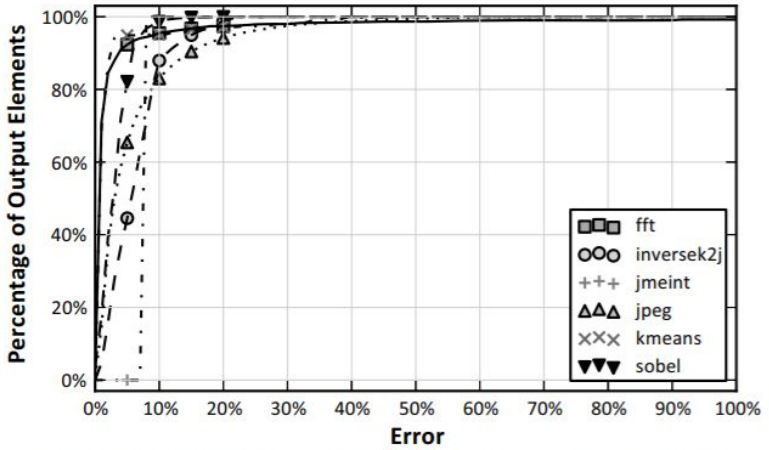
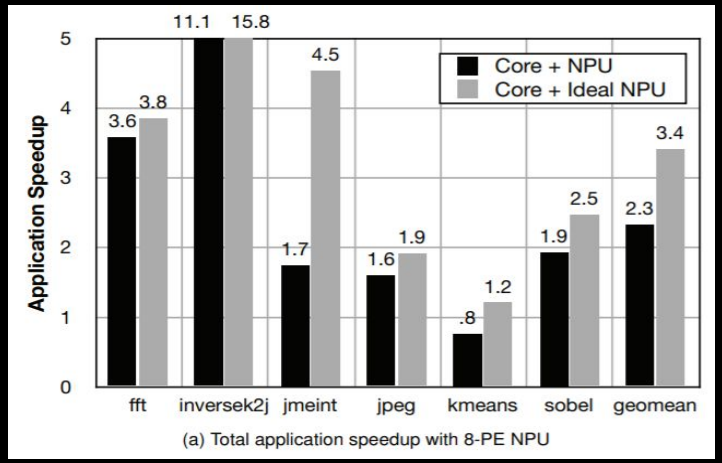


Figure 6: Cumulative distribution function (CDF) plot of the applications' output error. A point (x, y) indicates that y fraction of the output elements see error less than or equal to x.

NPU Speedup vs Software Slowdown

- Running a neural network in software to approximate something else in software is not really an option, and would likely only work well for a very long running region of code that could be approximated by a relatively small NN.



(a) Total application speedup with 8-PE NPU

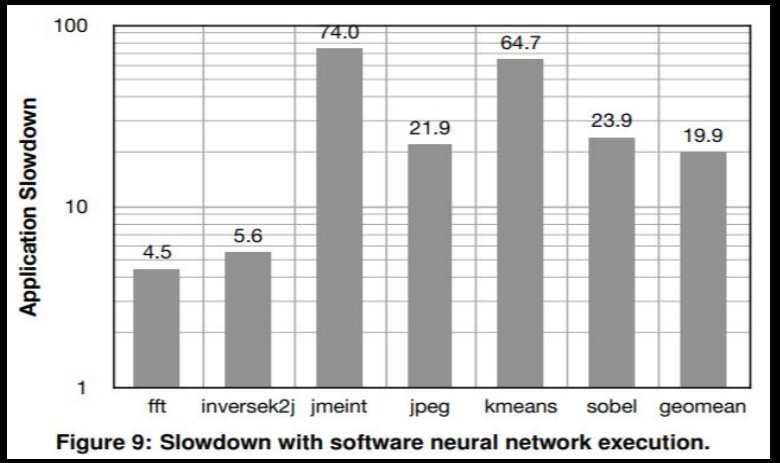


Figure 9: Slowdown with software neural network execution.

Number of Instructions vs Energy vs Speedup

- Energy savings tightly correlated to speedup and inversely correlated to number of instructions
- jmeint has the highest proportion of NPU instructions, and the largest discrepancy in realistic/idealized NPU performance
- Executing fewer instructions does not imply speedup

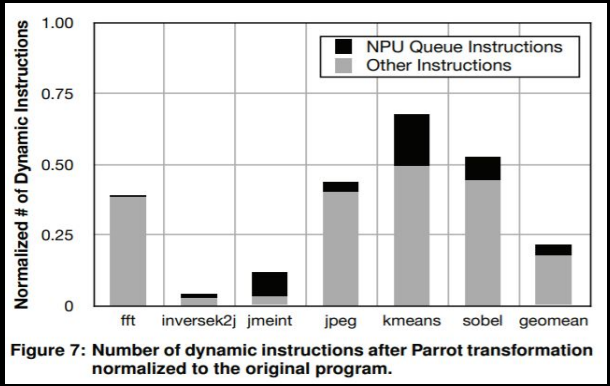
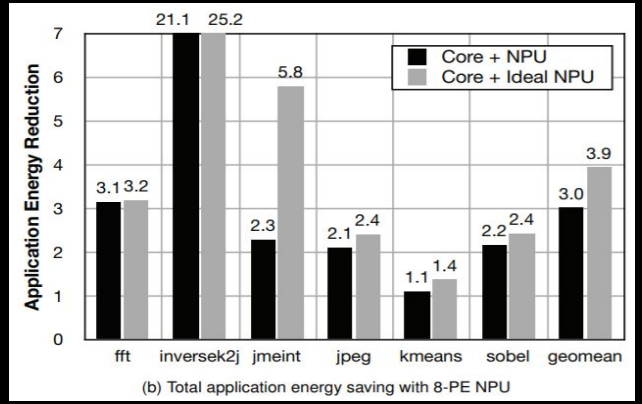
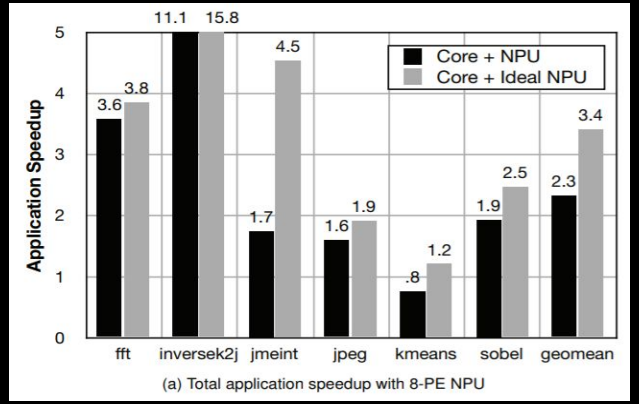


Figure 7: Number of dynamic instructions after Parrot transformation normalized to the original program.



(b) Total application energy saving with 8-PE NPU



(a) Total application speedup with 8-PE NPU

NPU Latency

- NPU still improves performance even if it takes longer to access
- Could be useful if architecting an NPU very tightly with a core is impractical
- Could make NPU access via memory mapped FIFOs feasible

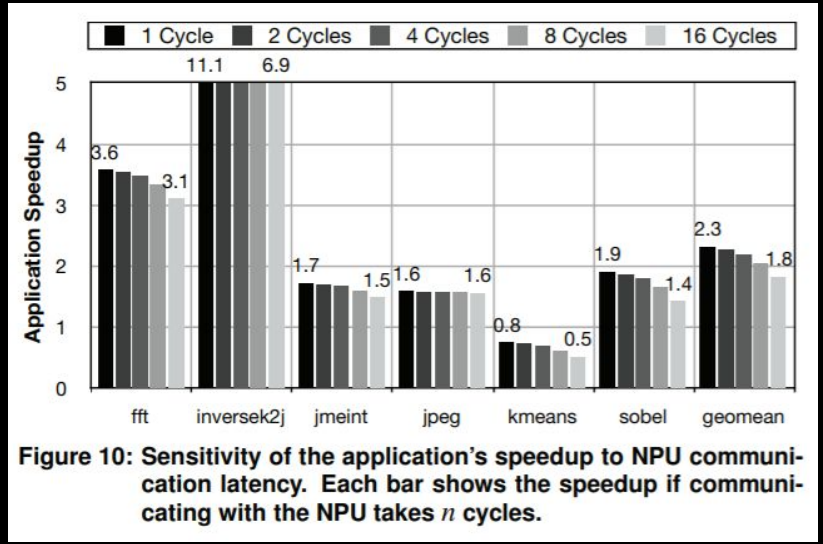


Figure 10: Sensitivity of the application's speedup to NPU communication latency. Each bar shows the speedup if communicating with the NPU takes n cycles.

Number of NPU PEs

- Neural Nets have a lot of inherent parallelism, so naturally more PEs means more speedup
- Increasing PEs can give speedup, but the amount of speedup per leap decreases, and per added PE is much smaller
- Adding a PE is not free - Must pay for it in energy, area, complexity

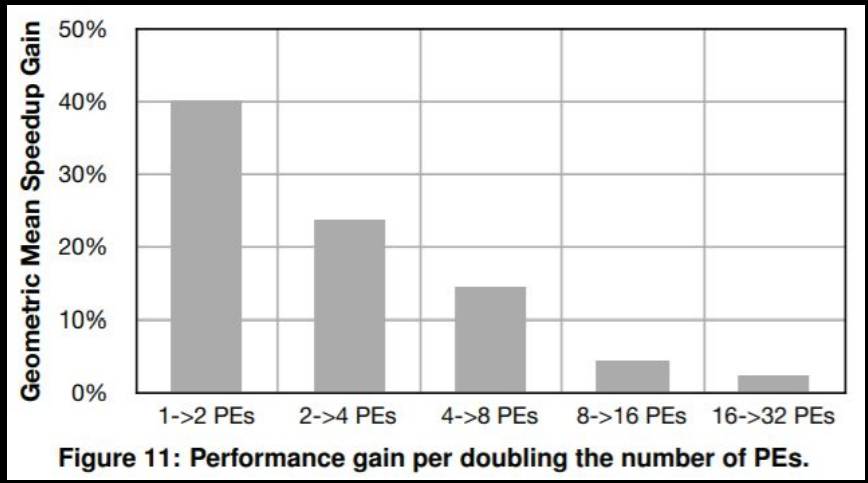


Figure 11: Performance gain per doubling the number of PEs.

Discussion

- Not super cheap (~85KB) but most of this comes from 8 8KB sigmoid LUTs, which can presumably be shared, bringing the total down ideally to about ~28KB. We saw that this can also be tolerate relatively high latencies, so the area costs may be tolerable.
- Doesn't interfere with memory model
- Existing strategies can be used to make the NPU work in a realistic environment with context switches and interrupts, though these may harm performance
- Theoretically one could use the NPU as a small NN accelerator if one is already running a neural network.

Pros and Cons

Pros

- Can dramatically reduce number of executed instructions, as well as runtime and energy.
- Expected error is often low, and many application areas can deal with impressicion

Cons

- Can only get benefit if application area tolerates error, if functions can be approximated well, and if function is actually used enough/long enough to justify NPU envocation
- Programmer must annotate code and provide training examples