THE UNIVERSITY OF CHICAGO

NEAT: A TOOL FOR AUTOMATED EXPLORATION OF APPROXIMATE FPU DESIGNS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
LEE EHUDIN

CHICAGO, ILLINOIS
MAY 25

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Much recent research is devoted to exploring tradeoffs between computational accuracy and energy. In particular, a number of techniques have been proposed for producing and using approximate arithmetic units that return an inexact answer with greatly reduced energy consumption. As the number of approximate techniques increases, the options for creating approximate programs explodes, creating the need for tools that help programmers explore the effects of approximation and combine different approximation techniques to achieve the lowest energy consumption for an accuracy constraint or the best accuracy for an energy constraint. To address this need, we present NEAT: a PIN tool that automatically explores the accuracy-energy tradeoff space for floating-point computation. NEAT accepts one or more user-defined approximate floating-point implementations and rules for when to substitute different implementations. NEAT then computes the floating-point operations in an application using those implementations and rules. We evaluate NEAT through a case study on 8 different applications and compare a set of rules that allows only one floating-point implementation per program to a set of rules that allow one approximation per function. We find that more of the accuracy-energy design space can be explored with the per-function rules than the single floating-point implementation. We also find that data collected from smaller inputs using both sets of rules is highly correlated to data collected from moderately-sized inputs.

CHAPTER 1

INTRODUCTION

Approximation has become an important design consideration for computing systems, as numerous researchers have found ways to trade reduced program accuracy for reduced resource usage—typically, time or energy. Early work in this area demonstrated the tremendous energy and execution time reductions that approximation techniques can acheive [6]. As this work has matured, it has transitioned from fairly coarse-grained techniques—such as pruning logic from funtional units [29] or running at near-threshold voltage—to more sophisticated approaches that make a variety of functional units available—each with different levels of approximation [10, 11].

The proliferation of different approximate functional units on a single core creates tremendous opportunity, but it also creates a new problem. Specifically, how do programmers decide which level of approximation to use at different points in their application? Just considering moderate sized programs with 10 functions and 10 different levels of approximation, we already have an intractably large design space. Typically, the decision of which approximation to use where would be made by an expert in numerical methods—who could pick the optimal accuracy level for each point in the program to ensure the maximum energy (or run time) savings. It is not reasonable, however, to ask software engineers to acquire a new skill set so they can benefit from the growth of approximate computing. Similarly, it is infeasible to do brute force exploration to find the best approximation per function for any reasonably sized application.

We observe that most work on approximation has been about exploring new alternatives for approximation across the system stack, but very little work has been done helping programmers navigate the huge tradeoff space enabled by allowing multiple approximations within a single program. This observation motivates us to propose NEAT—Navigating Energy Approximation Tradeoffs—a tool that helps users explore different levels of approximation within a program. NEAT accepts a user program, a set of approximate floating point implementations, and a set of programmable rules for when to use a specific implementation. A rule, for example, could specify that an FFT function uses an approximate FPU, while a singular value decomposition uses a full accuracy implementation. In general, the rules are quite flexible and can be conditional on the program state. NEAT then runs the program and dynamically replaces floating point instructions with the approximate version as specified by the rules. NEAT outputs the program's output, plus the number of bits used, the floating point operations used at each point in the program, and the total number of FLOPs per function. Thus, NEAT can be used as a tool to explore the tradeoff space of approximate programs without requiring deep numerical expertise.

We impelment NEAT for x86 using the Pin binary instrumentation system [20]. We demonstrate NEAT's value through a case study where we compare the approximations produced by two different rule sets. In the first, we simply pick one floating point implementation for the entire program; *i.e.,* the rule is a simple one-to-one replacement. in the second, we allow the top 10 most commonly executed functions to each use a different approximation. In this second scenario, we use a genetic algorithm to guide NEAT's exploration of the enormous resulting search space.

Our results show that the per-function configurations are able to explore more of the FPU combination design space than the whole-program configurations. This proves that our tool provides value to programmers. Our results also show that data collected from smaller training inputs is highly correlated with data collected from reasonably-sized inputs. This makes it easier to use NEAT because smaller inputs can be used to iteratively find optimal points in the design space in less time than moderately-sized inputs.

In summary, this paper makes the following contributions:

- The NEAT framework that helps users explore the design space of FPU combinations.

- A case study that compares whole-program FPU configurations with per-function FPU configurations for a number of benchmarks.

- A discovery of the correlation of the accuracy of small inputs with the accuracy of moderately-sized inputs for a number of benchmarks when instrumented with NEAT.

CHAPTER 2
BACKGROUND & MOTIVATION

2.1 Prior Work

There has been a substantial amount of effort to reduce the accuracy in programs in order to save power. However, there is a lack of solutions that allow the user to specify their own floating-point arithmetic implementations. This paper provides such a solution.

Approximation Knobs [16] provide a way to lend performance and energy gains to existing power knobs. These gains could be achieved by altering the accuracy level of applications from a set of variable accuracy implementations of the application. A machine learning algorithm is used to dynamically switch between these implementations based on a power cap. However, our proposal lets users examine and change the accuracy of floating-point operations manually, giving them more control over the floating-point computations in a program.

Quora [31] is a quality programmable processors where the notion of quality is codified in the instruction set of the processor. This is similar to our proposal, since we aim to make the quality of floating-point arithmetic programmable. However, Quora uses precison scaling to automatically modulate the accuracy of instructions, whereas we give users tools to examine and change the accuracy of floating-point operations manually. Additionally, Quora does not target floating-point arithmetic operations and our propoal does.

ApproxHadoop [14] provides a framework for creating and running approximation-enabled MapReduce programs. This system allows user-defined approximation: the user can provide a precise and approximate version of the code for a MapReduce task. This is similar to our proposal, but ApproxHadoop is specialized for the domain of MapReduce programs, whereas our proposal can be used on any binary performing floating-point arithmetic.

Another example of user-defined approximation is Green [3], which is a system that allows

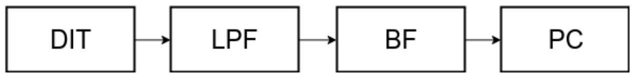


Figure 2.1: Design of Radar Application

programmers to supply approximate versions of loops and while-blocks that terminate early. However, this tool targets loops and while-blocks for approximation, whereas our tool targets floating-point arithmetic.

A growing body of recent work explores a variety of approximation techniques to save power. These techniques include (a) approximate storage designs [19], (b) voltage over-saling [11, 23, 5, 25, 15], (c) memoization [1, 2], (d) limited fault recovery [7], (e) approximate circuit synthesis [32, 28, 26, 29, 17, 18, 9], (f) neural acceleration [12, 30, 22, 24, 33], and (g) probabalistic computing [27]. Although these solutions report promising power benefits from approximate computing, they do not explore replacing floating-point arithmetic operations with user-defined implementations.

We can see from all of these proposals that approximation is an important part of program design. However, there needs to be more help with exploring approximation design spaces so that users can make more informed decisions about how to approximate and how much to approximate. Most new developments in approximate computing are aimed towards finding new forms of approximation, not helping users with approximation. Thus, this space is rich for exploration.

2.2 Motivating Example

Consider a synthetic aperture radar. The software for such a synthetic aperture (shown in Figure 2.1) is an embedded system, and must meet strict power constraints. Embedded system designers are used to hand-optimizing accuracy to meet these strict constraints. Typically they just pick one bit width for the whole application. Picking the right bit width requires both computer science and numerical analysis skills.

Table 2.1: Possible FPU combinations for radar application when one FPU is used for the entire application or when one FPU is used for each function in the application.

	Whole Program	Per Function
Possible Implementations	24	1.7e26

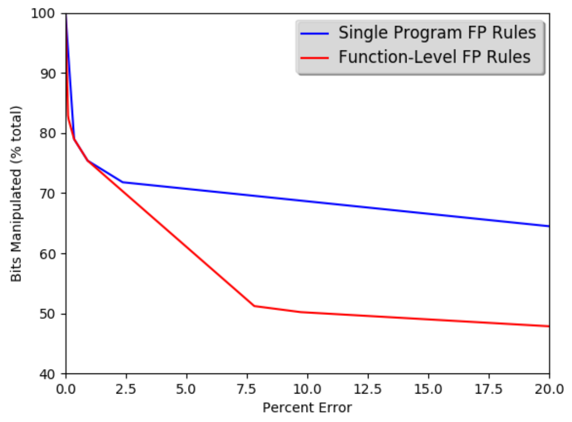


Figure 2.2: Percent Error vs. Bits Used for Radar Application

Recent proposals advocate putting many different approximate FPUs on a single core [13]. These approximate FPUs can make meeting power constraints easier, but introduce more complications when programming. The challenge in a complicated program is how to figure out which FPU to use in each part of the program. Previous results [10] have shown that using a different FPU for each function in a program allows a user to explore more of the approximation design space than only using one FPU for the entire appplication, but this requires tedious hand-tuning.

Returning to our radar aperture, we can see an example of this challenge. Consider just FPUs that decrease the number of bits used in the mantissa of each floating-point number in an application. There are 23 bits used in the mantissa of a 32-bit IEEE floating-point number, so there are 24 possible FPUs of this type. Even if we consider using only one FPU per function in the radar application, the size of the design space of FPU combinations is huge, as can be seen in Table 2.1. Further, all of these different FPU combinations would have to be coded by hand, which can use a large amount of programmer time and be error-prone.

One way to address this challenge is to restrict the design space of FPU combinations for an application. For example, the number of FPU combinations for the radar application could be shrinked by looking only at using one FPU for the entire application instead of one FPU for each function of the application. However, after measuring the accuracy and power usage of our radar application for both whole program combinations and per-function combinations (shown in Figure 2.2), we can see that this solution will restrict the size of the design space that we explore. If we explore more of the design space, we can find more optimal values for power usage and accuracy loss.

CHAPTER 3
SYSTEM DESIGN

Prior work has shown that there is a benfit to exploring the design space of different FPU combinations in a program [10], and we have shown that this design space is huge. We propose a tool to explore this space of approximations. This tool, named Navigating Energy and Accuracy Tradeoffs (henceforth referred to as NEAT) allows users to collect data from applications using custom implementations of floating-point arithmetic. This provides a fast and easy way to collect data from the design space of FPU combinations for different applications.

The design of this tool is shown in Figure 3.1. The main purpose of this tool is to allow users to replace floating-point arithmetic operations in applications. Users can specify multiple implementations of floating-point addition, subtraction, multiplication, and division. Further, the tool allows users to provide rules that decide which implementation to use for each operation. These rules are supplied contextual program information obtained from NEAT to help them make decisions. This contextual information includes the type of operation, the operands, and the name of the function containing the operation. The rules can also include callbacks that NEAT executes when functions are entered and exited in the application. These callbacks can be used to keep track of more program information, such as the current call stack of the program.

3.1 User Inputs

There are three user inputs to this tool: a user application to insturment, the desired floating-point arithmetic implementations to use in the application, and a set of rules to choose which floating-point arithmetic implementation to use for each operation.

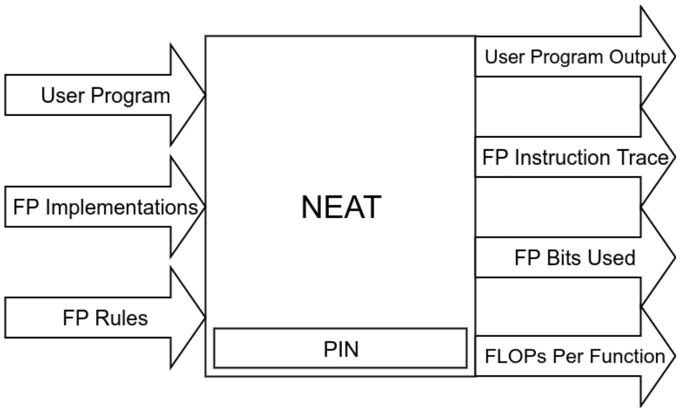


Figure 3.1: NEAT Design

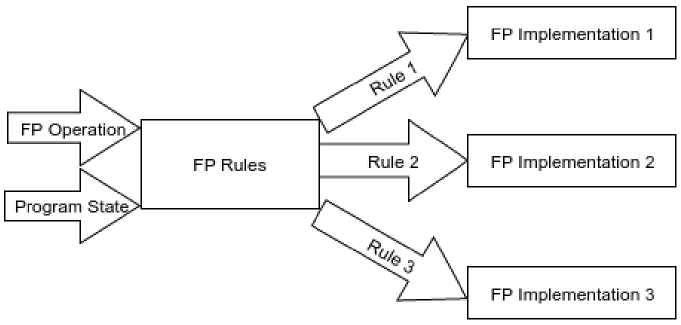


Figure 3.2: FP Rules Design

3.2 Floating-Point Rules

Defining an implementation of floating-point arithmetic is fairly trivial. The computation performed by each arithmetic operator can be defined as a function. The main challenge with changing floating-point arithmetic implementations dynamically is the way to specify which floating-point arithmetic implementation to use for each operation. NEAT addresses this challenge through a set of FP rules that can be supplied by the user. The design of the FP rules are shown in Figure 3.2. Whenever a floating-point arithmetic operation should be computed, NEAT captures information about the current state of the application. This information is provided to the FP rules supplied by the user. NEAT then uses these rules to determine which floating-point arithmetic implementation is used to calculate the result of that operation.

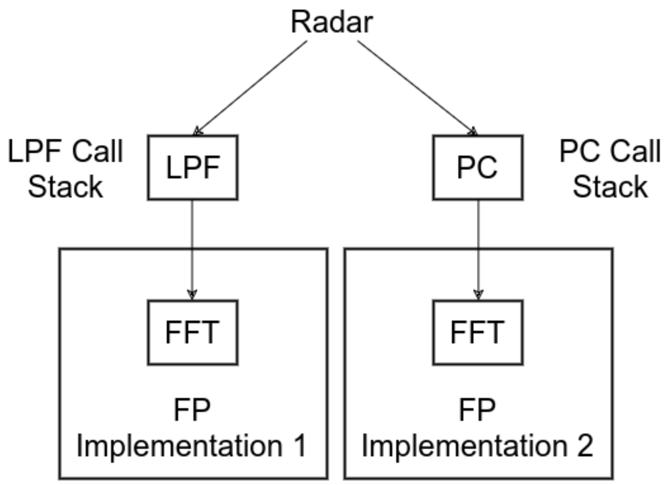


Figure 3.3: FP Rules Example

We show an example of the versatility of these rules in the context of our radar application from earlier in Figure 3.3. The radar application from earlier includes both a low-pass filter (LPF) and pulse compression (PC). Both of these components use a fast fourier transform (FFT). FP rules can be used to keep track of the call stack of the radar application, and use one FP implementation for the FFT in the LPF, and a second FP implementation for the FFT in the PC. This can all be done in a handful of lines of code.

3.3 NEAT Outputs

There are four outputs from this tool: the output from the user application, a trace of the operands and result of every FLOP executed by the program, the total number of bits used in FLOPs in the execution of the program, and the number of FLOPs executed per function in the program.

The trace of the FLOPs executed by the instrumented application is written to a file while the application is running. If floating-point implementations are supplied to NEAT by the user, the result of each operation will be printed after the operation is calculated with the chosen floating-point implementation. The operands and result of each operation are printed as hexadecimal numbers so that there is no confusion in rounding the floating-point values.

The total number of bits used in FLOPs in the execution of the program is output to a file after the application has finished running. The count that is output is the total number of bits manipulated in the operands and result of every FLOP in the instrumented program. This can be used as a platform-independent way to evaluate the approximate amount of power used by FLOPs when instrumenting a program. Since the number of bits used in the exponent and sign of a floating-point number vary much more than the number of bits used in the mantissa of a floating-point number, NEAT only calculates the number of bits manipulated in the mantissa. NEAT calculates the number of bits not manipulated as the number of zeroes in the binary representation of the floating-point number, starting with the least significant bit.

3.4 Implementation

The NEAT dynamic instrumentation tool was written in C++ using the Intel Pin instrumentation system [20]. NEAT performs run-time instrumentation to facilitate the analysis and replacement of floating-point arithmetic operations during the execution of compiled C and C++ binaries. Our implementation is publically available at `https://github.com/NeatTool/NEAT`.

3.4.1 Pin Instrumentation System

The Pin instrumentation system was chosen as the backbone for this tool because of its clean API and efficient implementation. The Pin API makes it possible to write instrumentation routines to observe and alter the architectural state of a process. Pin uses a JIT compiler to generate new instrumented code that can be executed without extra run-time overhead from instrumentation.

3.4.2 Floating-Point Operations

For the purposes of this tool, we identify floating-point arithmetic operations as the Streaming SIMD Extensions (SSE) instructions for scalar arithmetic. These instructions are included in a SIMD instruction set extension to the x86 architecture and operate on 32-bit single-precision floating point numbers. More specifically, the instructions we use for our definition of floating-point operation are `ADDSS`, `SUBSS`, `MULSS`, and `DIVSS`.

3.4.3 Floating-Point Rules

NEAT allows users to define rules that determine which floating-point implementation is used to calculate each FLOP in a program. Every time a FLOP is about to be calculated in the user application, NEAT will examine all of the rules that have been supplied by the user, and use them to determine which floating-point implementation will be used to calculate the result of the FLOP. The user can also register callbacks through NEAT that can be executed whenever a function is entered or exited in the instrumented application. This allows more complex information to be collected about the program state, such as the call stack of the application.

NEAT comes packaged with three predefined sets of floating-point rules that cover many use-cases and show off its versatility. The first set of rules uses the same floating-point implementation for every FLOP in a program. The second set of rules allows the user to specify a map of function names to floating-point implementations, and uses each floating-point implementation for the FLOPs in the corresponding function. The final set of rules also allows the user to specify a map of function names to floating-point implementations. It uses callbacks registered with NEAT to keep track of the call stack of the program. The floating-point implementation used for each FLOP in the program is the implementation corresponding to the function most recently put on the call stack. If no functions in the call stack match the names of those in the user-supplied map, a default implementation is used.

CHAPTER 4

CASE STUDY

We performed a case study to show the expresiveness and usability of NEAT. In this case study, we use NEAT to explore the design space of whole-program and per-function FP rules on a number of different benchmarks.

This case study shows that NEAT provides a helpful and versatile framework for exploring the design space of approximate FPU combinations. Without NEAT, exploring this design space would be slow and error-prone. But with NEAT, this process just involved supplying the right inputs to NEAT, and waiting for NEAT to collect the data with no supervision.

4.1 Benchmarks

We collected data from eight benchmarks. `blackscholes`, `bodytrack`, `ferret` [21], and `fluidanimate` are all taken from the PARSEC benchmark suite [4]; `heartwall`, `kmeans`, and `particlefilter` are all taken from the Rodinia benchmark suite; `radar` is a small radar application with a design similar to Figure 2.1. For each application, we acquire a set of representative inputs, then partition the smaller inputs into *training* sets, and the larger inputs into *test* sets. The training inputs are used to run the NSGA-II [8] genetic algorithm on the function-level FP rules. The test inputs are used to evaluate the benchmarks on the whole-program FP rules and on the Pareto-optimal points for the per-function FP rules obtained from the NSGA-II algorithm. Table 4.1 summarizes the sources of these inputs.

Data from the per-function FP rules was not collected for `blackscholes` or `heartwall`, since they each only have one function that performs floating-point arithmetic.

Table 4.1: Summary of Training and Test Inputs for Each Benchmark

Benchmark	Training Inputs	Test Inputs
`blackscholes`	16 options	4096 options
`bodytrack`	50-100 particles and 2-3 annealing layers	500-1000 particles and 3-5 annealing layers
`ferret`	database with 100 images	database with 3544 images
`fluidanimate`	15000 particles	35000 particles
`heartwall`	2 frames	3 frames
`kmeans`	32-128 points	256-1024 points
`particlefilter`	20-30 frames with 1000-2000 particles	50-100 frames with 10000-20000 particles
`radar`	128-512 input channels	1024-4096 input channels

4.2 Floating-Point Implementations

We used 24 different floating-point implementations to collect data from the benchmarks. Each floating-point implementation performs normal floating-point arithmetic, but zeroes out a different number of bits in the mantissa of the result of each operation. This allows us to have 24 floating-point implementation that each have a different level of accuracy.

4.3 Whole Program FP Rules

The first set of FP rules we used to collect data replaces every floating-point operation in the applications with one floating-point implementation.

4.4 Function-Level FP Rules

The second set of FP rules we used to collect data is slightly more complicated: they use a different floating-point implementation for 10 functions in each application. There are 24^{10} combinations of floating-point implementations we could use for this set of rules This is far too many combinations to explore manually, so we use the NSGA-II genetic algorithm to help us find Pareto-optimal points in this design space. We chose this algorithm because it has a better asymptotic runtime compared to similar algorithms [8]. More specifically, we

Table 4.2: Parameters for the NSGA-II Algorithm

Parameter	Value
Initial population size	24
μ	10
λ	15
Crossover probability	98%
Individual crossover probability	50%
Mutation probability	2%
Individual mutation probability	10%
Number of generations	25

used a $\mu + \lambda$ evolution strategy with uniform crossover, uniform mutation, and NSGA-II selection. The evolutionary parameters we supplied to the algorithm can be fund in Table 4.2.

We first used NEAT to find the 10 functions in each application that contain the most floating-point operations. We then used the NSGA-II algorithm to collect data from each application using the training data and a different floating-point implementation for each of these 10 functions. A default floating-point implementation was also chosen to use for every other function in each application. Once the algorithm had been run for a sufficeint number of generations, the points on the Pareto-optimal front were re-collected using the testing data.

We initialized the floating-point implementations for each of the 10 functions for each individual to the same value. This is done for each of the 24 initial individuals, which mimics the points collected by the whole-program FP rules. This represents a best-guess for the initial population based on our data collected from the whole-program FP rules.

CHAPTER 5

EXPERIMENTAL EVALUATION

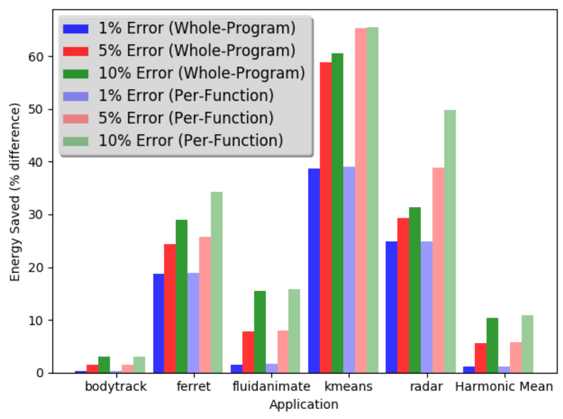
5.1 FP Rules Coverage

We can approximate the energy consumed by each floating-point arithmetic operation as the total number of bits manipulated by that operation. This includes the number of bits used in the operands and result of the operation. We included functionality in NEAT to measure this quantity.

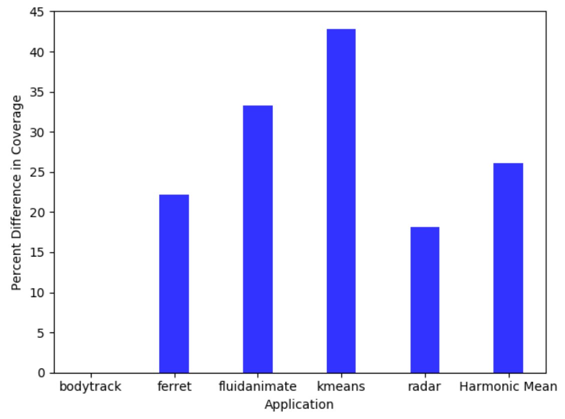
If we plot the total number of bits used in floating-point arithmetic operations in an application against the percent error of the output, we can visualize the power-accuracy design space of the application. We can plot points in this space that correspond to different FP rules and floating-point implementation combinations. The best points achieved by each set of FP rules are on the lower convex hull of the set of points we measuered for these sets of rules.

One measure of the amount of energy saved by the per-function FP rules over the whole-program FP rules is the difference in height of these convex hulls at different points. Figure 5.1a shows the difference in height between the whole-program FP rules and per-function FP rules using both methods of initialization for 1%, 5%, and 10% error.

We can also measure the difference between the whole-program approach and per-function approach by calculating the difference in coverage between the points on the convex hulls of the two approaches. This difference in coverage represents the points in the design space that can be obtained using the per-function FP rules but not the whole-program FP rules. Figure 5.1b shows the difference in coverage between the whole-program FP rules and per-function FP rules using both methods of initialization under 100% error.



(a)



(b)

Figure 5.1: **(5.1a)** The energy saved by each application at 1%, 5%, and 10% error **(5.1b)** The difference in coverage between whole-program and per-function FP rules for each application

5.2 Per Application Results

Figure 5.2 shows the convex hulls for each benchmark and set of FP rules. Data was collected for 3 inputs from the set of test inputs for each benchmark, and the number of bits used and percent error of the output were taken as the median of the 3 inputs. These graphs show the lower convex hulls of these points with up to 20% error.

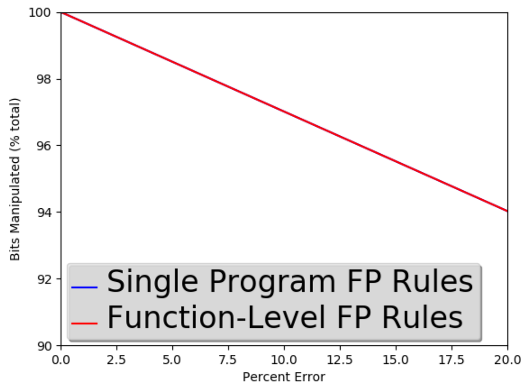
5.3 Training vs. Test Results

We want to show that the per-function FP rules perform better than the whole-program FP rules for reasonably sized-inputs. However, running NEAT on these inputs takes a considerable amount of time. It would be useful if we could gain insight about how different sets of FP rules would affect reasonably-sized inputs by collecting data from smaller inputs. To show that there is a correlation between these two sizes of data, we have collected the percent error from both sets of inputs for each benchmark using the whole-program and per-function FP rules. Data was collected for 3 inputs from each set of inputs, and the percent error of each point was taken as the median of these 3 inputs. We calculated the correlation

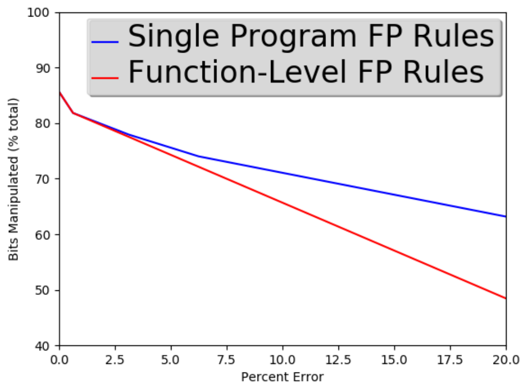
Table 5.1: Correlation Coefficients for Accuracy of Training and Test Inputs for Each Benchmark

Benchmark	Whole-Program	Per-Function
`blackscholes`	0.723777385384	0.723777385384
`bodytrack`	0.997932273171	1.0
`ferret`	0.970756520673	0.890890667442
`fluidanimate`	0.801022035884	0.907055750855
`heartwall`	0.999314953268	0.999314953268
`kmeans`	0.92734086092	0.979481949313
`particlefilter`	0.450245400438	0.399299721127
`radar`	0.997640234514	0.998818677211

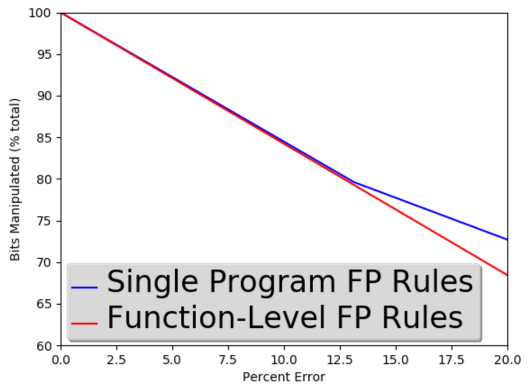
coefficients for each set of collected data up to 100% error, shown in Table 5.1. From these correlation coefficients, we can see that there is a high correlation between the percent error collected from each set of inputs.



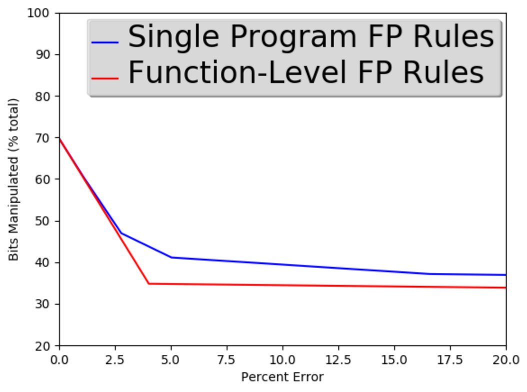
(a) Bodytrack



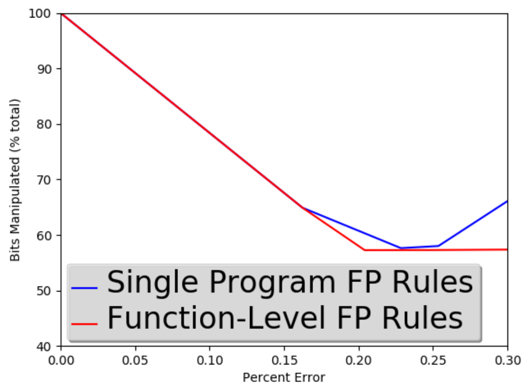
(b) Ferret



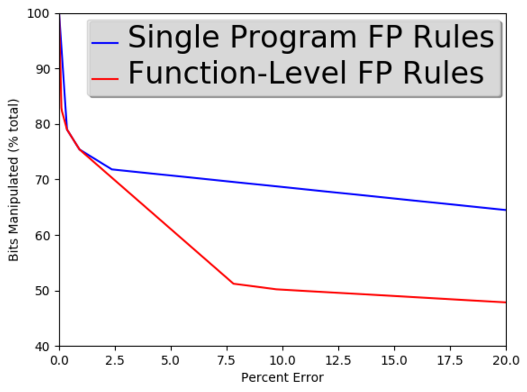
(c) Fluidanimate



(d) Kmeans



(e) Particlefilter



(f) Radar

Figure 5.2: Whole-Program vs. Per-Function FP Rules

CHAPTER 6
FUTURE WORK

One possible extension of this work is exploring different combinations of FP rules using NEAT. In our case study we only collected data using whole-program and per-function FP rules. However, there are many more combinations of FP rules that can be used in NEAT. Each set of FP rules creates its own FPU combination design space, which allows even more exploration of power-accuracy tradeoffs.

Optimizing the parameters for the NSGA-II algorithm when searching through FPU design spaces, or finding a more efficient heuristic for exploring these design spaces is another area for future work. We chose the NSGA-II algorithm for our case study because it is widely cited as an efficient algorithm, but did not tune the parameters for the algorithm because it was efficient enough with the initial parameters we chose. Tuning these parameters for different FPU combination design spaces would make the exploration of these design spaces even faster. Additionally, research could be done to determine if there is an even better heuristic to explore these design spaces.

Further, since heuristic techniques only provide best-guess exploration of these design spaces, better techniques could be researched to more confidently explore these design spaces. For example, active learning is an interactive strategy for finding optimal data points when provided with a large amount of unlabeled data. By using better techniques for exploring the design spaces, we can decrease the error bounds around each data point we collect and be more confident that we are finding optimal points in these design spaces.

Finally, a compiler tool could be created to work with NEAT and compile programs using the specified FP rules and FP arithmetic implementations supplied by a user. This would be useful in creating a final executable with the desired accuracy and power usage from the input program. The final executable would then be able to run directly on the hardware of a user's computer and not be subject to overhead from NEAT.

CHAPTER 7

CONCLUSION

In this work, we proposed NEAT, a tool for automated exploration of approximate FPU design. NEAT provides help to programmers trying to explore the design space of combinations of approximate FPU implementations. We performed a case study with NEAT to collect data from multiple benchmarks with whole-program and per-function FPU configurations. We found that the per-function configurations are able to explore more of the FPU combination design space than the whole-program configurations. We also found that data collected from smaller training inputs is highly correlated with data collected from reasonably-sized inputs.

REFERENCES

[1] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers*, 54(7):922–927, July 2005.

[2] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 529–540, Piscataway, NJ, USA, 2014. IEEE Press.

[3] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[5] Lakshmi N. Chakrapani, Bilge E. S. Akgul, Suresh Cheemalavagu, Pinar Korkmaz, Krishna V. Palem, and Balasubramanian Seshasayee. Ultra-efficient (embedded) soc architectures based on probabilistic cmos (pcmos) technology. In *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*, DATE '06, pages 1110–1115, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[6] A. P. Chandrakasan and R. W. Brodersen. Minimizing power consumption in digital cmos circuits. *Proceedings of the IEEE*, 83(4):498–523, Apr 1995.

[7] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 497–508, New York, NY, USA, 2010. ACM.

[8] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, Apr 2002.

[9] Peter Düben, Jeremy Schlachter, Parishkrati, Sreelatha Yenugula, John Augustine, Christian Enz, K. Palem, and T. N. Palmer. Opportunities for energy efficient computing: A study of inexact general purpose processors for high-performance and big-data applications. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 764–769, San Jose, CA, USA, 2015. EDA Consortium.

[10] Peter D. Düben, Jaume Joven, Avinash Lingamneni, Hugh McNamara, Giovanni De Micheli, Krishna V. Palem, and T. N. Palmer. On the use of inexact, pruned hardware in atmospheric modelling. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 372(2018), 2014.

[11] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 301–312, New York, NY, USA, 2012. ACM.

[12] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society.

[13] N. Gajjar, N. M. Devahsrayee, and K. S. Dasgupta. Scalable leon 3 based soc for multiple floating point operations. In *2011 Nirma University International Conference on Engineering*, pages 1–3, Dec 2011.

[14] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 383–397, New York, NY, USA, 2015. ACM.

[15] R. Hegde and N. R. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In *Proceedings. 1999 International Symposium on Low Power Electronics and Design (Cat. No.99TH8477)*, pages 30–35, Aug 1999.

[16] A. Kanduri, M. H. Haghbayan, A. M. Rahmani, P. Liljeberg, A. Jantsch, N. Dutt, and H. Tenhunen. Approximation knob: Power capping meets energy efficiency. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2016.

[17] Avinash Lingamneni, Christian Enz, Krishna Palem, and Christian Piguet. Synthesizing parsimonious inexact circuits through probabilistic design techniques. *ACM Trans. Embed. Comput. Syst.*, 12(2s):93:1–93:26, May 2013.

[18] Avinash Lingamneni, Kirthi Krishna Muntimadugu, Christian Enz, Richard M. Karp, Krishna V. Palem, and Christian Piguet. Algorithmic methodologies for ultra-efficient inexact architectures for sustaining technology scaling. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 3–12, New York, NY, USA, 2012. ACM.

[19] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages*

and Operating Systems, ASPLOS XVI, pages 213–224, New York, NY, USA, 2011. ACM.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[21] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Ferret: A toolkit for content-based similarity search of feature-rich data. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 317–330, New York, NY, USA, 2006. ACM.

[22] Divya Mahajan, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, and Hadi Esmaeilzadeh. Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 66–77, Piscataway, NJ, USA, 2016. IEEE Press.

[23] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 309–328, New York, NY, USA, 2014. ACM.

[24] T. Moreau, A. Sampson, and L. Ceze. Approximate computing: Making mobile systems more efficient. *IEEE Pervasive Computing*, 14(2):9–13, Apr 2015.

[25] Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L. Jones. Scalable stochastic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 335–338, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[26] Kumud Nepal, Yueting Li, R. Iris Bahar, and Sherief Reda. Abacus: A technique for automated behavioral synthesis of approximate computing circuits. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 361:1–361:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[27] Krishna V. Palem, Lakshmi N.B. Chakrapani, Zvi M. Kedem, Avinash Lingamneni, and Kirthi Krishna Muntimadugu. Sustaining moore's law in embedded computing through probabilistic and approximate design: Retrospects and prospects. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 1–10, New York, NY, USA, 2009. ACM.

[28] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan. Aslan: Synthesis of approximate sequential circuits. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014.

[29] J. Schlachter, V. Camus, K. V. Palem, and C. Enz. Design and applications of approximate circuits by gate-level pruning. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, PP(99):1–9, 2017.

[30] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose code acceleration with limited-precision analog computation. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 505–516, Piscataway, NJ, USA, 2014. IEEE Press.

[31] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 1–12, New York, NY, USA, 2013. ACM.

[32] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan. Axilog: Language support for approximate hardware design. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 812–817, March 2015.

[33] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. Neural acceleration for gpu throughput processors. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 482–493, New York, NY, USA, 2015. ACM.