NEAT: A Tool for Automated Exploration of Approximate FPU Designs

Lee Ehudin
University of Chicago
ehudinl@uchicago.edu

Saeid Barati
University of Chicago
saeid@cs.uchicago.edu

Henry Hoffmann
University of Chicago
hankhoffmann@cs.uchicago.edu

Abstract

Much recent research is devoted to exploring tradeoffs between computational accuracy and energy efficiency at different levels of system stack. Approximation at the floating point unit (FPU) allows saving energy by simply reducing the number of computed floating point bits in return for accuracy loss. Although, finding the most efficient floating point implementation for various applications with minimal effort is the main challenge.

To address this issue, we propose NEAT: a pin tool that helps users automatically explore the accuracy-energy tradeoff space induced by various floating point implementations. NEAT helps programmers explore the effects of approximations induced by using multiple floating point implementations to achieve the lowest energy consumption for an accuracy constraint or vice versa. NEAT accepts one or more user-defined floating point implementations and rules for when to apply them. NEAT then replaces floating point operations with different implementations based on user-specified rules during the runtime.

We evaluate NEAT by enforcing 24 different floating point implementations with two sets of rules on two benchmarks. We find that assigning different floating point implementations per function provides more efficiency than using a single implementation for the whole application.

1 Introduction

Early work in approximate computing demonstrates the tremendous energy and execution time reductions by making a variety of functional units available [3, 4, 7, 8]. 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 and navigate through this immense tradeoff space enacted by allowing multiple approximations within a single program? Just considering moderate-sized programs with 10 functions and 20 different levels of approximation (aka floating point implementation), we already have an intractably large design space with 20^{10} points to explore.

This 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. NEAT then runs the program and dynamically replaces floating point operations (FLOPs) with the approximate version as specified by the rules. NEAT outputs the program's output and the number of bits used. Thus, NEAT can be used as a tool to explore the tradeoff space of FPU design without requiring deep numerical expertise.

We implement NEAT for x86 using the Pin binary instrumentation system [13]. We demonstrate NEAT's value by comparing the approximations produced by two different rule sets. In the first, we simply pick one FP implementation for the entire program; *i.e.* the rule is a simple one-to-one replacement (whole-Program rule). In the second, we allow the top 10 executed functions with the most FLOPs to each use a different FP implementation (per-Function rule). Next, we employ a genetic algorithm to guide NEAT's exploration of the enormous resulting search space.Our results show that the per-Function configurations uses less energy at the same error rate than the whole-Program configurations.

In summary, this paper proposes:

- The NEAT framework that helps users explore the design space of FPU combinations.
- A case study that compares whole-Program vs. per-Function FPU configurations for two benchmarks.

2 Background & Motivation

While there has been a substantial amount of effort are aimed towards finding new forms of approximation [1, 5, 9, 12, 14, 16], there is a lack of solutions that helps the user to specify their own approximations for a single application. Approximation Knobs provide a way to lend performance and energy gains to existing power knobs[11] . Quora is a quality programmable processor where the notion of quality is codified in the instruction set of the processor [15]. Another example of user-defined approximation is Green, which is a system that allows programmers to supply approximate versions of loops and while-blocks that terminate early [2].

While these techniques provide energy and runtime savings, but they do not help users make more informed decisions about how to approximate and they are not flexible about how much to approximate.

PL'17, January 01–03, 2017, New York, NY, USA
2017.

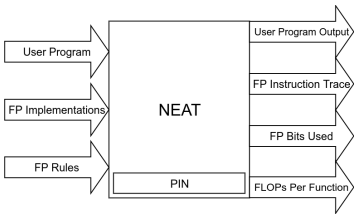
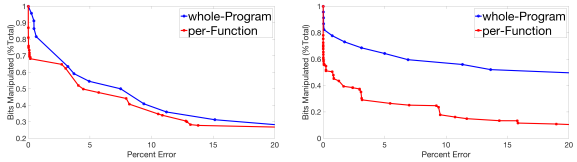


Figure 1. NEAT Design



(a) blackscholes **(b)** ferret

Figure 2. Whole-program vs. per-Function FP rules

Recent proposals advocate putting many different approximate FPUs on a single core [10] and have shown using a different FPU for each function provides more energy efficiency [7] but this requires tedious hand-tuning. Therefore the challenge is how to figure out which FPU to use in each part of the program.

An FPU approximation can be done by masking the mantissa bits in the IEEE floating point representation. Since there are 23 mantissa bits in a single precision floating point number, we can have 24 possible FPUs for this type. Even if we consider a single FPU per function, for a moderate-sized application with 10 functions, the FPU design space becomes untractable. NEAT provides such a solution to find the most efficient FP implementation for the whole program or per-function.

3 System Design

The main purpose of NEAT is to allow users to replace FLOPs with approximate implementation in the applications. Users can specify multiple floating point implementations (by modifying operands or individual arithmetic operations) to assign a single FP implementation to the whole program or an individual function. NEAT then calculates each FLOP in a program using the FP implementation dictated by the rules.

Figure 1 illustrates the NEAT structure. User inputs of NEAT include: a user application to instrument, the desired FP arithmetic implementations, and a set of FP rules to choose which FP implementation to use for each operation. Sets of rules are specified as C++ functions that accept the program state as input and return a single FP implementation as output. Whenever a FLOP should be computed, NEAT captures information about the current state of the application and uses the rules to calculate the result of that operation.

NEAT comes packaged with two predefined sets of FP rules. The first set of rules uses the same FP implementation for every FLOP in a program. The second set of FP rules allows the user to specify a map of function names to FP implementations and uses each of them for the FLOPs in the corresponding function.

There are four outputs from NEAT: 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.

NEAT calculates the number of bits used in the binary representation of the floating point number, starting with the least significant bit. This metric can be used as a platform-independent way to evaluate the approximate amount of power used by FLOPs when instrumenting a program.

4 Experimental Evaluation

We evaluate NEAT by exploring the design space of whole-program and per-function FP rules on `blackscholes` and `ferret` benchmarks. We used 24 different FP implementations by zeroing out a different number of bits in the mantissa. NEAT considers top 10 functions with the most FLOPs to enforce the FP rules in the per-Function approach. Since tradeoff space of per-Function approach is huge (24^{10} different combinations), we use the NSGA-II genetic algorithm to help us find Pareto-optimal points in this design space[6].

Figure 2 shows the lower convex hulls for each benchmark and set of FP rules. The horizontal axis shows percent error caused by approximation while the vertical axis represents the floating point bits executed in the program normalized to default FP implementation. We can estimate the energy consumed by each floating point arithmetic operation as the total number of bits manipulated by that operation. The curves that are closer to the origin are considered more efficient. For the same precent error, the per-Function rule uses less FP bits than whole-Program approach. For example, at the 5 and 10 % error for `ferret`, whole-Program rule uses 54.4% and 40.9% of total FP bits respectively while the per-Function approach executes only 29.4% and 18.3%. These graphs indicate that specifying the FP rules at a finer granularity results in locating more efficient combinations of FP implementations. In other words, per-Function FP rules use less energy with the same error comparing to use a single FP implementation for the whole application.

5 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 two benchmarks with whole-program and per-function FPU configurations. We found that the per-function approach provides more efficient combinations of FPU design comparing to single FPU for the whole program.

References

[1] C. Alvarez, J. Corbal, and M. Valero. 2005. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.* 54, 7 (July 2005), 922–927. https://doi.org/10.1109/TC.2005.119

[2] Woongki Baek and Trishul M. Chilimbi. 2010. 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)*. ACM, New York, NY, USA, 198–209. https://doi.org/10.1145/1806596.1806620

[3] Lakshmi N. Chakrapani, Bilge E. S. Akgul, Suresh Cheemalavagu, Pinar Korkmaz, Krishna V. Palem, and Balasubramanian Seshasayee. 2006. 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)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 1110–1115. http://dl.acm.org.proxy.uchicago.edu/citation.cfm?id=1131481.1131790

[4] A. P. Chandrakasan and R. W. Brodersen. 1995. Minimizing power consumption in digital CMOS circuits. *Proc. IEEE* 83, 4 (Apr 1995), 498–523. https://doi.org/10.1109/5.371964

[5] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 497–508. https://doi.org/10.1145/1815961.1816026

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (Apr 2002), 182–197. https://doi.org/10.1109/4235.996017

[7] Peter D. Düben, Jaume Joven, Avinash Lingamneni, Hugh McNamara, Giovanni De Micheli, Krishna V. Palem, and T. N. Palmer. 2014. 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). https://doi.org/10.1098/rsta.2013.0276 arXiv:http://rsta.royalsocietypublishing.org/content/372/2018/20130276.full.pdf

[8] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 301–312. https://doi.org/10.1145/2150976.2151008

[9] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 449–460. https://doi.org/10.1109/MICRO.2012.48

[10] N. Gajjar, N. M. Devahsrayee, and K. S. Dasgupta. 2011. Scalable LEON 3 based SoC for multiple floating point operations. In *2011 Nirma University International Conference on Engineering*. 1–3. https://doi.org/10.1109/NUiConE.2011.6153274

[11] A. Kanduri, M. H. Haghbayan, A. M. Rahmani, P. Liljeberg, A. Jantsch, N. Dutt, and H. Tenhunen. 2016. Approximation knob: Power Capping meets energy efficiency. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/2966986.2967002

[12] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. 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)*. ACM, New York, NY, USA, 213–224. https://doi.org/10.1145/1950365.1950391

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. 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)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[14] Krishna V. Palem, Lakshmi N.B. Chakrapani, Zvi M. Kedem, Avinash Lingamneni, and Kirthi Krishna Muntimadugu. 2009. 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)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/1629395.1629397

[15] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Quality Programmable Vector Processors for Approximate Computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2540708.2540710

[16] 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. 2015. Axilog: Language support for approximate hardware design. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 812–817. https://doi.org/10.7873/DATE.2015.0513