Energy-Efficient Computing Using Adaptive Table Lookup Based on Nonvolatile Memories

Jason Cong*, Milos Ercegovac*, Muhuan Huang*, Sen Li*, Bingjun Xiao†
*Computer Science Department and †Electrical Engineering Department
University of California, Los Angeles
Los Angeles, California, 90095
{cong, milos, mhhuang, senli, xiao}@cs.ucla.edu

***Abstract*—Table lookup based function computation can significantly save energy consumption. However existing table lookup methods are mostly used in ASIC designs for some fixed functions. The goal of this paper is to enable table lookup computation in general-purpose processors, which requires adaptive lookup tables for different applications. We provide a complete design flow to support this requirement. We propose a novel approach to build the reconfigurable lookup tables based on emerging nonvolatile memories (NVMs), which takes full advantages of NVMs over conventional SRAMs and avoids the limitation of NVMs. We provide compiler support to optimize table resource allocation among functions within a program. We also develop a runtime table manager that can learn from history and improve its arbitration of the limited on-chip table resources among programs.**

I. Introduction

Application-specific accelerators can fully exploit parallelism and customization while providing orders-of-magnitude improvement in power-efficiency over CPUs [1]. The extreme example of an accelerator is the table lookup which allows computing functions with very low energy consumption. For example, the computation of the commonly-used function $\sin(x)$ involves many hundreds of instructions. Our experiments using the Simics [2] and Gems [3] simulator on the Ultra-SPARC-III processor shows 226 clock cycles in a single execution of $\sin(x)$ with an energy consumption of 104nJ — 450x the energy of an instruction of integer add. If all possible results of $\sin(x)$ are pre-stored in a large table, then a single table lookup is sufficient for each function call of $\sin(x)$, and the computation of $\sin(x)$ becomes very energy-efficient. Table lookup methods are representative cases that trade space for time and energy savings. They have been the subject of research for many years [4–7]. Since the straightforward table lookup implementation needs a very large table (>16GB for a 32-bit operand), bipartite methods have been proposed to reduce the table size [4]. This method uses two smaller tables, but needs one extra operation to add the two lookup results. Multipartite methods were further proposed to allow more design choices for the number and size of lookup tables, and thus more flexibility in optimizing total table size and number of additions [6]. These works focus on the optimal table lookup design for a specific function, e.g., $\sin(x), \log(x), \exp(x)$, etc.

Existing table lookup methods are applied to ASICs only. However the rapid increase of non-recurring engineering cost and design cycle of ASICs in nanometer technologies makes it impractical to implement most applications in ASICs. The goal of this paper is to popularize table lookup methods in general-purpose processors to achieve significant energy savings. Since general-purpose processors execute numerous user applications (see examples in Table I), we cannot afford to have a fixed lookup table for each function. Instead, we need to adapt table content to users' applications.

There are two main challenges to realizing adaptive table lookup. First, in order to reconfigure lookup tables according to users' demands, tables can no longer be implemented in read-only memories or silicon anti-fuse with near-zero area overhead. Tables will be implemented in writable memories which have much lower storage density and may consume leakage power. For example, the on-chip memory, cache, is usually built from SRAMs which need at least six transistors to store one bit. A 32MB cache in an Intel®Itanium processor [8] occupies about 50% of the total chip area. In addition, SRAMs need a power supply to keep the stored value during standby, which consumes about 35% of the total chip power [8]. The second challenge is that once lookup tables can be composed to cover different functions in different programs, these functions and programs will compete for the limited table resource. Functions and programs have different impacts on the overall processor energy efficiency. Given the total reconfigurable table size available on a chip, an interesting research problem is to explore smart table allocations among functions and programs to achieve the maximum energy savings of the system.

To enable table lookup in general-purpose processor, we will solve all the challenges discussed in Section I. The contributions of this paper are as follows:

- We provide a novel approach to build the reconfigurable lookup tables based on emerging nonvolatile memories (NVMs), which takes full advantages of NVMs over conventional SRAMs and avoids the limitation of NVMs.
- We provide compiler support for adaptive lookup tables so that within each compiled program and with any given table size allocated to the program, we optimize the distribution of the table resource among table-accelerated functions.
- We develop a runtime table manager which can learn from history and improve its arbitration of the limited on-chip table resources among programs.

II. NVM-Based Table Lookup

A. Emerging Nonvolatile Memory

With the advancement of material and device technology, emerging nonvolatile memories (NVMs) are gaining maturitive. NVMs include spin-transfer torque RAM (STTRAM), phase-change RAM (PCRAM), and resistive RAM (RRAM). While the fabrication choice of RRAM is still in hot debate [9], various semiconductor companies (e.g., Samsung and Hynix) are manufacturing STTRAM and PCRAM in mass volume. All the emerging NVMs have CMOS-compatible fabrication process, and show a significantly higher storage density than conventional SRAMs and competitive read performance, as shown in Table II. The nonvolatility of NVMs also greatly reduce the leakage power during standby. The main disadvantage of NVMs is the poor performance of write operations concerning latency, energy, and endurance. There are numerous papers that explore methods for reducing the number of writes to NVMs which are used as cache or main memory [13–16]. From our perspective, since the asymmetry of NVMs' between read performance and write performance is more than three orders of magnitude (as shown in Table II), it is difficult to apply NVMs to cache or main memory where read frequency and write frequency are at the same scale. The task of finding a suitable application for NVMs that matches their asymmetry will benefit greatly from

Table I: Computation-intensive functions in different applications where table lookups have potential energy savings.

Application	Computation
Geometry	$\sin(x), \tan(x), \arcsin(x)$, ...
Statistics	Error function $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$, chi-squared function, ...
Wave propagation	Bessel function $J_\alpha(x) = \frac{1}{2\pi} \int_0^{2\pi} cos(\alpha\tau - xsin\tau) d\tau$, ...
Field potential calculation	Legendre polynomial $P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} [(x^2-1)^n]$, ...
Analytical chemistry	Gaussian equation for Spectra $A_A = \frac{C_A \epsilon_A}{\sigma\sqrt{2\pi}} e^{(\lambda - \lambda maxA)^2/(2p^2)}$, ...
Financial	Black-Scholes $\phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x e^{-t^2/2} dt$, ...
Medical imaging	Riciandenoise3D $f(r) = \frac{r(2.38944+r(0.950037+r))}{4.65314+r(2.57541+r(1.48937+r))}$, ...

Table II: Comparison of SRAM, STTRAM [10] and PCRAM [11] at 32nm technology node and 512KB by NVSIM [12]. F: feature size.

technology	cell size	read latency	read energy	write latency	write energy	write cycles	leakage power
SRAM	$140F^2$	1.90ns	103pJ	1.449ns	0.102nJ	3×10^{16}	0.16mW
STTRAM	$20F^2$	1.73ns	91.3pJ	101ns	19.5nJ	1×10^{12}	~0
PCRAM	$4F^2$	1.43ns	92.4pJ	301ns	10.3nJ	1×10^9	~0

the progress of this advanced technology. NVM-based FPGAs [17–19] are a good example since there are write accesses only during FPGA programming and the number of programming cycles is expected to be small (<500) for typical FPGA users. We believe that reconfigurable lookup tables for function acceleration will be another promising application since the tables are frequently read (upon each lookup) but are much less often updated (only upon launching a new program).[1]

B. Direct Impact of NVMs on a Single Lookup Table

Compared to SRAMs, the direct impact of NVMs on a single lookup table is that energy consumption is reduced for each table size, as shown in Fig. 1. For each table size in the figure, we

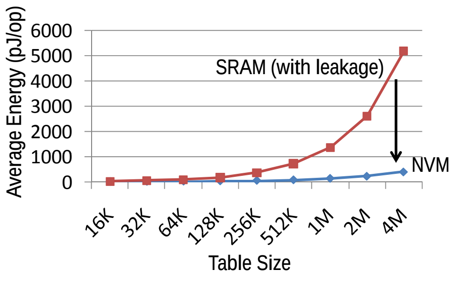


Figure 1: The direct impact of NVMs on a single lookup table.

compute the energy consumption per read operation for SRAMs and NVMs using the memory modeling tool NVSIM [12]. The leakage power of SRAMs is converted into the energy consumption per read operation based on the average ratio of total runtime over the total number of read accesses among a set of benchmarks. We can clearly observe that NVMs can save huge energy consumption compared to SRAMs, especially when the table size is large.

C. Scheme and Tradeoff of Table Lookup Design

Now we want to optimize the table lookup design of the computation of a function. The basic design tradeoff is that larger table size leads to fewer arithmetic operations. The extreme case is that if we want to implement a function with a 32-bit ($w_l = 32$) operand into a single table lookup without any post-processing, a 16GB table will be needed to store all possible results. It is not possible to allocate such a large table to a single function. Bipartite and multipartite methods are proposed to enable tradeoff between table size and arithmetic operations [4, 6]. The design scheme of the most general multipartite method is shown in Fig. 2. The 32-bit operand

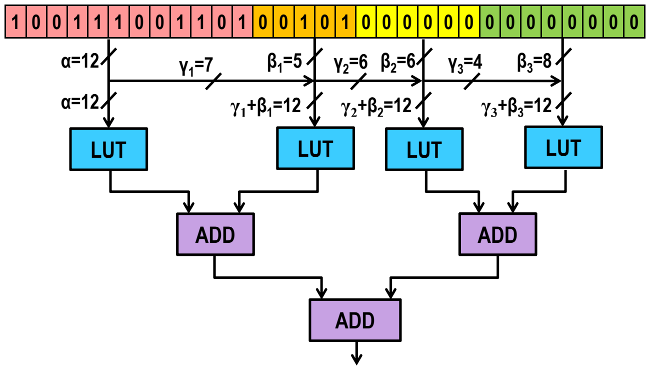


Figure 2: Execution flow of the computation by the multipartite method [6]. This method keeps the full precision of the output allowed by the 32-bit data width and offers tradeoff between lookup tables and arithmetic additions.

is decomposed into $m = 4$ partitions in this example. The most significant $\alpha = 12$ bits (let's say A) will index 2^{12} segments out of the 2^{32} possible inputs. For each segment, an initial value is looked up by A and the other values are interpolated by adding, to this initial value, offsets computed out of the $32 - 12 = 20$ least significant bits of the input word, as shown in Fig. 3. The idea behind the multipartite method is further decompose the 20 least significant bits into β_i-bit fragments ($i = 1, 2, ..., m-1$). For each β_i-bit fragment, the method groups the 2^α input intervals into 2^{γ_i} larger intervals such that the slope of the segments can be considered constant at cost of accuracy loss. As long as the accuracy loss is smaller than the precision of 32-bit digit representation, the result will remain correct. Since the least significant $\beta_i, \beta_2, \beta_3$ bits B_1, B_2, B_3 have decreasing impact on the final results, the precision of their corresponding slopes to multiply can be relaxed to fewer

[1]FPGAs also contain an array of look-up tables but with much smaller size (16 bits to 64 bits per table). The programmability of FPGAs mainly comes from their reconfigurable interconnects.

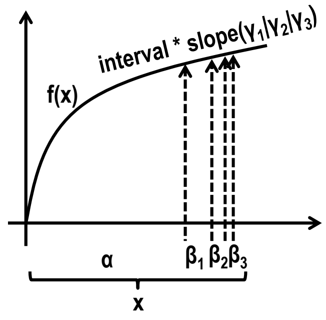


Figure 3: Illustration of how the final result is computed by the multipartite method [6].

bits $\gamma_1, \gamma_2, \gamma_3$, as shown in Fig. 2. The multipartite fomula is written as

$$\begin{aligned} f(x) =& F_0(A) + F_1(A_1, B_1) + F_2(A_2, B_2) \\ &+ \cdots + F_{m-1}(A_{m-1}, B_{m-1}) \end{aligned}$$

In this way, we only need to store 2^{α} initial values plus $2^{\beta_1+\gamma_1} + 2^{\beta_2+\gamma_2} + \cdots + 2^{\beta_{m-1}+\gamma_{m-1}}$ offsets, i.e., 2^{14} values in all for this case, much less than the original 2^{32} values.[2] There are flexible design choices on the number of tables and the exploration of values $(\alpha, \beta_i, \gamma_i)$ that lead to different table sizes and addition cost.

D. Impact of NVM on Table Lookup Design

When a designer decides the number of lookup tables m, the multipartite method [6] will give the optimal setting of $(\alpha, \beta_i, \gamma_i)$. For each design choice concerning how many tables to implement, we can calculate the corresponding energy consumption of all the lookup tables implemented in SRAMs and NVMs. When added to the energy consumption of ALUs (scaled with m), we arrive at the total energy consumption of one lookup-based execution of the function as shown in Fig. 4. We observe that while SRAM-

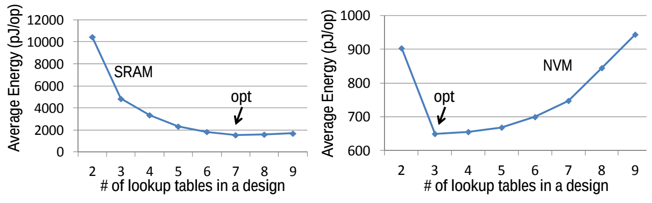


Figure 4: For each design choice on # of tables, Total energy consumption (table lookups plus adders).

based table lookup prefers more tables with smaller table sizes, NVM-based table lookup prefers fewer tables with larger table sizes, because NVM-based implementation does not need to pay leakage overhead for larger tables.

E. Design Curve for NVM-Based Table Lookup

Note that although we get the optimal energy consumption for any design choice in the number of lookup tables, the number of tables is just a tunable design parameter. What constrains a table lookup design is the table size that is allocated for the design. Since the available table size cannot be adjusted by programmers and might change according to runtime system resource utilization, what we can do is always try to get the maximum energy savings under an arbitrary table size constraint. To realize this goal, we also calculate the total size of lookup tables for each design choice that concerns the number of tables, as shown in Fig. 5. Then we combine it with

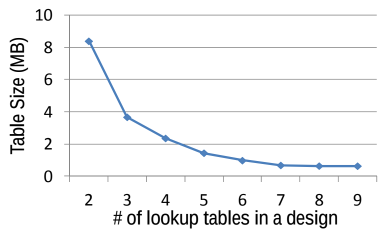


Figure 5: For each design choice on # of tables, minimum table size to be allocated. The table size graduality is 128 bytes under the assumpiton of 32x32 memory subarrays to fit the default NVsim settings.

Fig. 4 and get the design curve of the NVM-based table lookup, as shown in Fig. 6. This curve will be provided to tools at higher

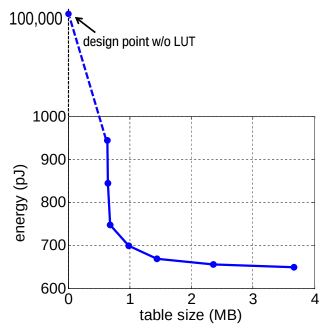


Figure 6: Design curve of NVM-based table lookup. Design point with table size = 0 means using a conventional processor without help of any lookup table.

levels to determine the size of the table that will be allocated to this function. Note that the design point at the number of tables equal to two is removed in the curve since it is not a pareto-optimal point. The design point at three tables is strictly better concerning both table size and energy consumption. Also we added the design point at table size equal to zero because it corresponds to the case in which computation is performed in the conventional way without table lookup.

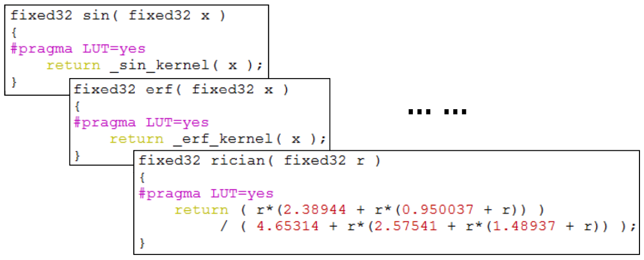
III. Compiler Support

We allow programmers to specify in their codes whether an attempt should be made to implement a function in lookup tables or not, as shown in Fig. 7.[3] Within a program, multiple functions may be marked for table lookups. For the example shown in Fig. 4, the optimal table lookup design needs 3MB size. If we increase processor size by 0.5% for NVM-based lookup tables, there will be only 10MB in total (assume $4F^2$ cell size in Table II).[4] Multiple programs running in the processor will compete for the limited table resource. Multiple functions within a program will compete for the table resource as well. Since the impact of each function on the total energy consumption of the program can be analyzed during compile time, we first enhance the compiler to optimize the allocation of

[2]The choice of $(\alpha, \beta_i, \gamma_i)$ in this example is not generated by the multipartite method but set for illustration only.

[3]Our flow does not support floating point computation yet.

[4]We use this setting in our first attempt of adaptive table lookup. After wide acceptance of this technology, it will be interesting to do design space exploration on this setting.



```
fixed32 sin( fixed32 x )
{
#pragma LUT=yes
    return _sin_kernel( x );
}
```

```
fixed32 erf( fixed32 x )
{
#pragma LUT=yes
    return _erf_kernel( x );
}
```

```
fixed32 rician( fixed32 r )
{
#pragma LUT=yes
    return ( r*(2.38944 + r*(0.950037 + r)) )
         / ( 4.65314 + r*(2.57541 + r*(1.48937 + r)) );
}
```

Figure 7: Pragma to allow programmers to specify in their codes whether to attempt to implement a function in lookup tables or not.

table resource among functions within a program. In this section, we describe our compiler support to perform such an optimization. The arbitration policy for table allocation among programs will be discussed in Section IV.

A. Problem Formulation

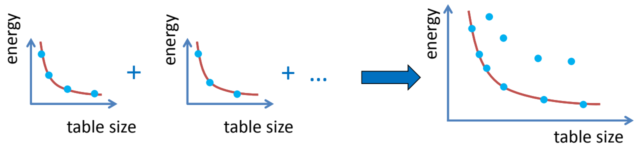


Figure 8: Composition of program pareto curve from function pareto curves.

As shown in Section. II, for each function that may be implemented using lookup tables, we will generate a pareto curve that trades off the energy and table size. We refer to such a pareto curve as a *function pareto curve*. A program can have several function pareto curves. By composing function pareto curves, we can arrive at a pareto curve for the program, which is referred as the *program pareto curve*. Figure. 8 shows the relationship between the function pareto curve and the program pareto curve. Such program pareto curve will be provided to the runtime table manager. During runtime given the current available table size in the system, the table manager can derive the minimum energy consumption from the program pareto curve. The pareto curve is not continuous in real applications. Instead, we are given a set of discrete points on the pareto curve. We refer to this set of discrete points as a pareto set. Thus our problem can be formulated as the following: given a pareto set (that trades off energy and area) for each function which can be implemented by table lookup techniques, derive a pareto set for the whole program.

B. Dynamic Programming Algorithm for Program Pareto Curve Generation

We propose an algorithm to generate the program pareto curve based on dynamic programming. Suppose the number of functions which can be implemented using lookup tables is n. The pareto set of the ith function is F^i. F^i_j is the jth pareto point of function i. The pareto set of the program is P, and P^i is the ith pareto point of the program. Each P^i consists of 1) the value of energy consumption E^i for the ith pareto point, and 2) an array T^i of size n where T^i_j represents the table size allocated for function j in pareto point P^i. For easy demonstration, we assume that the sizes of the pareto sets of the functions and the program are all m which equals to the total available table size, whereas in reality the size of the pareto set of each function and the program could be different, and our algorithm can be easily adapted to such scenario. The goal now is to compute E^i, $i \in \{1, 2, \ldots, m\}$.

Basically we compute the value of E^i as follows:

$$E^i = min_{j=0}^{i-1}(E^j + \Phi(i-j, j))$$

$$\Phi(t, i) = min_{j=1}^{n}(F^j_{T^i_j+t} - F^j_{T^i_j}).$$

The function $\Phi(t, i)$ computes the maximum energy savings if we allocate t table size to one of the functions based on the table allocation of the ith pareto point of the program. We first compute E^0 which represents the energy consumption of the program when no table is allocated to any of the functions. In order to compute E^i, for each of the computed $E^j, j \in \{0, 1, 2, \ldots, i-1\}$, we compute the sum of E^j and $\Phi(i-j, j)$, and select the minimum sum to be the value of E^i. The time complexity of our algorithm is $O(nm^2)$. The space complexity is $O(nm)$.

Overall our compiler support for table resource allocation contains the following two steps:

1) Profiling to capture the occurrence of the functions that can be implemented by table lookup. Within a program, different functions are called at different times. We should accelerate the functions that are called more frequently by allocating more resource to them. Therefore, we collect the number of function calls by profiling, and then scale up the energy for the function pareto curve.
2) Given the function pareto curves, we generate the program pareto curve using the dynamic programming algorithm discussed above.

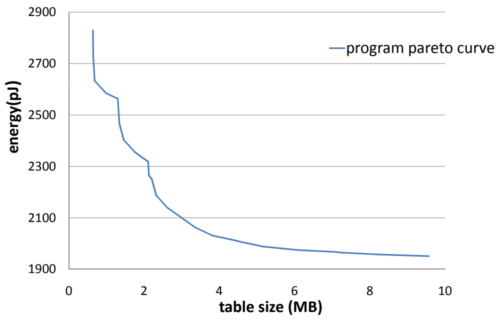


Figure 9: Generated program pareto curve.

C. Experimental Results for Compiler Support and Discussions

In Fig. 9, we show the program pareto curve generated in our experiment. The program in our experiment contains three function pareto curves, each has eight pareto points on its curve. The combined program pareto curve of a program is fed into the runtime table manager. During runtime, the table manager maximizes the overall energy efficiency by allocating a certain amount of table size to each incoming program using the method discussed in the next section.

Since a program will be among different pareto points under different table size allocations, the contents of tables used by this program will be changed dynamically. Dynamic creation of table contents is not a good solution since it involves lots of computation during program loading. Instead, we pre-calculate the table contents of each pareto point of each function in a program during compile time and store them in separate binaries. The table contents of some typical functions (e.g., $\sin(x)$) can even be provided by public maths libraries on the internet. After a program is allocated with a table size during the runtime (i.e., which program pareto point for the program to run on is decided), the corresponding binaries will be linked the function calls in the main program.

IV. Runtime Table Manager

In this section we discuss how we allocate table resources to each incoming application in order to maximize the overall energy efficiency. We first describe the problem formulation, and present an efficient machine-learning-based online table management approach.

A. *Problem Formulation*

During runtime, each job/application arrives at the system with the pareto curve generated in the previous section representing its table size/energy consumption tradeoff. We define a step to be the time between two successively arrived jobs. The goal for the runtime table manager here is to continuously decide and allocate the size of table resource to each incoming job so that the average system energy consumption per step is minimized.

One simple and intuitive approach is to use the greedy strategy where the runtime table manager allocates the amount of table size corresponding to the minimum energy consumption for each job. While it is a reasonable heuristic in such a resource-constrained environment, it makes decisions based only on the current system status and incoming application — regardless of the past and future job executions, and consequently may result in a less optimal solution.

This paper addresses the shortcomings of the greedy algorithm by formulating the problem as a Markov Decision Process (MDP) which can by solved by the Reinforcement Learning (RL) methodology [20] [21]. We first describe the MDP formulation below and then introduce our RL-based runtime table management algorithm in the next subsection.

An MDP for a single agent (in our case the runtime table manager) can be described by a quadruple (S, A, r, T) [20] where

- S: a finite set of states
- A: a finite set of actions
- r: a reward function $S \times A \times S \rightarrow R$
- T: a state transition function $S \times A \rightarrow PD(S)$ mapping the current state and action of the agent into the probability distributions of S.

At each step t, the agent observes the system's current state $s_t \in S$ and selects an action $a_t \in A$ accordingly. The system state then changes from s_t to s_{t+1} according to T. The agent then receives a reward $r(s_t, a_t, s_{t+1})$. The goal for the agent is to find a stationary policy $\pi : S \rightarrow A$ that maximizes the average reward per step starting from any initial state. This is mathematically defined as:

$$\rho^{\pi} = \lim_{T\rightarrow\infty} \frac{1}{T} \sum_{t=0}^{T-1} r(s_t, \pi(s_t), s_{t+1}).$$

B. *Algorithm for Runtime Table Management*

The key components of the management approach are 1) predicting the expected future energy efficiency per step, and 2) updating the prediction based on the action taken in the current step.

For the first problem, the most common approach is to design some learning function approximation architecture with tunable parameters (such as an artificial neural network) which can be adjusted to improve the quality of approximation. Regarding the second problem, a reinforcement learning (RL) process can be used to tune the parameters in the learning functions. As a demonstration, we adopt the approach in [21] to our problem using a fuzzy rulebase to approximate learning functions and RL process to update the fuzzy rulebase.

The overall runtime table management algorithm of allocating table to each incoming job is of the following steps:

1) Compute the input vector of the fuzzy rulebase x^i for each of the pareto point i in the pareto curve.
2) Compute V_i representing the expected energy consumption per step using the computed vector x^i and the fuzzy rulebase.
3) Select the smallest V_i and allocate the corresponding table size to the job.
4) Update parameters in the fuzzy rulebase using the RL process which is an iterative process and guarantees the convergence if certain conditions are satisfied [21].

C. *Experiments and Results of Runtime Table Management*

We implemented the runtime table management method by building a software runtime layer and instrumenting the management codes using LLVM [22]. The benchmarks we used in our experiment are listed in Table III. Jobs arrived continuously at different time steps — each having its own pareto curve. The runtime table manager allocates the table size for each of the incoming job based on the algorithm described in the previous subsection.

We compared our algorithm with 1) a simple greedy algorithm in which each incoming job receives the table size corresponding to the minimum energy consumption, and 2) a golden table management algorithm that has complete knowledge of the information of applications and provides an upperbound of the potential energy savings. The result is shown in Fig. 10.

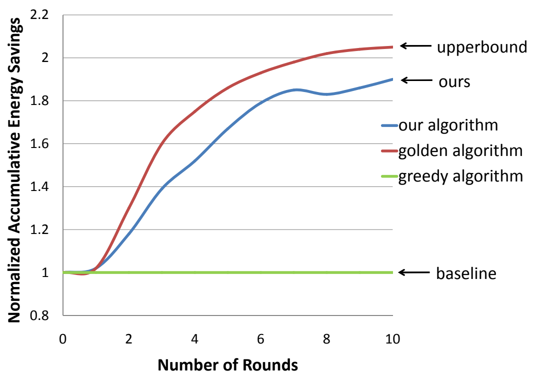


Figure 10: Energy savings of our approach compared with greedy algorithm.

The energy savings of our algorithm and the golden algorithm are normalized to the greedy algorithm. The x-axis shows the number of rounds of executions. During each round, each of the six benchmarks arrives at the system at a certain probability successively. The energy saving at each round is the cumulative energy saving from the beginning of the first round. From the results, we can make the following observations:

1) In the beginning of the execution there is little difference among the three approaches because the available table size is enough to be allocated to the incoming jobs and the greedy algorithm itself is optimal.
2) In the following several rounds the available table size begins to saturate. In this case both the golden algorithm and our algorithm perform better than the greedy algorithm since the golden algorithm performs global optimization with full information of arrived jobs, and our algorithm allocates table size by learning from the previous execution patterns of jobs. However since the number of rounds is not enough to fully train the fuzzy rulebase, the gap of energy savings between our algorithm and the golden algorithm is still relatively large.
3) In the later rounds, since the fuzzy rulebase is trained by a sufficient number of rounds of executions, the energy saving of our algorithm begins to converge to a fixed number (around

Table III: Benchmarks for runtime table management.

Benchmark	Description
SimSpectra	Simulation of Spectra with Gaussian equation
Blackschole	Pricing a portfolio of options with the Black-Scholes equation
Fluidanimate	Simulation of fluid motion for realtiime animation purposes with SPH algorithm
Riciandenoise3D	Denoise of 3D medical images with rician noise
Acousticradiation	Solving for acoustical radiation with Bessel functions
ERF	Computation of error function

$1.9x$), and the gap between our algorithm and the golden algorithm decreases.

V. Conclusion and Future Work

This paper provides the complete flow for adaptive table lookup in general-purpose processors as shown in Fig. 11. As the first attempt to enhance table lookup methods in order to actively meet users' demands, this paper primarily focuses on the key technologies of bringing it to the reality. Many opportunities exist for further improvement. For example, some functions have mathematical relationships, such as $\cos(x) = \sin(\frac{\pi}{2} - x)$ and $\tan(x) = \frac{\sin(x)}{\cos(x)}$. These functions can use common lookup tables, and compilers can allocate table resources for this group of functions rather than approaching the problem function by function. Another example is that certain function, such as $\sin(x)$, can appear in several programs, and the runtime table manger can take this information into account for further improvement. We believe that adaptive table lookups will play an important role in processors where energy efficiency is one of the primary design goals.

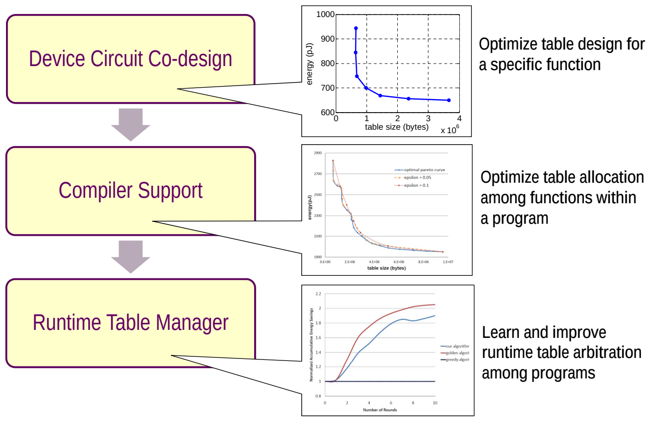

Device Circuit Co-design
Compiler Support
Runtime Table Manager
energy (pJ)
table size (bytes)
x 10^6
Optimize table design for a specific function
Optimize table allocation among functions within a program
Learn and improve runtime table arbitration among programs

Figure 11: The overall flow for adaptive table lookup in a general-purpose processor.

VI. Acknowledgements

This work was supported by the Center for Domain-Specific Computing (CDSC) funded by NSF "Expeditions in Computing" award 0926127. It is also supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] J. Cong *et al.*, "Architecture Support for Accelerator-Rich CMPs," *DAC*, pp. 843–849, 2012.

[2] P. Magnusson *et al.*, "Simics: A full system simulation platform," *Computer*, pp. 50–58, 2002.

[3] M. M. K. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, pp. 92–99, 2005.

[4] D. Das Sarma and D. Matula, "Faithful bipartite rom reciprocal tables," in *Proc. of Computer Arithmetic*, 1995, pp. 17–28.

[5] J. Stine and M. Schulte, "The symmetric table addition method for accurate function approximation," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 21, pp. 167–177, 1999.

[6] F. de Dinechin and A. Tisserand, "Multipartite table methods," *Computers, IEEE Transactions on*, pp. 319–330, 2005.

[7] N. Brisebarre *et al.*, "(m, p, k)-friendly points: a table-based method for trigonometric function evaluation," in *ASAP*, 2012, pp. 46–52.

[8] R. Riedlinger *et al.*, "A 32nm 3.1 billion transistor 12-wide-issue itanium processor for mission-critical servers," in *ISSCC*, pp. 84–86.

[9] R. Huang *et al.*, "Resistive switching of silicon-rich-oxide featuring high compatibility with CMOS technology for 3D stackable and embedded applications," *Applied Physics A*, pp. 927–931, 2011.

[10] K. Tsuchida *et al.*, "A 64mb mram with clamped-reference and adequate-reference schemes," in *ISSCC*, 2010, pp. 258–259.

[11] S. Kang *et al.*, "A 0.1um 1.8-v 256-mb phase-change random access memory (pram) with 66-mhz synchronous burst-read operation," *Solid-State Circuits, IEEE Journal of*, vol. 42, no. 1, pp. 210–218, 2007.

[12] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *ISSCC*, pp. 994–1007, 2012.

[13] G. Sun *et al.*, "Improving energy efficiency of write-asymmetric memories by log style write," in *ISLPED*, 2012, pp. 173–178.

[14] Y.-T. Chen *et al.*, "Static and dynamic co-optimizations for blocks mapping in hybrid caches," in *ISLPED*, 2012, pp. 237–242.

[15] D. Lee *et al.*, "High-performance low-energy stt mram based on balanced write scheme," in *ISLPED*, 2012, pp. 9–14.

[16] Y. Li *et al.*, "A software approach for combating asymmetries of non-volatile memories," in *ISLPED*, 2012, pp. 191–196.

[17] J. Cong and B. Xiao, "mrFPGA: A Novel FPGA Architecture with Memristor-Based Reconfiguration," in *NANOARCH*, 2011, pp. 1–8.

[18] Y. Chen *et al.*, "3D-nonFAR: Three-Dimensional Non-Volatile FPGA ARchitecture Using Phase Change Memory," in *ISLPED*, 2010, p. 55.

[19] C. Wen *et al.*, "A Non-volatile Look-Up Table Design Using PCM (Phase-Change Memory) Cells," in *Symposium on VLSI Circuits (VLSIC)*, 2011, pp. 302–303.

[20] D. Vengerov, "A reinforcement learning approach to dynamic resource allocation," *Engineering Applications of Artificial Intelligence*, pp. 383–390, 2007.

[21] D. Vengerov, "A reinforcement learning framework for utility-based scheduling in resource-constrained systems," *Future Generation Computer Systems*, pp. 728–736, 2009.

[22] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.