ENERGY EFFICIENT ARCHITECTURES FOR IRREGULAR DATA STREAMS

A Dissertation
Presented to
The Academic Faculty

By

Sriseshan Srikanth

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2020

ENERGY EFFICIENT ARCHITECTURES FOR IRREGULAR DATA STREAMS

Approved by:

Dr. Thomas M. Conte, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Dr. Erik P. DeBenedictis
Zettaflops, LLC

Date Approved: March 2, 2020

sarvam śrīkrṣṇārpaṇamastu

May all that is, be dedicated to the Lord

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my thesis advisor Prof. Thomas M. Conte for providing me with guidance and motivation based on his vast experience and distinguished career. I thank him for supporting me on multiple levels - research, development, career-wise, financially - in spite of the non-determinism in the paper acceptance process.

I would like to thank Dr. Erik P. DeBenedictis for being a repository of ideas that eventually ended up shaping the direction of my thesis. I also thank him for his prompt technical insight in our periodic interactions.

I would like to thank my thesis committee members from whom I've received formal and informal feedback on my research over the years.

I would like to thank the current and former student members, affiliated researchers and administrators of the TINKER research group and the CRNCH center, as well as colleagues and faculty from the Georgia Tech computer architecture community for our informal discussions over the years.

I would like to thank Dr. Lavanya Subramanian for our varied discussions during and after my internship.

An extremely special thanks to my dear family for being patient over the past several years of the unpredictable PhD process and the ups and downs that came with it. Last but not the least, I thank the cosmos and the forces that drive it.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

For a vast majority of microprocessor history, the single thread performance of computing systems has increased at a near-exponential rate. The rate of improvement, however, has slowed down significantly over time. Three fundamental bottlenecks, dubbed the memory wall, the locality wall, and the power wall, are commonly attributed as root causes to this.

The memory wall is a result of a memory access taking orders of magnitude longer than a compute operation at the CPU. While this has been typically circumvented to a great extent for dense data applications, thanks to caching and parallelism, applications that present a spatio-temporal locality bottleneck still suffer from poor performance.

The locality wall is a result of a class of emerging applications that operate on sparse, irregular streams of data that have become increasingly prevalent today. These sparse data applications exhibit low locality of reference and low degree of reuse, therefore resulting in low cache bandwidth utilization, or worse, low cache hit ratios (that exacerbate the problems posed by the memory wall), thereby limiting performance.

We hit the power wall over a decade ago because successive generations of transistors no longer consumed less power when they were made smaller. Furthermore, wires were no longer an expendable commodity, meaning that they incurred a significant latency and power overhead, thereby placing a limit on speculative approaches for performance. As a result, the power wall has limited the peak performance achievable for computationally dense applications.

This work finds that mitigating each of these three bottlenecks requires non-traditional solutions that tackle various forms of memory access irregularities. Therefore, in this thesis, I propose *energy efficient architectures for irregular data streams*.

A prime source for *memory access irregularities* stems from sparse data applications as they exhibit a low locality of reference.These applications have become critical today, and span important domains including machine learning, cybersecurity, graph analytics and high performance computing.

The first part of this thesis tackles the fundamental problem of latency of main memory accesses of sparse data applications while also significantly reducing overheads of data movement through the memory hierarchy, thereby improving energy efficiency. With the help of novel representations, algorithms and near memory processing, sparse applications are accelerated, on average, to within 8% of an idealized approach that executes the operations of the sparse kernel in zero time, while also reducing uncore energy consumption for the kernel by over $5\times$.

The second part of this thesis focusses on sparse data applications that exhibit high cache hit ratios, but suffer from low cache bandwidth utilization. Introducing accelerators that leverage the on-chip cache subsystem enables such sparse kernels to realize performance improvements of $2.2\times$ while also reducing energy consumption by $9\times$ on average.

The third and final part of this thesis focusses on dense (percentage of non-zeros close to 100%) data applications that leverage millivolt switches to tackle the power wall. These next generation devices are fast switching even at few tens of millivolts, but as a result, are vulnerable to thermal noise perturbations. Energy-efficient computational error correction is therefore needed to properly leverage these devices. A novel architecture that uses residue codes for computational error correction is proposed, although residue codes cause *memory access irregularities*. When compared with conventional architectures built using such ultra-low-power-devices, the proposed architecture not only corrects intermittent, stochastic bit errors in logic, but also realizes improvements of over $2\times$ in energy delay product, on average for computationally dense applications, in spite of the induced memory access irregulaties.

CHAPTER 1
INTRODUCTION

1.1 The Memory Wall

The general notion of the memory wall is that memory is slower than the CPU, and that it scales at a slower rate when compared to the CPU. As with any system, there are two components to determining the overall access time for any given request, and one strives to keep both of them to a minimum:

1. Queueing delay, decided by the available *bandwidth* to the memory system.
2. Service delay, decided by the raw *latency* to perform the read/write once the request reaches the head of the queue.

Typically, a memory system logically consists of a hierarchy of faster, higher cost per bit, smaller memory devices and slower, lower cost per bit, larger memory devices, in that order. For the purposes of this thesis, an SRAM cache is representative of the former fast memory, and a DRAM main memory device is representative of the latter. Here, the term "fast" applies to both bandwidth and latency.

For applications that have predictable memory access patterns, a variety of hardware and software techniques have been developed such that most accesses to the memory system are serviced by the cache, thus not hitting the memory wall.

Yet, for a variety of other applications such as those in graph analytics, high performance scientific applications and cybersecurity algorithms, the focus has shifted away from dense computationally intensive kernels, and towards light-weight processing of irregular data. These hyper-sparse applications exhibit low locality of reference and a low degree of reuse such that cache hit rate is low, thereby exposing DRAM access time onto the critical path.

Fortunately, DRAMs have made rapid progress on bandwidth (and capacity), improving by $20\times$ (and $128\times$) respectively over the past two decades. However, DRAM latency has reduced by only 30% [1]. As a result, in an effort to hide the service delay, the slack in queuing delay as well as the increased relative availability of compute cores has been exploited by programmers to parallelize such applications and maximize memory level parallelism (MLP), even if it comes at the cost of increased movement of redundant data across the memory hierarchy.

A recent, well-cited study of warehouse scale applications running in Google's data-centers reports that memory latency is more important than memory bandwidth [2]. The key insight being, with a reduced effective service delay, the benefits of memory level parallelism (MLP) are compounded. It has also been reported that the cost of moving data between caches and main memory accounts for a large portion of total energy in several application domains and is cited as the key challenge in future exascale systems, consuming two orders of magnitude higher energy than that of a floating point operation [3].

The first part of this thesis proposes a new and promising mechanism to tackle the fundamental main memory latency-bound inefficiencies of sparse data streams, while also significantly reducing overheads of data movement through the memory hierarchy. This mechanism leverages a novel DRAM aware, hierarchical two-level representation as well as associated algorithms for operations on hyper-sparse data, which can be used in conjunction with near memory processing.

1.2 The Locality Wall

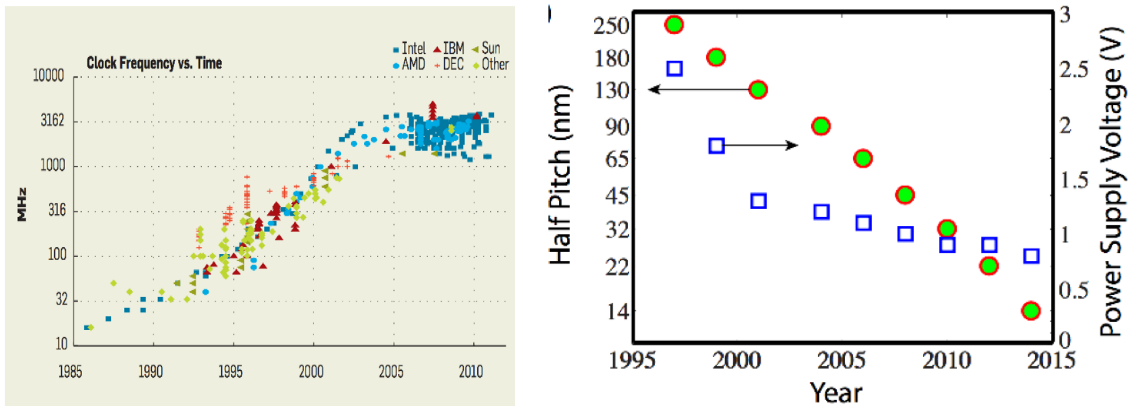
The problem of hitting the locality wall [4] has arisen because several important applications in the domains of graph analytics, high performance scientific computing, cybersecurity and machine learning now exhibit low spatio-temporal locality of reference. As a result, these sparse data applications suffer from either, (i) low cache hit ratio , or, (ii) high cache hit ratio but low cache bandwidth utilization

While the former issue is tackled in the first part of the thesis through memory-centric computation (as described earlier), the second part of this thesis tackles the latter via processor-centric acceleration of sparse data streams by automatically extracting sequential locality through intelligent data marshaling within the on-chip cache subsystem. A combination of mechanisms presented in the first and second parts of the thesis are recommended for sparse applications whose cache behavior is unknown.

1.3 The Power Wall

Dennard scaling [5] refers to the phenomenon of transistors retaining their power density as they scaled down. Historically, this has been a great driver in providing near-exponential improvements in performance of microprocessors over time. However, Dennard scaling ended over a decade ago [6], and as a result, adding more and more transistors no longer improved performance without also significantly increasing energy consumption. In other words, we hit the power wall in 2005, which meant that increasing clock frequency was no longer a technique to improve performance (Figure 1.1a). Furthermore, single core performance plateaued as a result of the power wall although there is more instruction level parallelism (ILP) inherent in programs, waiting to be exploited [7, 8, 9]. A sure-shot mitigation technique is to strive to improve the fundamental single core power-performance profile from the ground up, and the obvious benefits of doing so would be compounded in general when extended to multiple cores and specialized accelerators. This is of particular interest for computationally intensive dense kernels found in machine learning, signal processing and high performance scientific applications.

Dynamic power scales proportionately with frequency and with the square of operating voltage, therefore, lowering V_{dd} is more beneficial. However, V_{dd} does not seem to have reduced to much below a volt for over a decade (Figure 1.1b). Although the theoretical lower limit for V_{dd} for inverter functionality is $2\frac{kT}{q}$ (or about $36mV$) [12, 13, 14], there are challenges to operating at this level [15]. With conventional MOSFETs, reducing supply



(a) Single core performance has plateaued since the end of Dennard scaling [10]. **(b)** Supply voltage has not reduced much below 1 V [11].

Figure 1.1: Historic trends of microprocessors.

voltage below ˜0.7V results in increased switching delay, irrespective of channel length. Coupled with their gentle subthreshold slope ($> 60mV/decade$), the upshot is that V_{dd} reduction actually causes higher leakage energy, negating the benefits of the reduction in the first place and furthermore forcing a significantly slower clock rate.

Theis and Solomon [16] suggest that new device concepts [17] within the purview of two-dimensional lithography technology, such as tunneling FETs, enable reduction of the $\frac{1}{2}CV^2$ energy to small multiples of kT, without resulting in a significantly low switching speed [18] when compared to MOSFETs. Similarly, research on ferroelectric transistors, *aka* negative capacitance FETs (NCFETs) demonstrates a sub-$60mV/dec$ slope as well as a higher drive current [19, 20, 21, 22], both of which are necessary in rendering V_{dd} reduction beneficial to energy reduction without significantly sacrificing performance [23], when compared to MOSFETs.

These next generation devices are fast switching even at few tens of millivolts, but as a result, are vulnerable to thermal noise perturbations. This translates into *intermittent, stochastic bit errors in logic*. With signal energies approaching the kT noise floor, future architectures will need to treat reliability as a first class citizen, by employing efficient computational error correction.

Codes based on a Redundant Residual Number System (RRNS), which represents a number using a tuple of smaller numbers, are a promising candidate for implementing energy-efficient computational error correction, although they result in irregular memory accesses. The third and final part of this thesis proposes a novel architecture, which when used atop ultra-low-power devices and residue codes, has the potential to punch through the power wall to restart single core performance scaling via architectural advances abandoned at the end of Dennard scaling a decade ago.

1.4 Thesis Statement

Energy efficiency and performance of data irregular applications can be improved via near memory and near processor sparse data stream acceleration and address remapping.

1.5 Thesis Organization

Chapter 2 provides an introduction to irregular data streams that form the context and motivation for the architectures proposed as part of this thesis.

Chapter 3 presents the first part of this thesis, which targets sparse data applications that suffer from the phenomenon wherein main memory latency is on the critical path and tackles the intersection of the memory wall and the locality wall. This chapter presents solutions from DRAM-centric representations and algorithms to methods of energy-efficient implementation. The peer-reviewed publications that validate this chapter are [24, 25, 26, 27].

Chapter 4 presents the second part of this thesis, which targets sparse data applications that do not effectively utilize the on-chip and off-chip cache bandwidth available in modern vector processors and tackles the locality wall. This chapter presents a programmable accelerator that takes advantage of on-chip cache real estate to maximize cache bandwidth utilization for sparse applications from several important domains, with minimal programmer intervention. The peer-reviewed publication that validates this chapter is currently under review.

For presentation clarity, Chapters 3 and 4 are categorized as DRAM-centric for hyper-sparse and cache-centric for moderately-sparse data applications respectively. However, the distinction between hyper-sparse and moderately-sparse is not rigid and this thesis recommends using a combination of these two classes of techniques.

Chapter 5 presents the third and final part of this thesis, which targets computationally dense applications and tackles the power wall. This chapter begins with introduction of the concepts of ultra low power device technologies, residue codes, and the general architectural approach to designing cores that are well-suited to these unconventional paradigms. It then addresses the most significant bottleneck that one would face when using a residue number system based compute core, where the memory addresses generated are no longer in the conventional weighted binary format. The peer-reviewed publications that validate this chapter are [28, 29, 30, 31].

An overall conclusion is found in Chapter 6.

CHAPTER 2
IRREGULAR DATA STREAMS IN MODERN APPLICATIONS

This chapter introduces the sources of irregularity in the data streams that motivate the architectures proposed in this thesis.

2.1 Taxonomy of Irregular Data Streams

2.1.1 Inherent vs Induced

The first kind of data irregularity stems from zeros being inherent in the application. For example, in a graph, not all vertices have edges that connect them. In general, such zeros are omitted from representations and processing, resulting in irregularity. Such sparse applications are particularly common in domains of graph analytics, high performance computing and cybersecurity.

Another source of data irregularity is due to explicit modification of the original application by some entity in the computing stack for the purposes of improved performance and/or energy efficiency. Two such induced irregularities are of interest to this work: (i) pruned deep neural networks, and (ii) computationally error-tolerant post-Moore processing. The former is an emerging technique of reducing the computational intensity of machine learning workloads by leveraging their algorithmic tolerance to 'dropping' weights. This results in induced zeros, thus resulting in sparsity. The latter is a technique of attaining error tolerance in computationally intensive applications via transformation of the address and data to a different domain, known as the redundant residue number system, which will be introduced in Section 2.4. Unlike the former, this class of induced-irregular accesses are an example of irregularity that is not due to interspersed zeros.

2.1.2 Conventional Sparse vs Post-Moore Dense

Data irregularity can also be classified as that which is due to interspersed zeros (inherent or induced), and that which is irregular simply due to a non-obvious access pattern that lacks spatial locality (induced, in this work), that has a bijective transformation to a pattern that has high spatial locality.

The former category is typically known as 'sparse', whereas the latter is fundamentally dense and is relevant to computationally error-tolerant post-Moore processing.

2.1.3 Hyper-Sparse vs Moderately-Sparse

In the sparse category of data-irregular applications, a classification may be made based on the degree of sparsity, or the fraction (percentage) of non-zeros in the data stream. If the percentage is under approximately 1%, these are known as hyper-sparse, whereas the others are moderately-sparse. The threshold of 1% is not a rigid one. What separates hyper-sparse applications from moderately-sparse applications is the impact of main memory latency on the critical path. Moderately-sparse applications typically have a much higher cache locality than hyper-sparse applications, and are therefore more tolerant to slower off-chip memories. A dense data stream has a percentage of close to 100%.

Hyper-sparse data streams are seen typically in large scale graph analytics, high performance computing and cybersecurity. Moderately-sparse data streams are seen in machine learning, in addition to the domains mentioned above.

Depending on the level of sparsity, these result in significant energy consumption, due to the following sources of overhead.

- Poor cache performance due to poor on-chip and off-chip cache bandwidth utilization.
- Poor cache performance due to poor hit ratios.
- Poor DRAM performance because of a high number of row activations and precharges for a relatively smaller portion of useful data.

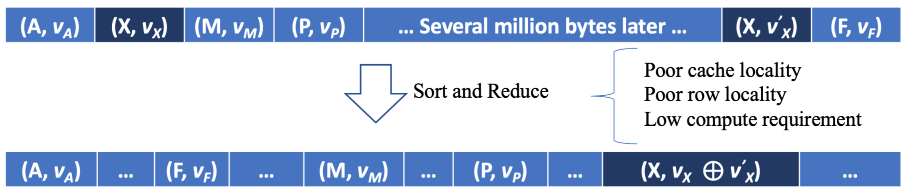


Figure 2.1: Reduction of key-value pairs in a sparse data stream.

- Increased data movement through the memory hierarchy:
  - Within a DRAM bank, due to redundant row activity.
  - Within a cache structure.
  - Between constituents of the cache hierarchy, the processor and DRAM, most of which is redundant.

2.2 Sparse Reductions as a Programmable Abstraction

The utility of sparse data is well known to the community due to its prevalence in machine learning algorithms (e.g., support vector machines (SVM) [32], text analytics [33]), in scientific-computing applications [34] (e.g., Schur complement method in hybrid linear solvers [35], algebraic multigrid (AMG) methods [36], finite element analysis [37], molecular dynamics [38], many-atom systems [39]), in graph analytics (e.g., breadth-first-search (BFS) [40], PageRank [41], minimum spanning tree (MST), single source shortest path (SSSP) [42], matching [43], contraction [44], reachability [45], clustering [46], triangle counting [47], and cycle detection [48]), and in cybersecurity [49, 50]. Furthermore, novel deep learning algorithms that are being proposed employ network pruning and effectively "sparsify" [51, 52, 53, 54, 55, 56] them to reduce memory footprint as well as required computation.

A significant and important operation for sparse data streams occurs during associative array reduction (Figure 2.1). This is a set of associative operations performed on the values of two/more key-value pairs that share the same key. General terms for key-value pairs are

```
for n in oc:
 for weight, (r, s) in nzw:
  offset = f(n, c, r, s, ifm.dim)
  for y in ofm.height_dim:
   for x in ofm.width_dim:
    ofm[n][y][x] += weight * ifm[offset]
```

Figure 2.2: Sparse Convolution as a sparse-matrix-dense-matrix multiplication, adapted from [58]. *oc*: Set of output channels where *nzw*: set of non-zero-weights (sparse filters of dimension $R \times S$ each) represented in compressed sparse row (CSR) format [59]; *ofm*: output feature map; and, *ifm*: input feature map; *offset* is a linear combination *f* of *n*, input channel (*c*), non-zero-weight index (*r*, *s*) and the *ifm* dimensions.

typically *kv-pairs*, *records*, *index-nonzeros* (for sparse matrices) or *tuples*, where the key, value and reduction operator are application dependent. Examples of these are introduced next. A programmatic view of sparse reductions is shown in Figure 4.1.

2.2.1 Deep Learning

Deep convolutional neural networks requires a large number of parameters (AlexNet has 60 million [57], for instance). The fully-connected layers' parameters are primarily responsible for the memory footprint of these networks. The operations due to convolution layers' parameters are primarily responsible for computation time. Accordingly, *pruning* techniques [55, 51, 58] have been proposed with an objective of reducing memory footprint and/or improving performance, but result in moderately sparse structures, the latter of which is the focus of SortCache (Chapter 4).

Highly compressed convolutions, or sparse convolutions, can be viewed as a sparse-matrix by dense-matrix multiplication (SpMDM), as shown in Figure 2.2. More formally, such a multi-dimensional convolution can be defined as $O(n, y, x) = \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} W(n, c, r, s) I(c, y + r, x + s)$, where there are C input channels and $R \times S$ filters. When W and I are flattened appropriately, the values at the same given (y, x)-position of all N output channels can be computed collectively as a sparse-matrix by dense-vector multiplication, which when applied to all output positions results in SpMDM described above [58].

Notice how this is similar to sparse reduction as depicted in Figures 2.1 and 4.1, where the key is a linear combination of n, y and x, and the value is the non-zero partial product obtained by multiplying the corresponding filter weight with the input feature map.

2.2.2 Graph Analytics

While graph analytics is a very broad domain, this work focuses on three important applications that span various instances of sparse reductions. Each of these applications can be expressed as a generalized sparse-matrix by dense-vector multiplication (SpMV) problem with different "multiply" (partial product generation) and "add" (reduce) operations. The sparse matrix here is the transpose of the adjacency matrix, and the vector is an application-dependent state vector spanning all vertices of the graph. SpMV maps directly to sparse reductions in that the key is the output matrix index and the values are the non-zero partial products corresponding to that matrix index.

There are two approaches to SpMV: inner-product and outer-product. While the former is better for locality of output vector updates ("add"), the latter is better for input vector reads ("mul"). From a graph analytics perspective, these are sparse-pull-dense-push (or just "pull") and dense-pull-sparse-push (or just "push"), respectively. Whether to push or pull or employ a hybrid is an active research area [60]. Push is particularly attractive because it reduces locality-based inefficiencies, performance-ambiguities and redundancies in reading the input vector, especially in algorithms where not all vertices are active in a given iteration, as is demonstrated by GraphMat [61], a well-cited software framework for easily-programmable, matrix-based graph analytics.

The drawbacks of push are poor locality and potential synchronization costs with multiple pushers for sparse reductions. However, hardware accelerators that explicitly manage data (such as [62, 63, 27]) are able to overcome both of the these drawbacks, at various levels of the memory hierarchy. Finally, these graph problems are repeated applications of SpMV, and the reduction operators are associative. This allows for asynchronous processing of sparse reductions, which SortCache leverages in a manner that maximizes effective cache bandwidth utilization. In summary, push-based SpMV is efficient in gathering the input and offers sparse reductions as the bottleneck, making it a prime candidate for hardware acceleration.

Breadth-First Search

BFS begins at a given source vertex and iteratively explores all vertices in its component. From an SpMV perspective, the state vector maintains the first parent node obtained during traversal. The vertex index of such a parent node is specified as a result of the *mul* operation (assignment), and the *add* operation (conditional assignment) assigns the parent node to the corresponding element in the state vector if and only if it was previously unassigned.

Single-Source Shortest Path

SSSP is an example of another iterative-search application. In addition to performing graph traversal, it computes the minimum cumulative distance at each frontier. Formally, at each iteration, for a vertex v, $d(v) = min_{u \in N^-(v)}\{d(u) + w(u,v)\}$, where d is the current minimum distance to a given vertex from the source vertex, w is the edge weight, and $N^-(v)$ is the incoming neighborhood of v. From an SpMV perspective, the state vector represents the distance to each vertex from the source. The *mul* operation is summation (of distance to neighbor and edge weight), and the *add* operation is the minimum operator (to find the current minimum distance at this iteration).

PageRank

PageRank computes the probability of a random walk through edges ending in a given vertex. Formally, at each iteration i, $P_{i+1}(v) = r + (1-r)\sum_{u \in N^-(v)} \frac{P_i(u)}{|N^+(u)|}$, where r is a constant normalized randomness fraction, $N^-(u)$ and $N^+(u)$ are the incoming and outgoing neighbors of vertex u. From an SpMV perspective, the state vector maintains the PageRank of each vertex. Unlike BFS and SSSP, PageRank has a consistently dense state vector. The *mul* operation is division (of PageRank of incoming neighbor and its out-degree), and the *add* operation is summation (over all incoming neighbors).

The equivalence between graph analytics and sparse linear algebra is well-studied [59], and similar transformations from graph to SpMV problems can also be found in [62]. For brevity, initial conditions, convergence criteria etc. are omitted from the above.

2.2.3 SpGEMM - High Performance Computing

Efficient generalized sparse-matrix by sparse-matrix multiplication (SpGEMM) is critical to several HPC and graph analytics applications [64]. In a manner similar to SpMDM and SpMV described above, SpGEMM maps directly to sparse reductions in that the key is the output matrix index and the values are the non-zero partial products corresponding to that matrix index.

$C = A \times B$ can be performed using row-wise (RW) or outer-product (OP). Both of these algorithms generate a *sparse set of partial products* that need to be reduced (summed / accumulated, in this case).

With RW, C is computed one row at a time by fetching one row of A and all rows of B in a row-wise manner such that $C_{i,:} = \sum_k A_{i,k} \times B_{k,:}$. As a result, all partial products generated in this manner prior to reduction (accumulation) are confined to a single output row of C.

With OP, the k^{th} column of A is multiplied by the k^{th} row of B to generate the k^{th} partial product matrix. Reduction of all such partial product matrices results in the final output matrix such that $C = \sum_k C^k = \sum_k A_{:,k} \times B_{k,:}$.

Observe that OP fetches the input exactly once but generates a large space of temporaries (partial products), whereas RW fetches the input several times although it enables caching of temporaries. It can be theoretically shown that OP fetches a factor of N fewer inputs

Table 2.1: Summary of moderately-sparse workloads evaluated. The number of sparse reductions refer to the number of input records presented to SortCache (Chapter 4). The percentage of non-zeros refers to the ratio of non-zero elements in the corresponding input matrix. The Amdahl's percentage refers to the contribution of sparse reductions to application overall runtime, excluding I/O.

Name	Application	Domain	Num Sparse Reductions	% non-zeros	Amdahl's %
conv2	Sparse Convolution	Machine Learning	8.2 G	14.4	55%
conv3			1.7 G	6.9	74%
conv4			1.2 G	8.2	61%
conv5			1.2 G	11.5	75%
2cubes_sphere	BFS, SSSP, PageRank	Electromagnetics	1.5 M, 3.0 M, 49.5 M	6.6, 3.4, 0.2%	78, 74, 49%
ca-CondMat		Collaboration N/W	0.2 M, 0.2 M, 7.5 M	11.7, 9.2, 0.3%	71, 80, 64%
email-Enron		Email N/W	0.4 M, 0.7 M, 16.5 M	9.3, 4.9, 0.2%	67, 81, 53%
m133-b3		Combinatorics	1.6 M, 20.5 M, 48.01 M	12.5, 1.0, 0.4%	86, 77, 53%
mario002		2D/3D Problem	1.9 M, 4.4 M, 57.9 M	20.9, 8.8, 0.7%	65, 61, 57%
p2p-Gnutella31		Peer N/W	0.3 M, 0.5 M, 11.5 M	21.2, 13.5, 0.5%	65, 90, 66%
patents_main		US Patent Citations	1.1 M, 1.7 M, 46.0 M	20.8, 13.8, 0.5%	75, 72, 60%
TSOPF_RS_b39_c7		Power N/W	0.5 M, 0.9 M, 17.1 M	3.0, 1.6, 0.1%	70, 51, 51%
wiki-Vote		Wiki Votes N/W	0.2 M, 0.4 M, 8.1 M	3.5, 1.8, 0.1%	51, 69, 51%
CAG_mat364	SpGEMM	Combinatorics	586 K	10.2%	72%
ex2		Fluid Dynamics	464 K	13.8%	68%
filter2D		Model Reduction	23 K	0.4%	67%
Journals		Undirected Graph	203 K	78.5%	66%
LeGresley_4908		Power N/W	237 K	0.1%	75%
mhda416		Electromagnetics	188 K	5%	63%
qc324		Electromagnetics	388 K	25.5%	66%
rotor2		Structural	85 K	1.7%	68%
Si2		Quantum Chemistry	113 K	3%	76%
Trefethen_700		Combinatorics	63 K	2.6%	74%
west0381		Chemical Process Sim	12 K	1.5%	75%

than RW assuming uniform random matrices. When these are simulated with real input considered in this work, $30000\times$ fewer input accesses are seen with OP. This insight is also demonstrated by [65]. Given its higher ops per byte fetched, OP is chosen as the choice of implementation.

2.2.4 Firehose - Cybersecurity

Firehose [49] is a collection of open-source stream processing benchmarks designed to represent cyber-security applications. It models a variety of sparse network event distributions and their subsequent (soft) real-time analysis, *requiring incremental updates to sparse data*. It consists of three stream generator algorithms as its front-end, namely power law, active set and two-level, to simulate a variety of network traffic. As with the various classes of workloads described above, each generator produces a stream of key-value pairs. Here, key is a proxy for an IPv6 address and value is a counter (+bias bits) associated with each key. The goal of the benchmark suite is to track the number of identical keys received, and

Table 2.2: Hyper-sparse input and application properties. nnz_{stream} denotes the number of non-de-duplicated kv-pairs in the input stream, out of which $\%repeated$ of the pairs have identical keys whose values have to be reduced. The Amdahl's fraction is calculated using a scalar binary tree based reducer (described in Section 2.3.1), and indicates a high relative importance from accelerating reduction (compute operations and associated memory reads/writes). Finally, all workloads, with the exception of *pl*, exhibit low locality of reference in their streams. The *pl* stream has a high degree of reuse owing to the fact that it was generated using a relatively low range of static keys (100,000) by default. Therefore, this work reports results for *pl*, but *omits them from averages*.

Application	Input	Domain	nnz_{stream}	% repeated keys	Amdahl's fraction	LLC Miss Ratio of kernel
SpGEMM	2cubes_sphere (2cu)	Electromagnetics	6.3 M	50	0.8	0.24
	belgium_osm (bel)	Road Networks EU	1.5 M	< 1	0.5	0.49
	ca-CondMat (caC)	Undirected Graph	700 K	28	0.9	0.36
	mario002 (mar)	2D/3D Problem	2.7 M	22	0.8	0.58
	netherlands_osm (net)	Road Networks EU	2 M	< 1	0.5	0.57
	p2p-Gnutella31 (p2p)	Directed Graph	500 K	< 1	0.7	0.45
	patents_main (pat)	Wt. Directed Graph	2.6 M	12	0.8	0.56
	roadNet-CA (roa)	Road Networks US	3 M	1.5	0.6	0.55
Firehose	Power Law (pl)	Cyber Security	500K/1M/5M	88/91/96	n/a	0.01
	Active Set (as)			5/8/17	n/a	0.33/0.41/0.54
	Two Level (tl)			28/30/31	n/a	0.29/0.38/0.53

* The Amdahl's fraction for Firehose is not shown because the kernel *is* the application.

whenever the number of identical keys exceeds a certain threshold, the key is flagged (and the bias bits associated with such keys are then tested against the ground-truth bias bits included in the input).

As can be expected of anomalous network events, these three generators are designed to be unpredictable and to have a varied dynamic range. Therefore, the incoming key-value stream is sparse and anomaly detection can be performed by applying a reduction (saturating increment) operator on elements with identical keys.

Tables 2.1 and 2.2 summarizes several key characteristics of the workloads evaluated. Most notable is the relatively high Amdahl's percentage of sparse reductions across multiple domains and levels of sparsity.

2.3 Limitations of Prior Work on Sparse Reductions

The section summarizes prior work on software-based and hardware-based acceleration of sparse reductions. The techniques proposed in Chapters 3 and 4 of this thesis are demonstrably superior to, or are complementary and beneficial when used in conjunction with these approaches.

Recall that sparse reductions are a combination of three components: sparse gather from a given key, application of reduction operation to the value associated with that key, sparse scatter to the key.

2.3.1 Software Reducers

Most of prior work on software sparse reductions are scalar word accesses. Fundamentally, these suffer from redundant data movement through the memory hierarchy as memory arrays perform best only when there is sequential locality in the access stream. Unfortunately, sparse reductions possess such sequential locality if and only if the algorithm is modified to generate sparse reductions in-order. This is not possible in several classes of applications such as dynamic/streaming graphs and Firehose. In other cases, such modification introduces inefficiencies into the generators, as evidenced by the row-wise vs outer-product discussion above.

Conventional Sparse Representations

There are several sparse representations that are widely used today [59, 66, 67, 68, 69]:

1. **Implicit.** An array-based representation of sparse data is when the key is not explicitly stored, but is implicitly inferred from the offset from the base array address. However, the downside is that zeros have to explicitly allocated memory, thus resulting in significant and redundant storage (footprint) overhead. Furthermore, reductions on the array result not only in redundant data movement of non-zeros, but that of the zeros as well. Therefore, such an implicit representation is seldom used, unless the workload is known to be dense or likely to be mostly-dense.

2. **Coordinate - COO.** To avoid the performance, energy and storage overheads of storing zeros in sparse data, a coordinate system has been proposed. In addition to storing

the non-zero values, their corresponding indices (keys) are also stored. The example used above to introduce sparse reductions in Figure 2.1 uses this representation. The kv-pairs are typically stringed together either as a linked-list or as a pair of vectors (one for the key and one for the value). Furthermore, these representations can be sorted by key, or remain unsorted, resulting in ordered coordinate, and unordered coordinate representations. First, ordered, vector-based representations allow for rapid lookup (sparse gather), but their update performance (sparse scatter) suffers if insertions have to be made into the middle of the vectors. Insertion into the middle of vectors causes significant data movement as all the kv-pairs that occur at a later point in space of the insertion point have to be shifted. Second, ordered, list-based representations allow for equally fast sparse gather or scatter (even if insertions have to be made), however, performance degrades as the distance from the head node increases. Finally, unordered coordinate representations allow for constant time insertions, but both sparse gather and scatter suffer.

3. **Compressed - CSR/CSC/DCSR/DCSC.** The memory footprint of ordered-coordinate representations can be potentially reduced (while also resulting in reduced data movement) if a contiguous set of kv-pairs share the same key, or share the same portion of the key. For example, if the keys represent a two dimensional matrix index, then all the non-zeros of the same row need not all repeat representing the row index. The most common implementation of this compression deploys 3 vectors: an offset vector, a column vector and a value vector. The latter two are near-identical to the ordered, vector-based representation above, with the distinction being that the row information is encoded in the offset vector. This is known as Compressed Sparse Row (CSR), which is the transpose of Compressed Sparse Column (CSC), where the offsets encode column indices and the second vector stores row indices explicitly instead. For an N dimensional square matrix with $nnz < N^2$ non-zeros, these vectors grow as $O(N)$, $O(nnz)$ and $O(nnz)$ respectively. If there are very few or no non-zeros

in a given row, such as with hyper-sparse data, CSR/CSC pay a price due to the offset vector and the compression ends up causing bloating instead. More recent techniques such as Doubly-CSR/CSC (DCSR/DCSC) solve this issue by introducing yet another level of abstraction by not encoding the row (column) numbers of those rows (columns) that have no non-zeros, and are successful in reducing the bloat if there are a significant number of contiguous rows (columns) that have no non-zeros. Therefore, apart from their memory footprint characteristics, all of these compressed representations are fundamentally rooted in the ordered, vector-based representations, and therefore share the same issues [70] with insertions into the middle of a vector that sparse scatters may induce.

Therefore, none of these approaches are able to robustly and efficiently handle both sparse gathers and scatters, and therefore, sparse reductions.

Hashmaps

An alternative to linked-list based or vector-based coordinate representation is a hashmap-based coordinate representation. In doing so, the efficiency of sparse reductions hinges on the hash strategy.

One approach is to enforce a global ordering, such as with a scalar binary search tree, where each node of the tree corresponds to one kv-pair, with pivot comparisons defined to compare keys. When the tree is balanced, lookup, update and insertion (and therefore sparse reductions) can be done in a logarithmic number of steps rather than a linear number of steps in the list- or vector-based approaches. Also, this approach lends itself as a natural baseline in this work, given that this thesis proposes the use of a vectorized binary search tree to accelerate sparse reductions. In particular, `std::map` is a CPU-only associative container that contains unique key-value pairs, and is implemented as a balanced red-black binary tree [71], allowing for lg N insertion complexity, where N is the number of unique key-value pairs inserted.

Another approach is to aim for a constant time hashmap, such as std::unordered_map. However, its performance with real workloads considered in this work is similar to, or worse than std::map. This is attributed to an inefficient default hash function with long chains, resulting in frequent linear number of steps. Instead, a non-STL reducer built using the kokkos framework is a more suitable hashmap, and is in-fact the state-of-the-art software reducer.

The Kokkos HashMap Accumulator [72] is representative of a CPU-GPU sparse reducer, consisting of 4 parallel arrays where the hashmap is stored in a linked list structure. The *Ids* and *Values* arrays store the keys and values respectively. The *Begins* array holds the beginning indices of the linked list corresponding to the hash values, and the *Nexts* array holds the indices of the next elements within the list. While this approach is significantly better than the baseline, it is not explicitly designed to reduce DRAM ACT/PRE overheads or improve effective cache bandwidth.

2.3.2 Hardware Reducers

Even the best performing software reducers above are scalar in their manner of data reorganization, meaning that there is still significant scope for cache-centric and memory-centric optimizations of sparse reductions that inherently lack sequential locality.

Sparse-to-Dense Convertors

GraFBoost [62] is the most recent, state-of-the-art, out-of-core approach (such as GraphChi [73], X-stream [74], etc.) for DRAM-centric sparse reductions. It accelerates random key-value pair updates through a specialized sort-reduce accelerator which sequentializes fine-grained random accesses for data pairs stored in secondary (Flash) storage. The accelerator makes use of increasingly larger levels of reductions, all of which are based on multi-level merge sort. The near-memory reducer relevant to this work is a 16-1 merge tree comprised of 2-1 bitonic mergers. While this work helps reduce redundant data movement by staying cognizant of DRAM row inefficiencies when compared to the scalar approaches above, there are two drawbacks of this approach, First, a source of inefficiency is the requirement to successively merge exponentially increasing batches of kv-pairs. Second, from a functionality perspective, GraFBoost requires the entirety of the data stream that has to be reduced be made available in memory before the in-memory reducer can be launched, rendering it unsuitable for accelerating workloads such as Firehose (and dynamic/streaming graphs) that require support for incremental update.

SPiDRE [75] performs LLC prefetching via a programmer-specified rearrange function, therefore improving cache hit ratios and also useful cache bandwidth. Although this improves sparse-gather performance, sparse-scatter is not accelerated, and therefore cannot accelerate sparse reductions. However, the techniques presented in this thesis can benefit from SPiDRE as it can potentially accelerate the front-end generation of sparse reductions and downstream sparse gather-only operations as well.

Unlike the near-memory GraFBoost, PHI [63] is a near-cache approach to reducing DRAM traffic for the sparse reductions problem. In particular, PHI focusses on push-based updates in the context of graph analytics. It successfully leverages temporal locality in keys via coalescing, i.e., by treating portions of the cache hierarchy as a write buffer. Traffic to main memory is further lowered by batched updates. However, during their in-cache update phase, which is fundamentally scalar in nature, unless the stream of sparse reductions has significant sequential locality to begin with (which, according to Figure 4.2, is not common unless the program was explicitly vectorized), PHI is unable to effectively utilize the wide cache bandwidth available in modern processors, which is where techniques proposed in this thesis (particularly Chapter 4) excel.

In-Memory Acceleration

There is another class of hardware accelerators that leverage the analog properties of memory subarrays [76, 77, 78, 79] to perform useful computations. For example, bitline

Table 2.3: Summary of advantages and limitations of prior work on accelerating sparse reductions.

Type	Approach	Primary advantage	Primary impediment to efficient sparse reduction
Sparse representation	Implicit	Mostly-dense data	Redundant storage and data movement
	Ordered COO (vector) [82]	Sparse gather	Data movement for sparse scatter
	Ordered COO (list) [59]	No explicit data movement	Requires traversing the entire structure for every operation potentially
	Unordered COO [59]	Constant time insertion	
	CSR / CSC [83]	Sparse gather, lower storage for several sparse patterns	Data movement for sparse scatter unless performed in key order
	DCSR / DCSC [67]		
Hashmap based reducer	Scalar BST [71]	Bounded complexity	Subarray block/row agnostic
	Unordered map [84]	Constant time on average	Approaches worst case due to collisions
	Kokkos hashmap [72]	Optimized hash function	Subarray block/row agnostic
Sparse to dense hardware convertor	GraFBoost [62]	DRAM row conscious	Unable to handle incremental updates, data-agnostic all-all comparison
	SpiDRE [75]	Sparse gather	Sparse scatter not supported
	PHI [63]	Lower DRAM traffic	Scalar updates don't leverage cache b/w
In-memory accelerators [76, 77, 78, 79]		Reduced data movement	Requires higher level data layout strategy

computing [80, 81] activates multiple word-lines in parallel and infers an *and* (or *nor*) operation by sensing the shared bitline (or its complement).

While such approaches are able to reduce data movement across the memory hierarchy in general, in the absence of a higher level entity that handles data layout, they are unable to accelerate irregular problems such as sparse reductions by themselves as there are no guarantees that the target data are wordline or bitline aligned. Furthermore, these techniques generally require intrusive changes to the memory hierarchy, thereby hindering commercial adoption.

A summary of the discussion above is presented in Table 2.3. The mechanisms proposed in Chapters 3 and 4 strive to improve upon the advantages of these approaches while addressing their disadvantages.

2.4 Data-irregular Accesses in Error Tolerant Post-Moore Computing

Next generation devices are fast switching even at few tens of millivolts, but as a result, are vulnerable to thermal noise perturbations, as will be described in further detail in Section 5.1. This translates into *intermittent, stochastic bit errors in logic*. With signal energies approaching the kT noise floor, future general-purpose post-Moore architectures will need to treat reliability as a first class citizen, by employing efficient computational error correction.

One of the most promising techniques to efficient computational error correction is via the Redundant Residue Number System (RRNS). The RRNS representation of a number is in the form of a tuple of low bit-width *residues*. The set of residues is obtained via the modulus (remainder) when the number is divided by a pre-determined set of co-prime numbers. A key property of RRNS is that the tuples can operate independent of each other, for addition, subtraction and multiplication. Besides having performance and energy-efficiency benefits from the resulting higher level of bit-level parallelism in arithmetic, RRNS provides the additional benefit of fault isolation between the residues. An error that occurs (storage, communication or computation) in a given residue does not affect the value in any of the other residues. As a result, carefully introducing redundant residue(s) allows for reconstruction of the errant residue, thereby effectively performing error correction, even if the error was due to a faulty logic operation.

A chief obstacle to successfully leveraging the RRNS representation is that conversion to and from the conventional position-weighted binary representation is expensive. This means that it is prudent to maintain all data in the RRNS domain itself, thus including literals, immediate operands, register data, memory contents and therefore memory addresses to allow for indirect addressing. However, as a consequence, spatial locality in conventional dense (data-regular) data applications is broken.

For example, the address referred to conventionally as 0x00 would be represented as 0x0, 0x0, 0x0, 0x0 in an RRNS with 4 residues, and the address 0x01 would be 0x1, 0x1, 0x1, 0x1 in the RRNS domain. Naively treating these tuples, such as via concatenation, results in an address difference of 0x1111 of what should be adjacent bytes, as intended by the original application.

In other words, leveraging RRNS for efficient computational error correction in post-Moore processors induces a new kind of data-irregularity that is different from sparsity-based irregularity that has been described earlier in this chapter. More details about the nature of these RRNS-induced irregularity challenges and architectures to tackle them is presented in Chapter 5.

CHAPTER 3
METASTRIDER: ARCHITECTURES FOR SCALABLE MEMORY-CENTRIC REDUCTION OF SPARSE DATA STREAMS

This chapter describes a memory-centric approach to improving energy efficiency to sparse data applications in which data-irregularity is inherent, and main memory latency is on the critical path of the application.

3.1 Memory centric approach to improving energy efficiency

Recall from Section 2.1.3 that low locality of reference, low degree of reuse and low compute-to-communicate ratios are intrinsic characteristics of hyper-sparse data streams, rendering even large caches ineffective. Therefore, main memory (DRAM) access latency and redundant data movement through the memory hierarchy are fundamental bottlenecks.

Numerous proposals to improve the status quo in sparse data research have been made by parallelizing the application to maximize use of memory level parallelism (MLP) and parallel compute resources (CPU [85, 86, 87, 88, 89, 90, 72] and GPU [91, 92, 93, 94, 95, 72]). With intelligent and careful design, these techniques are by and large successful in hiding memory latency, although *they come at the cost of increased data movement, a significant fraction of which is often redundant*.

For instance, as representative examples, consider three recent SpGEMM implementations: a CPU-GPU performance portable framework [72], a GPU-only implementation [94], and an Intel KNL-based analysis [64]. It is known that the outer-product algorithm (OP) for SpGEMM is more efficient [65] than Gustavson's original row-wise algorithm (RW) [96] because RW does significantly less useful work per unit input data read compared to OP. Yet, all of these works (as well as a majority of the literature [64]) employ RW instead of OP simply because caches cannot hold the large number of temporaries (partial products) that

OP generates. Such favoring of redundant gather for optimized scatter is understandable as their focus is to maximize cache usage. However, these are not necessarily tailored towards DRAM operational inefficiencies such as row activation (**ACT**) and precharge (**PRE**) overheads.

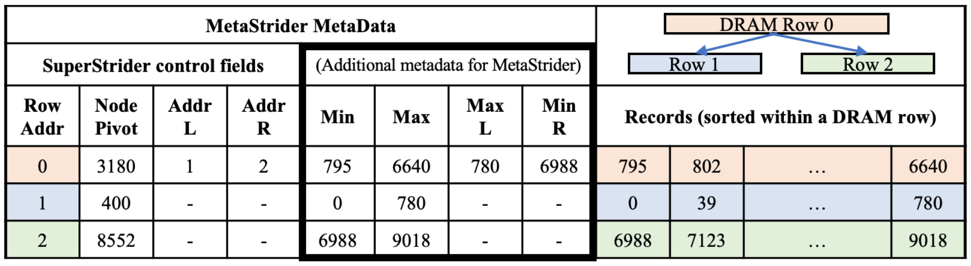
Proposals in the hardware accelerator space recognize that even large caches do not capture locality in such hyper-sparse data streams, and are often designed in a DRAM-centric manner. However, these either specifically target SpGEMM [97, 98, 65] or are not suited for handling incremental/streaming updates to sparse data (GraFBoost [62]), and therefore are incapable of accelerating applications such as Firehose. An exception to this is SuperStrider [24, 26, 25], although it still suffers from significant DRAM overheads despite being a memory-centric design.

This chapter presents SuperStrider and its successor, MetaStrider, both of which are based on vectorized binary search trees (Section 3.2).

SuperStrider employs a vectorized binary search tree in DRAM and a bitonic merge network near memory. To maximize useful bytes read per ACT (CAS per ACT), a DRAM row is treated as a first-class citizen and is the fundamental unit of operation. The row logically forms a node in the tree, where each node contains a vector of records sorted by key, as well as a pivot (key) and a pair of child pointers.

The tree is built in 2 phases. Phase 1 (`Addvec()`) sends records across a dynamically determined path of the tree from the root node to a leaf node while performing any possible reductions along that path. As a result, `Addvec()` results in possibly duplicate keys in different paths of the tree, that are yet to be reduced. Phase 2 (`Normalize()`) performs two depth-first-searches (DFS) on the entire tree to reduce and de-duplicate such records.

`Addvec()` is repeated multiple times to incrementally build the tree as new input that needs to be reduced is streamed in. `Normalize()` is performed once, at the end. As a result, SuperStrider returns significantly improved performance and energy efficiency when compared with prior software approaches.



MetaStrider MetaData								DRAM Row 0 → Row 1, Row 2			
SuperStrider control fields				(Additional metadata for MetaStrider)							
Row Addr	Node Pivot	Addr L	Addr R	Min	Max	Max L	Min R	Records (sorted within a DRAM row)			
0	3180	1	2	795	6640	780	6988	795	802	…	6640
1	400	-	-	0	780	-	-	0	39	…	780
2	8552	-	-	6988	9018	-	-	6988	7123	…	9018

Figure 3.1: Data layout of records and control fields/metadata in a DRAM-centric vectorized binary tree. The Metadata fields of MetaStrider are more expressive than the control fields of SuperStrider, and are decoupled from the kv-rows. This results in direct and indirect benefits as described qualitatively in Section 3.3.1 and quantitatively in Section 3.5. The values are omitted (in the Figure) from the kv-rows for brevity.

However, the energy overhead of `Normalize()` is equivalent to re-running `Addvec()` on the entire input stream again, a significantly excessive overhead. Given the promising nature of the memory-centric approach of SuperStrider, MetaStrider builds upon it, as is explained in the remainder of this chapter.

3.2 Vectorized Binary Search Trees

SuperStrider [26] introduces a novel, explicitly managed DRAM layout for sparse data to improve row locality, where each DRAM row forms a node in a vector binary tree. To manage the tree, control fields in the form of a pivot (key) and a pair of child pointers are needed at each node, i.e., for every block of K-sorted key-value pairs that constitute the node, as shown in Figure 3.1. **K=170** is used as the default in this work unless otherwise mentioned. This is derived assuming a 2 KB row [99] consisting of records whose keys are 64-bit and values are 32-bit (Section 2.2.3).

In general, two properties are desired of the tree:

1. *Property 1*: For each node, all records in its left subtree have keys smaller than the **pivot key** of the node, which, in turn is smaller than the keys of its right subtree.

2. *Property 2*: For each node, all keys in its left subtree are less than **all keys** of the node, which are in turn less than those of its right subtree.

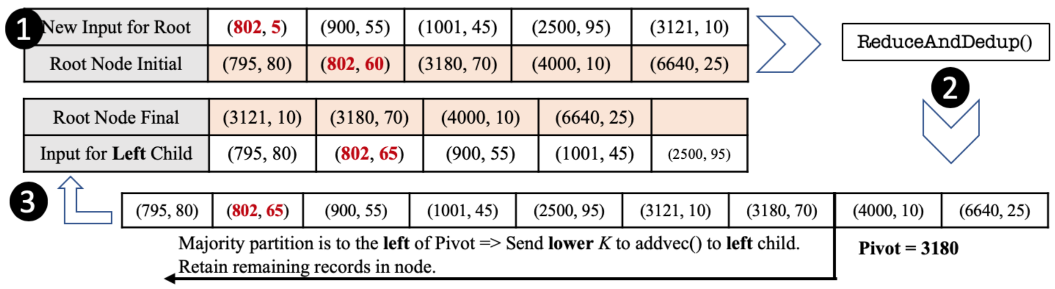


Figure 3.2: Example functionality of `Addvec()` with $K = 5$.
Step 1: Front-end supplies a pre-sorted 5-vector to insert and reduce.
Step 2: `ReduceAndDedup()` is applied on the root node, given the input. In this example, records with key=802 were reduced and de-duplicated.
Step 3: The pivot of the root node partitions the resultant vector, to determine which records need to be written back to the root, and which serve as input to recursively apply `ReduceAndDedup()` to one of the node's children. The relative direction of the larger partition determines the direction of recursion.

Property 1 is guaranteed to be true of the tree at any time including during incremental construction. *Property 2*, which is more strict, is guaranteed to be true for the final state of the tree, such that no duplicate keys exist.

The operational granularity being that of a DRAM row, the tree is built K records at a time. K pre-sorted records are added to the tree (and reduced and de-duplicated with existing records in the tree) via an operation known as `Addvec()`, which recursively applies a `ReduceAndDedup()` function along a path P of nodes (rows) of the tree, as shown in Figure 3.2. The `ReduceAndDedup` function takes two K-sorted vectors and generates a deduplicated vector of size at-most $2K$, post reduction. Depending on the pivot key at that node, the larger partition (of size K, in general) is sent as an input vector to be applied recursively to the left or right child of the node, and the smaller partition (of size $\leq K$, in general) is written back to the node. This acyclic path P traverses the tree starting at the root and ends in a leaf of the tree (potentially creating a new leaf), or when there are no records left as a result of `ReduceAndDedup()`, whichever is earlier. When a new leaf node is created, a constant pivot key is chosen for that node as the median of its associated records.

In traversing P, a key observation that enables SuperStrider performance is that `Addvec()` visits at most 1 node (row) per level in the tree while simultaneously per-

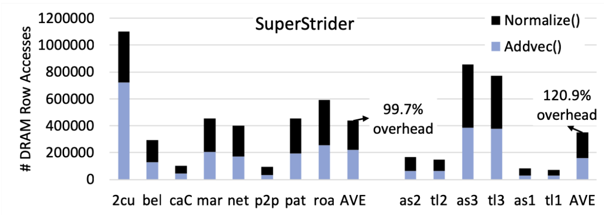


Figure 3.3: Overhead of `Normalize()` in SuperStrider is $>100\%$ on average, thus motivating an improved, energy-efficient design. (See Figures 3.11, 3.14 for a quick look at how much MetaStrider reduces this.)

forming the reduction operation and avoiding *random* accesses to DRAM. The downside, however, is that there may be duplicates along other paths of the tree, which have to be consolidated. A detailed example is available in the SuperStrider publications [26], but is omitted here for brevity. In the operation of `Addvec()`, *Property 1* is always maintained. As a result, duplicates along different paths of the tree can be reduced and deduplicated as a final post-processing step that fixes any nodes in violation of *Property 2*.

This step, known as `Normalize()`, consolidates the left and right subtrees of each node in a DFS manner. This is done by traversing the left (right) subtree, extracting the leaf that houses the maximum (minimum) key and performing an `Addvec()` at the parent node with the contents of the leaf as the input vector whenever *Property 2* above is violated. However, this tree traversal for `Normalize()` in SuperStrider is rather expensive in terms of energy, as quantified in Figure 3.3.

Furthermore, the latest version of SuperStrider employs a bitonic merge based systolic network to realize the `ReduceAndDedup()` functionality [25]. The control logic (and stride management) is such that the inputs to the network are always K-sorted, an invariant that is re-enforced by `Addvec()`. This means that `ReduceAndDedup()` can be done in O(K) steps rather than O(K lg K) steps that the bitonic merger would require.

Key design decisions based on analysis of the state-of-the-art are now summarized.

First, sparse streams have low locality of reference, rendering automatic caching ineffective. MetaStrider chooses to *bypass caches* to save energy.

Given that DRAM accesses are on the critical path of the application, maximizing row locality is key. The *vectorized binary tree* approach proposed by SuperStrider for optimizing data layout is a promising one, but it can be made better via improved Metadata and its management (Section 3.3.1).

DRAM access occurs in bursts (64-bit for DDR4 and 128-bit for HBM). `ReduceAndDedup()` is the fundamental compute operation in MetaStrider, and is implemented using a *light-weight merge network* at the memory controller (Section 3.3.2). Designing the merge network such that it incrementally constructs the output by consuming the input in this manner would natively hide merge network latency without additional mechanisms such as write buffer, row re-map memory, etc. [26], as described in Section 3.3.2.

Given that the merge network is small, it can be replicated in order to enable *parallel operations on the tree* (Section 3.4) in order to leverage MLP. Finally, two variants of MetaStrider are considered:

- One that uses the memory-centric design without any extra hardware, at the cost of potentially increased energy consumption due to traffic on the system interconnect. In such a design, `ReduceAndDedup()` is performed by the front-end core instead of dedicated hardware.
- One that uses dedicated logic for merging "near" memory, at the DRAM controller. Such near data processing (*NDP*) not only reduces data movement but also enables an even tighter logic-memory integration for future variants as technologies evolve in the third dimension [100]. Therefore, unless otherwise mentioned, this work assumes the latter variant (and not the non-NDP variant) to be the default for MetaStrider.

3.3 MetaStrider

3.3.1 Metadata

Recall from Section 2.3.2 that SuperStrider's `Normalize()` is rather inefficient. Careful analysis reveals the reason for this as redundant DRAM row activations in traversing down a given subtree in determining whether or not *Property 2* is violated at the root node of the subtree. As a solution, MetaStrider chooses to "memoize" certain range information such that testing for the property can be done without pointer chasing. If the range information indicates that *Property 2* is not violated, then no further actions are necessary (meaning no DRAM ACT) at that node, and the DFS super-step can continue. Typically, it is found that this is the majority scenario, which means that such extra "metadata" is worth the extra spatial overhead (a classic speed vs space tradeoff).

Table 3.1: Description of MetaStrider's metadata fields.

MetaStrider "metadata"					
SuperStrider "control fields"			*Additional fields unique to MetaStrider*		
Field	**Description**	**Bytes**	**Field**	**Description**	**Bytes**
RowAddr	DRAM row ID of node	2	Min	Min key of node	8
Pivot	Pivot key associated with node	8	Max	Max key of node	8
Addr L	DRAM row ID of left child	2	Max L	Max key of left subtree	8
Addr R	DRAM row ID of right child	2	Min R	Min key of right subtree	8

Table 3.1 describes the resultant Metadata fields with numerical examples apparent in Figure 3.1. Observe that *Property 2* can now be translated as follows: maxL $<$ min $<$ max $<$ minR for any node.

To avoid confusion, the terminology of Metadata is unique to MetaStrider, and "control fields" are used to describe the equivalent for SuperStrider. Since SuperStrider's `Normalize()` is now updated to benefit from these Metadata as described above, it is called `GlobalReduce()` in the context of MetaStrider.

The **primary advantage**, a direct one, therefore stems from the introduction of these extra fields. The overhead of DRAM ACT is paid only when absolutely necessary.

The **secondary advantage**, an indirect one, stems from decoupling the Metadata from the *kv*-rows into a separate Metadata Store. This helps in improving structural properties of the tree such as density and worst-case guarantees. For example, with AVL tree balancing [101][1], the tree is guaranteed to be dense irrespective of nature of sparse input, and

[1]An Adelson-Velsky and Landis (AVL) self-balancing binary search tree ensures that the heights of the two child subtrees of any nodes differ by at most one. In the context of MetaStrider's vectorized binary tree, AVL balancing is applied on the pivots of the nodes, without moving actual row contents.

such balancing can be done efficiently on the Metadata Store alone (without disturbing the *kv*-rows). It is found that decoupling and AVL balancing alone reduce the average number of DRAM ACTs by over 20% when compared to an implementation where all the Metadata fields above are present but are not decoupled. Formally, the number of ACTs per call to `Addvec()`, irrespective of input, is bounded by the number of levels of the tree, which in turn, thanks to the AVL property, is bound to lg N, where $N = \frac{nnz}{K}$ is the number of nodes in the tree.

Such decoupling also enables other downstream tree-optimizations such as partitioning, fanout, etc. (Section 3.4) to be made without having to re-build the system from scratch.

The **overhead** due to the Metadata Store is a small fraction of the data being stored. From Table 3.1, each node needs 46 bytes' worth of Metadata, 32 bytes of which are key-range information. For a 2 KB wide row being treated as a node, the overhead is therefore just about 2%. This overhead can be reduced via partial omission or compression of fields, the details of which are beyond the scope of this work. These mechanisms may be designed based on the following insights: (a) MetaStrider performance is not very sensitive to average access times of the Metadata Store (Section 3.5), (b) key-range information accounts for a significant fraction of the Metadata overhead (70%), (c) ranges are useful only during `GlobalReduce()`, (d) lower levels of the tree are less likely to be accessed, and (e) range re-computation can replace memoization for nodes with shallow sub-tree depths.

3.3.2 Architecture and Integration

Figure 3.4 presents, at a high-level, the micro-architecture that MetaStrider implements. Recall from Section 3.2 that the non-NDP variant utilizes the CPU to perform the functionality of the NDP unit. It can be deployed using off-the-shelf devices given the following two capabilities: ability to bypass caches, and ability to control row management (possibly via partial knowledge of row address interleaving). The remainder of this section focuses on describing the components of the NDP unit and then discussing techniques of integrating

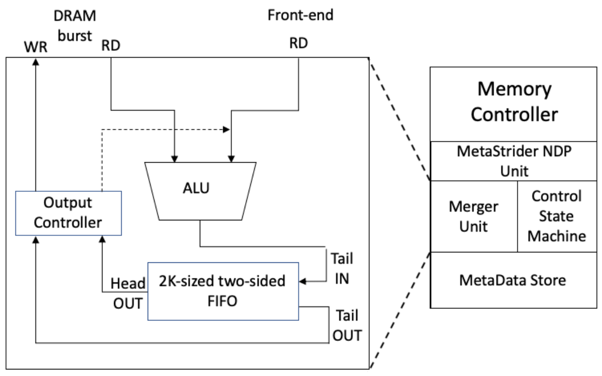


Figure 3.4: A schematic of MetaStrider microarchitecture, implemented as part of the memory controller. In this work, the memory controller is designed to be able to access both, system memory and MetaStrider memory (pre-segmented at the beginning of the program using the `Init()` API).

MetaStrider with conventional system software.

Merger Unit is responsible for performing `ReduceAndDedup()`, which is the recurrent sub-step used in `Addvec()`. It reads records in bursts from DRAM on one side and the front-end on the other. The ALU then compares the keys and performs reduction on their values if necessary. If no reduction is performed (i.e., distinct keys are input from both sides), then the record with the higher key is retained in the ALU input register for subsequent comparison, and that with the lower key is pushed to the tail of the output FIFO. This process is repeated until K records are read from each side. As described in Section 2.3.2, the output controller then sends the first/last few ($\leq K$) records (depending upon the pivot) back to the open DRAM row, and retains the remaining (typically K) records in the FIFO. These latter records are then internally streamed to the ALU for the next sub-step when records from one of the child DRAM rows are streamed-in.

For example, in the example from Figure 3.2, the first record read from DRAM and the input, respectively contain the keys 795 and 802. The ALU subsequently retains 802 and pushes 795 to the tail of the FIFO. Next, the record with key 802 is read from DRAM, and the ALU detects a collision and accordingly updates the associated value. Then, the record with key 900 is read from the input, causing the ALU to retain 900 and push the record with key 802 and its updated value to the FIFO. This process is repeated until both, the

DRAM row and the input batch are read, and this results in the FIFO containing 9 records in this example. As the output controller finds that a majority of records are less than the pivot, it retains the lower 5 records in the FIFO to stream as the input batch for the next `ReduceAndDedup()` operation (on the left child). The higher 4 records are written back to the open DRAM row.

Note that this design, unlike SuperStrider's bitonic network, interleaves logic (ALU) and memory operations at the granularity of a DRAM burst. Furthermore, the complexity is linear in K, as opposed to log-linear (SuperStrider).

Metadata Store is akin to an SRAM cache in utility and dimensions, although it is explicitly managed as a scratchpad. Ideally, it would be housed on-chip with the memory-controller for fast access. Alternately, the LLC, which is largely unused for MetaStrider, can be used for this purpose. However, either of these strategies may not be possible when the size of the tree grows such that the total Metadata overhead is over several MB. In such a scenario, realizing the following key insight is helpful: the first few levels of the tree (and therefore their Metadata) are significantly more likely to be accessed frequently. Therefore, it would be advisable to "cache" these on-chip, and spill the rest off-chip, perhaps into system memory. A detailed analysis of off-chip Metadata Store is beyond the scope of this work. However, Section 3.5.4 demonstrates that nominal values of average access latencies have an insignificant impact ($< 0.5\%$) on overall performance.

Control State Machine. The control logic is able to issue load and store commands to the Metadata Store and to the memory controller, in addition to Merger Unit data routing. These functions are well within the capabilities of modern, smart memory controllers that perform sophisticated scheduling, queue coalescing, address interleaving, idle row closure decision making, etc. [102, 103].

API. Table 3.2 summarizes a simple, offload-based blocking API to integrate with the MetaStrider engine. Note that the front-end producer of sparse data may be a general purpose CPU, accelerator, external flash device [62] or network I/O [49]. Without loss of

Table 3.2: MetaStrider Application Programming Interface (API)

API	Comment
Init (sz, n, f)	MetaStrider configures its K, MLP-strategy (Section 3.4), reserves corresponding segment(s) of memory.
EnqReduce (k, v)	Each core sends to memory K sorted records one by one.
Addvec()	Each core signals Addvec() on its tree after K EnqReduce() or end of input, whichever is earlier.
GlobalReduce()	Each core signals end of input.
Lookup(key)	Lookup the value of key or initiate in-order traversal.

generality, "core" is used to abstract these producers of sparse data. Sufficient memory is assumed available to house all records.

The most straightforward use-case scenario is where the programmer (or in certain cases, the compiler) identifies a sparse data stream to be reduced, and accordingly initializes MetaStrider with the size of records, number of cores supplying packed sparse data in parallel, and the desired reduction operator. MetaStrider accordingly configures itself based on available resources, reserves corresponding segment(s) of memory and returns the value of K to the program. Each core then scatters the gathered sparse data to be reduced by first pre-sorting these, K at a time, and then dispatching them in-order via `EnqReduce()`. Recall from Section 2.3.2 that this pre-sorting is a necessary requirement for scalable `Addvec()` functionality. Note that standard sparse input formats (see below) are pre-sorted by themselves. If the application is such that it is not possible to obtain a pre-sorted input, an incrementally sorted paradigm such as a priority queue may be used while K records are gathered and buffered at the front-end. Even if it is decided to perform a $K\lg K$ sort as a separate pre-processing step, our analysis in Section 3.5.5 reveals that the core frequency required for this to be in parallel with a preceding `Addvec()` is modest ($<$ 1GHz) for typical values of K (recall that `Addvec()` typically involves reading and writing multiple DRAM rows in succession, and is therefore a relatively longer latency operation).

After K such `EnqReduce()` calls, an `Addvec()` call signals the MetaStrider controller that the end of this batch (of size K or end of input) is reached. This process is repeated until the input is consumed. `GlobalReduce()` is then called to ensure the entire

```
// A and B may be stored in conventional sparse formats or in the MetaStrider format.
SpGEMM_OuterProduct (Matrix A, Matrix B):
  // Initialize MetaStrider memory.
  K = MetaStrider.Init(12, 1, "+")
  colA = A.readNextNonZeroColumn()
  rowB = B.readNextNonZeroRow()
  // Assume that the column index of colA and the row index of rowB are identical (k). If
      not, advance accordingly until they are (not shown).
  for (i, k, A_ik) in colA:
    for (k, j, B_kj) in rowB:
      key = makeKey(i, j)
      value = A_ik * B_kj
      sortedQueue.pushAndSort((key, value))
      if (sortedQueue.size() == K or EOF):
        // Stream records for Addvec().
        while (sortedQueue.size() > 0):
          MetaStrider.EnqReduce(sortedQueue.popMin())
        MetaStrider.Addvec()
  // Repeat lines 5-17 until A or B consumed. Then:
  MetaStrider.GlobalReduce()
  // Perform gather for downstream operations as needed.
  MetaStrider.Lookup()
```

Figure 3.5: Demonstration of MetaStrider API

dataset is reduced and de-duplicated, thereby priming the data for downstream operations.

For clarity, the above is also explained via a specific example in the form of how the MetaStrider API may be used in the context of a single-core SpGEMM, as shown in Figure 3.5.

Conversion to/from other representations. Commonly used linear sparse formats include ordered-coordinate, (doubly-)compressed-sparse-row (D)CSR [59, 66, 67, 68, 69]. For hyper-sparse matrices such as those of interest in this work, CSR offers little benefit over ordered-coordinate as there are few non-zeros per row. DCSR offers benefits only when a group of consecutive rows are all zero. Thus, ordered-coordinate (typically stored as arrays of keys and values) are most commonly used in this space [104]. All of these representations have the disadvantage of not being able to support dynamic updates efficiently as they fundamentally require insertion into the middle of an array. Nevertheless, MetaStrider operation is independent of and is compatible with inputs in any of these formats. Note that the MetaStrider representation most closely resembles ordered-coordinate with DRAM-friendly enhancements.

Conversion from these formats into MetaStrider format is a direct application of the API when the original input is streamed record-by-record into the MetaStrider engine. Thus, no explicit conversion is required as conversion can be combined (implicitly) with downstream operations.

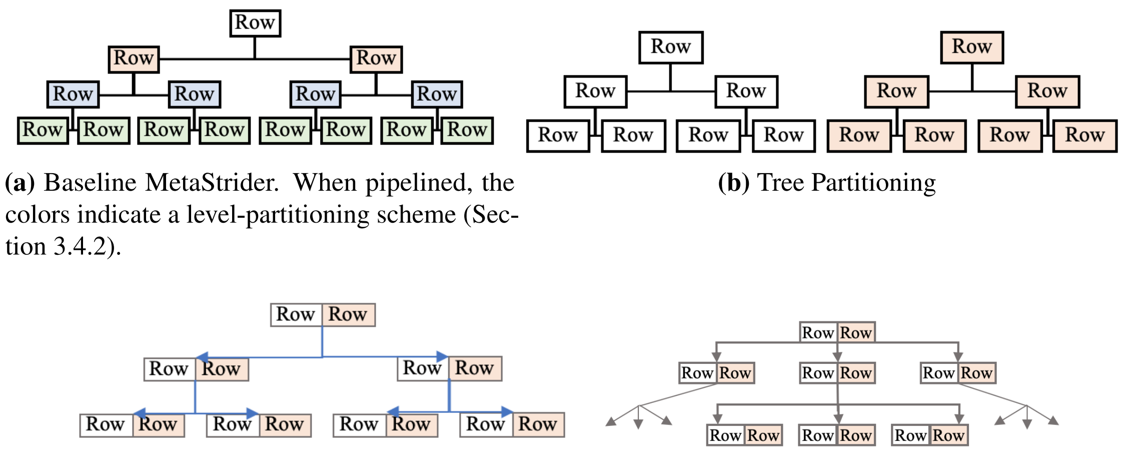
Converting back to a conventional linear ordered format from the MetaStrider format can be achieved easily by reading out the records back from the tree via an in-order DFS traversal. However, further operations on MetaStrider data do not require conversion to a traditional format as long as the operations are associative. In other words, downstream algorithms can stream in data from the MetaStrider tree in a DFS or BFS manner directly without having to store an intermediate linear matrix format into memory. Furthermore, several hash-based applications do not even require ordered input data [64]. As such, it is envisioned that such explicit write-out to occur only once all processing on data is complete, and only if necessary for backward compatibility.

Explicit or implicit conversion to/from these formats can thus be done on the fly, in parallel with downstream operations. Therefore, there is negligible impact on performance due to conversion.

Lookup. Finally, one can extract the value associated with key κ via `Lookup(`κ`)`, where the Metadata is consulted to perform a binary search over the nodes of the tree (using the pivots) to locate the node whose key-range includes κ. If such a node is found, the corresponding DRAM row is opened. Then, a binary search within the row is performed to locate κ as the records in the row are already sorted.

3.4 Leveraging Memory Level Parallelism (MLP)

So far, this chapter has focussed on MetaStrider from the perspective of a single bank of memory (MLP=1). Modern memory systems have significant MLP available via bank, rank and channel level parallelism. For example, HBM has 8 channels, each of which comprises 8 banks, resulting in an MLP of 64. This section explores several mechanisms for parallelizing a MetaStrider tree to leverage available MLP and obtain improved performance (assuming



(a) Baseline MetaStrider. When pipelined, the colors indicate a level-partitioning scheme (Section 3.4.2).

(b) Tree Partitioning

(c) Node Grouping.

(d) Increasing Fanout.

Figure 3.6: Four novel techniques of vectorized binary tree management to leverage MLP. Different colors indicate different banks of memory that can be accessed in parallel.

the input rate (front-end) can keep up).

Figure 3.6a depicts the baseline MetaStrider tree (N nodes and lg N levels) sprawling a single bank of DRAM, as well as parallel approaches described below.

This section focuses on `Addvec()` and not on `GlobalReduce()`. This is because our Metadata improvements (Section 3.3.1) have significantly reduced the overhead of the latter (Section 3.5.4) such that `Addvec()` is now the most critical step. Furthermore, parallelizing `GlobalReduce()` can benefit from years of parallel DFS research [105] and is therefore not necessarily a novel contribution of this work in itself.

Note that with the exception of Tree Partitioning approach, none of the other three approaches require the programmer or compiler to make any code changes. If a power user wishes to override the default strategy, it is sufficient to simply extend the `Init()` API to specify a bit-mask to indicate which mode(s) to deploy.

3.4.1 Tree Partitioning

The single tree can be spatially partitioned by key into T trees, each containing $\frac{N}{T}$ nodes and lg $\frac{N}{T}$ levels. This enables the trees to function independently and in parallel (Figure 3.6b shows $T = 2$). Furthermore, the unit of work for each tree is reduced as they each have fewer nodes and levels are significantly reduced when compared to the baseline MetaStrider tree. For these benefits to fully manifest, the following are desirable:

Load balancing. For shortest critical path, all the trees should have approximately an equal number of records to reduce. A round-robin distribution of keys (tree t gets keys such that $key\%T == t$, where $\%$ is the modulo operator) is a simple, yet effective approach at this. Load balancing can be further improved via hashing, provided the hash is perfect and preferably minimum and uniformly-distributing.

For example, it is known that the Residue Number System (RNS) [106] can help distribute contiguous data to a wider range [31, 30, 29]. Such a hash function would be computed by concatenating the set of residues (moduli) $R(\kappa)$ of key κ against a pre-determined set of co-prime bases B such that $R(\kappa) = \{\kappa\%b, b \in B\}$. For keys generated by indexing into a matrix of at most $100M \times 100M$ dimensions, it can be shown that $B = \{45, 46, 47, 49, 53, 59, 61, 67\}$, $B = \{2049, 2050, 2051, 2053\}$ or $B = \{4194305, 4194306\}$ for $T = 8, 4, 2$ respectively, are suitable for such RNS hashing. Other similar hash functions that help extract MLP in modern systems, such as XOR-based hashes [107, 108] may also help improve load balancing. Finally, note that any such record-to-tree assignment is to be applied by the front-end prior to the call to `EnqReduce()` in order to satisfy the pre-sorted input condition. The evaluation therefore is bounded by $T = 8$ so as to limit the amount of intra-front-end state communication.

Parallel NDP and front-end hardware. The front-end should be capable of feeding all trees in parallel for maximum benefit. In other words, the front-end should be able to issue T `Addvec()` calls in parallel. Next, the control logic, Merger Unit and Metadata Store need to be replicated at some level. Note that the former two are relatively light-weight by design and can therefore be replicated.

The Metadata Store need not increase in size as the total number of nodes (hence total Metadata) doesn't change with tree partitioning. For example, if the unpartitioned tree has 100 nodes, then with $T = 4$ partitions, each partition would nominally have 25 nodes (resulting in each partition being a shallower/faster tree). However, it is desirable to implement the Metadata Store as a multi-banked store for parallel access.

API usage. The programmer or compiler must perform the following for updating T trees in parallel, in order to benefit from the tree partitioning strategy:

1. Specify $n = T$ in `Init()`.
2. Create T threads, each with a thread-private version of `sortedQueue` (*cf.* Listing 1).
3. Each thread produces records, applies the load-balancing hash to its keys and then sends to appropriate thread.
4. Each thread then calls `EnqReduce()`, `Addvec()` and `GlobalReduce()`.

3.4.2 Pipelining

Given sufficient *resources*, pipelining is a compelling mechanism of extracting parallelism from a stream of independent *instructions*. For pipelining MetaStrider, *resource* is equivalent to MLP and *instructions* are equivalent to recurrent calls to `Addvec()`. Recall that each such call typically traverses several levels of the tree, accessing exactly one unique DRAM row (assume $F = 2$) per level (hence, no data stalls are possible). Therefore, given sufficient MLP (read no resource stalls), consecutive calls to `Addvec()` can occur at every time step (τ) as shown in Figure 3.7, where τ is the time required to perform `ReduceAndDedup()` on a row. This is similar to deep-pipelining or sub-pipelining the function units in the execute stage of a traditional processor.

Resource allocation. The intuition of our approach to minimize resource stalls stems from the following insight: assigning each pair of consecutive levels of the tree to different memory banks eliminates resource stalls for a (sub)pipeline depth of 2. It can then be seen that, for a tree of depth $D \leq MLP$, assigning each level to a different bank (level-partitioning) enables pipelines of depth MLP (Figure 3.6a shows $D = MLP = 4$).

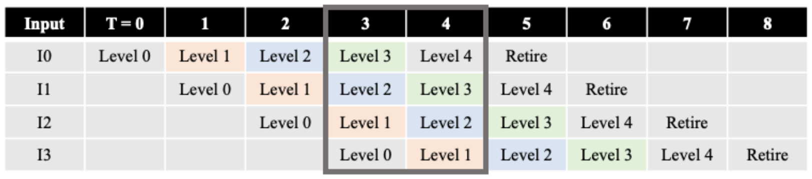
However, beyond a sufficiently deep level $D_{thres} \leq D$, the number of rows in that level may exceed the size of a bank. For example, with a 128 MB (64K x 2 KB rows) bank, $D_{thres} = 16$. To account for bank capacity, this level-partitioning approach is applied only to the first $D_{thres} - 1$ levels of the tree, and subsequent levels $d \geq D_{thres}$ are each sliced equally across the banks, resulting in batches of $\frac{2^d}{MLP}$ rows of level d per bank.

Load balancing. When $D > MLP$, the above level-partitioning mechanism requires further thought for levels at depth d, where $MLP \leq d < D_{thres}$. In other words, a relatively small MLP would result in increased resource stalls in the pipeline if the load distribution to the banks is not uniform. A simple mechanism to address this is to assign such levels to different banks in a round-robin (RR) manner. Note two properties of the tree, as one goes down the tree: (i) the number of rows in each level doubles and (ii) the probability of a level being accessed decreases. RR is not cognizant of either of these properties, thus begetting an improved heuristic.

If D were known apriori, levels 0 and D would then be mapped to bank 0, levels 1 and $D - 1$ to bank 1, and so on. Such an allocation factors in both of these properties above to realize a balance that is as close to the ideal as possible. However, as the size and percentage of repeated keys of the input is not known, D is not static and hence cannot guide resource allocation. Therefore, an incremental variant of this heuristic is proposed, where levels are assigned in a ping-pong (PP) manner such that level d maps to bank ($d\%MLP$) if $d\%(2 \times MLP) \leq MLP$, else, it maps to ($MLP - d\%MLP$).

More complicated approaches that are not level-partitioned are possible when this is expressed as a graph coloring problem. However, according to our analysis, a simple heuristic such as PP is able to leverage about 80% of available MLP on average (and reduces resource stalls by over 35% when compared to RR). Yet other designs may choose to literally cache the "rows" of the first few levels of the tree that tend to be accessed most frequently, although those approaches may not scale easily with increased number of trees.

Supporting hardware. Explicit pipeline buffers are not necessary as the DRAM rows



Input	T = 0	1	2	3	4	5	6	7	8
I0	Level 0	Level 1	Level 2	Level 3	Level 4	Retire			
I1		Level 0	Level 1	Level 2	Level 3	Level 4	Retire		
I2			Level 0	Level 1	Level 2	Level 3	Level 4	Retire	
I3				Level 0	Level 1	Level 2	Level 3	Level 4	Retire

Figure 3.7: The "EX" stage can be sub-pipelined to leverage MLP.

implicitly serve as buffers. The Merger Unit needs to be replicated in order to take advantage of MLP, similar to Section 3.4.1. The Metadata Store capacity remains unchanged but requires parallel access in order to serve multiple inputs simultaneously. The front-end still produces a single input to `Addvec()` at a time, its rate being determined by pipeline depth (MLP).

3.4.3 Grouping

Yet another mechanism of leveraging multiple banks is to group two rows of the same index across two banks into the same node (Figure 3.6c). The motivation is that it would increase K logically, thus improving algorithmic efficiency because of denser trees and higher probability of parallel reduction. For example, say $K = 170$ records fit in a row. By grouping such rows across 4 banks and associating them with a single tree node, $K = 680$ is realized.

This results not only in trees with reduced depth, but also in lower Metadata overhead thanks to the reduced number of nodes. However, grouping increases the load on front-end as it now has to pre-sort gathered data in batches of 680 records rather than 170 records before sending it across for scatter, although it reduces the Metadata Store overhead by $4\times$. Also, MLP is not utilized within a tree if the same Merger Unit as baseline MetaStrider is used. If MLP has to be utilized, a log-hierarchical network similar to SuperStrider's bitonic merger is better suited, although it does not fully support fine-grained logic-memory overlap.

3.4.4 Fanout

The arity/fanout (F) of a binary tree can be increased to $F > 2$ to decrease the number of levels of the tree to $\log_F N$ (Figure 3.6d shows $F = 3$), thereby facilitating faster MetaStrider operation. However, it then becomes necessary to increase the number of DRAM rows associated with any given node to $F - 1$ in order to maintain the granularity of operation to be that of a DRAM row (recall from Section 2.3.2 that row-granularity operations are fundamental to increasing row locality in a vectorized tree design).

As an example of contradiction, assume each node in a 3-ary tree is still associated with a single DRAM row. A 3-ary node would by definition have 2 pivots instead of 1 in order to result in 3 partitions corresponding to its 3 children. During `ReduceAndDedup()`, consider the task of partitioning (at most) $2K$ records in the output buffer of the Merger Unit into 3 partitions. Depending on the nature of the 2 pivots, it cannot be guaranteed that at least one partition has at least K records in it. Therefore, it is not possible to propagate any single partition to a child without violating the row-granularity of operations. Propagating $< K$ records to a child would result in significantly diminishing returns (increasingly lower useful bytes accessed per subsequent ACT) as one traverses down the tree. On the other hand, propagating two partitions down the same direction would violate *Property 1* (when generalized to $F \geq 2$, see below). Instead, if 2 DRAM rows are associated with a 3-ary node, then the task of partitioning would involve $(2K + K = 3K)$ records. It can then be *guaranteed* that at least one partition has at least K records.

The conclusion is that $MLP \geq F - 1$ is necessary to retain the row-granularity of operations. Increasing F is, in essence, another way of leveraging MLP. Note, increasing fanout can leverage MLP *without* requiring a parallel front-end.

The Merger Unit architecture is similar to the $F = 2$ case, with the output buffer being of size $F \times K$ and the control logic being updated to support F partitions.

Changes to the algorithm. One may gain intuition for the general working of updated `Addvec()` and `GlobalReduce()` based on the text above and on the generalized version of *Property1, 2* (Section 2.3.2) below, for each F-ary node:

1. *Property 1*: All records in its f^{th} subtree have keys smaller than the f^{th} **pivot key** of the node, which is in turn smaller than the keys of its $(f+1)^{th}$ subtree, where $0 \leq f < F$.

2. *Property 2*: All keys in its f^{th} subtree are less than **all keys** of the node, which are in turn less than those of its $(f+1)^{th}$ subtree, where $0 \leq f < F$.

Impact on Metadata. An F-ary node requires storage of $(F-1)$ pivots, addresses of its (F) children, and key-ranges of each of the following: (F) partitions of the node, $(F-1)$ rows associated with the node and (F) subtrees of the node (some of these metadata overlap and are rendered redundant when $F = 2$). As a result, when compared to the per node Metadata size in bytes of $F = 2$ (46 bytes: Section 3.3.1), an F-ary node's Metadata experiences an increase by a factor of about $0.7F$ where $F > 2$. However, note that upon increasing F, the number of nodes reduces because each node now consists of upto $(F-1)K$ records. As a result, the total overhead of Metadata in bytes, when compared to $F = 2$, is an input-dependent factor α, where $\alpha \in [\frac{0.7F}{F-1}, 0.7F]$ and $F > 2$.

Summary of Section 3.4. In this section, 4 orthogonal approaches to leveraging MLP are proposed, all of which can be used in conjunction with each other. This results in a tradeoff-rich design space, given hardware constraints and availability. Based on recent industry trends, the preference of these are as follows: Partitioning > Pipelining > Grouping > Fanout. In particular, a combination of Partitioning and Pipelining is found to be most practical and efficient. A more quantitative treatment is available in Section 3.5.5.

3.5 Experimental Results

3.5.1 Evaluation Methodology

Recall from Section 2.2.3, the focus of this work is to tackle main memory latency for sparse data. As such, the first-round of simulations deploy a single channel of 128MB of HBM main memory with a relatively liberal 4MB of LLC to favor the software reducers

Table 3.3: Simulation infrastructure

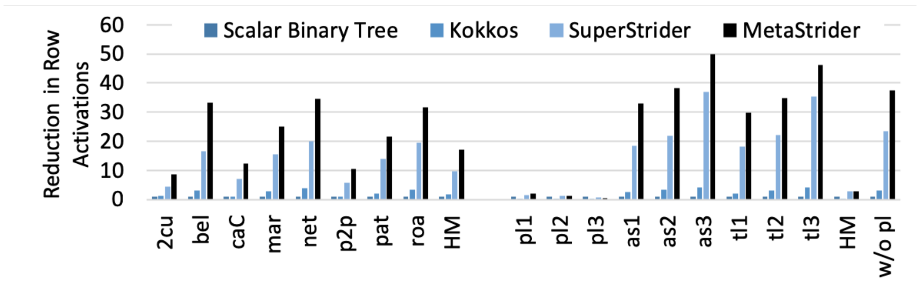
	Section	Tool
Baseline, kokkos	2.3.1	Unmodified gem5 [109]
MetaStrider	3.3.2	gem5 + cache bypass
SuperStrider	2.3.2	+ dedicated accelerator memory
GraFBoost	2.3.2	+ hardware at memory controller
Rapid design space exploration	3.4	Cycle-accurate, in-house simulator using HBM timing parameters [99]
Power, area	Logic	ALU [110, 111]
	Memory	SRAM-CACTI [112], DRAM [113]
(22nm models)	System Interconnect	OpenSMART [114]

conservatively. The software baselines benefit from bank-level parallelism (8 banks in a channel), whereas MetaStrider conservatively does not leverage this in these first round of simulations. As is standard practice, an FR-FCFS memory scheduler, and an open-adaptive row management policy are used. Although the reducers in this work (including MetaStrider) would perform with any DRAM without significant changes in efficiency, HBM is chosen because its stacked architecture naturally lends scalability to near-data processing (NDP), allowing for easier comparisons in future works.

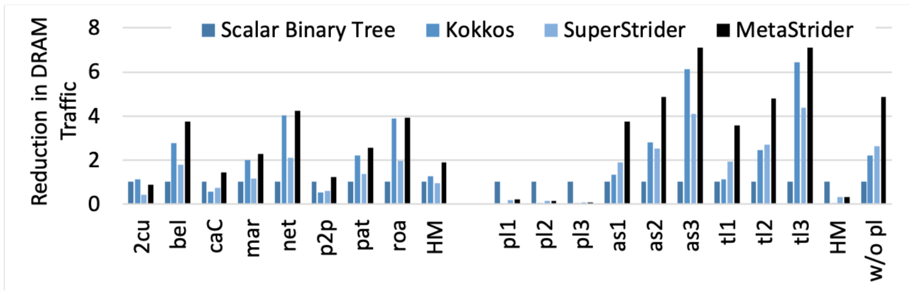
Since the workloads have low compute intensity, the LLC and DRAM are driven with an inorder core (with 8-way 32KB L1 I/D caches). The system is simulated using the gem5 full system simulator. In particular, MetaStrider is implemented as a separate MemObject connected to the system interconnect. MetaStrider API is implemented using gem5's pseudo-instructions at the front-end.

A second round of simulations are designed to evaluate the scalability of MetaStrider with available MLP. For simulation speed, a cycle-accurate, in-house simulator is used for this purpose.

A summary of the simulation infrastructure is in Table 3.3. This work makes an effort to compare MetaStrider against other state of the art reducers (Sections 2.3.1, 2.3.2) across various domains (CPU, GPU, accelerator, out-of-core) to provide reasonable insights. For fair comparison, this work re-implements these original works using the infrastructure above to better suit them for sparse associative reduction, making assumptions in their favor where



(a) Reduction in DRAM ACTs.



(b) Reduction in total bytes read from DRAM.

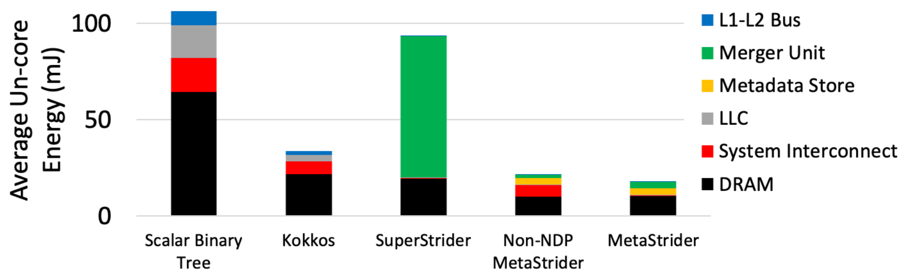
Figure 3.8: MetaStrider significantly reduces the number of activates as well as DRAM traffic, when compared to the baseline (higher reduction is better), thanks to its memory-centric design and intelligent Metadata.

possible (for example, ignoring the overhead of SuperStrider's control fields and row-remap memory). Finally, to compare against GraFBoost, which requires all the partial products to be available in memory apriori, a "batch" mode is also implemented.

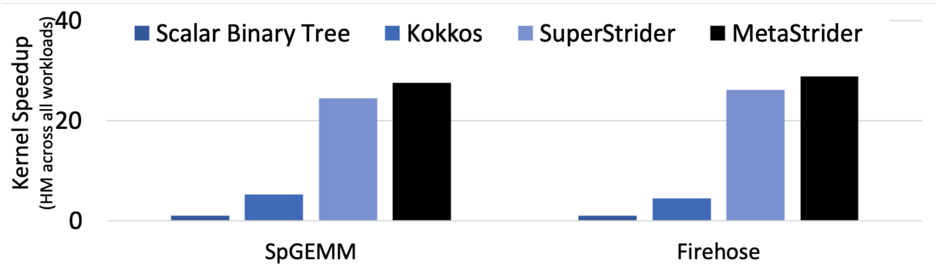
3.5.2 DRAM Performance

Row activation. Figure 3.8a shows a significant reduction in ACTs (and subsequent PRE). An average reduction of over $17\times$ and $37\times$ is seen for kernels in SpGEMM and Firehose respectively when compared to the baseline. Furthermore, these are $1.8\times$ and $1.6\times$ better than SuperStrider thanks to MetaStrider's improved Metadata capability.

Row hit rate and CAS per ACT. Bytes read per ACT of a 2 KB DRAM row is close to 64 for the baseline because the default read-out is the cache line size. Kokkos significantly improves locality characteristics, reading over 300 bytes per ACT, for a row hit rate of close to 80%. MetaStrider and SuperStrider achieve a row hit rate in excess of 95% and bytes per



(a) A breakdown of average un-core energy.



(b) Overall kernel speedup.

Figure 3.9: The most energy efficient reducer is the NDP variant of MetaStrider, realizing 5.3× energy savings when compared with the next best reducer, upon averaging across all workloads. Furthermore, it does so with a 11% performance improvement.

ACT of over 2000.

DRAM bytes read. As a result of the above observations, MetaStrider reduces DRAM pressure by 1.9× and 4.9× for SpGEMM and Firehose respectively when compared with the baseline, as shown in Figure 3.8b. These are significantly superior when compared with the other reducers (both hardware and software). In fact, SuperStrider ends up reading more bytes from DRAM on average for SpGEMM workloads on average when compared to the baseline because of its row-heavy operations, especially during its `Normalize()` phase.

3.5.3 Full System Results

Energy Savings. A summary of un-core kernel energy savings due to MetaStrider is depicted in Figure 5.5, when averaged across all workloads. (For fairness, for the non-NDP MetaStrider variant, the front-end core's contribution to `ReduceAndDedup()` is viewed as un-core energy, and is denoted by "Merger Unit"). Clearly seen, when compared to the baseline, MetaStrider reduces energy consumption significantly for all system components.

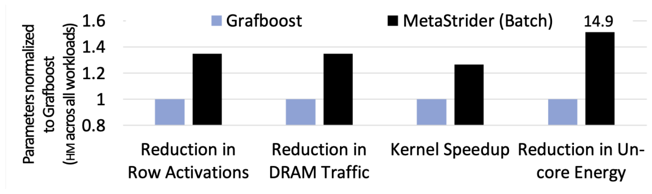


Figure 3.10: When all of the input kv-pairs are already available in memory, MetaStrider reduces un-core energy consumption by almost 15× and is 30% faster than the state-of-the-art approach for such merging because of its superior handling of DRAM rows.

The added overheads due to the Merger Unit and Metadata Store are dwarfed by these savings. Even when compared to SuperStrider, there is significant improvement in energy savings due to DRAM and the Merger Unit.

Area Overhead. The Metadata Store requires 7.74 mm^2 for 3MB of Metadata required, which can be significantly reduced if techniques such as grouping (Section 3.4.3) or compression (Section 3.3.1) are used. Furthermore, the store can also use existing LLC-SRAM or system DRAM if a dedicated budget is not available, as described in Section 3.3.2. The Merger Unit requires a mere 0.06 mm^2, which, when compared to an HBM die size of 40-100 mm^2 [115, 116, 117], is a very small fraction. Therefore, techniques that leverage MLP (Section 3.4) that may require some form of replication of the Merger Unit can also be easily realized on the same die. Note that this is significantly less resource intensive when compared with SuperStrider's bitonic merge network (0.24 mm^2) and GraFBoost's hierarchical merge tree (2.36 mm^2).

Kernel Speedup. Figure 3.9b depicts the kernel speedup over the baseline when averaged across all the workloads. The relative trend among the configurations can be explained easily based on the detailed breakdown presented thus far. For SpGEMM workloads, MetaStrider approaches the Amdahl limit with a deficit of less than 8%. Note that in the case of the Firehose benchmark set, the kernel *is* the application.

Batch Mode. When all the input kv-pairs are already available in memory, i.e., when the input is no longer an incremental stream of records, the state-of-the-art approach for such merging is GraFBoost. Note that while such batch-mode input may be possible for

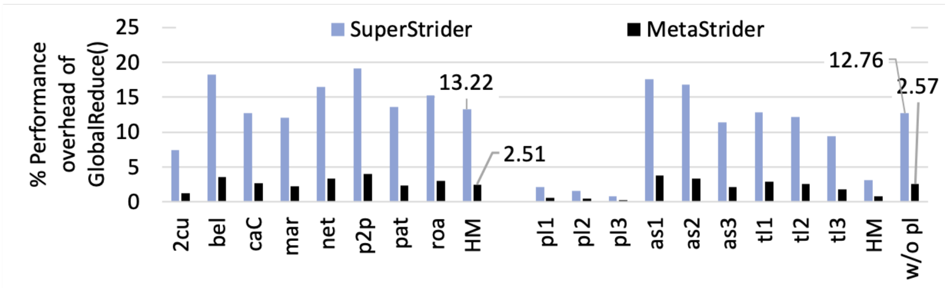


Figure 3.11: The % performance overhead incurred due to `GlobalReduce()` (lower is better) is just over 2% for MetaStrider, when averaged across all workloads.

SpGEMM workloads, it is infeasible for other incremental / streaming applications such as Firehose, for which only MetaStrider would be applicable. Figure 3.10 shows that in batch mode, MetaStrider reduces un-core energy consumption by almost 15× and while also improving performance by 30%, when compared with GraFBoost, thanks to its improved DRAM behavior and more efficient merger unit. Recall that GraFBoost uses a 16-1 merge network (and successively merges exponentially increasing batches of records), contributing to over 94% of its un-core energy.

3.5.4 Sensitivity Analysis

Overhead of GlobalReduce(). The performance overhead incurred due to `GlobalReduce()` is just about 2% on average for MetaStrider, shown in Figure 3.11. Thanks to enhanced Metadata, this is over 5.2× faster than SuperStrider's `Normalize()`.

Benefit of near data processing. Figure 3.12 shows that even the non-NDP variant of MetaStrider reduces traffic on the system interconnect when compared to the baseline by 1.5× and 3.2× for kernels in SpGEMM and Firehose workloads respectively. Upon deploying NDP, there is a further reduction of over 22× in traffic that is compounded. NDP helps further reduce energy consumption (without affecting performance despite its slower logic because the merger unit operation is completely overlapped with DRAM operation).

Speed of Metadata Store. When the Metadata size exceeds area budget, the Metadata Store needs to be adapted to use a combination of slow-fast memories (Section 3.3.2), or pay

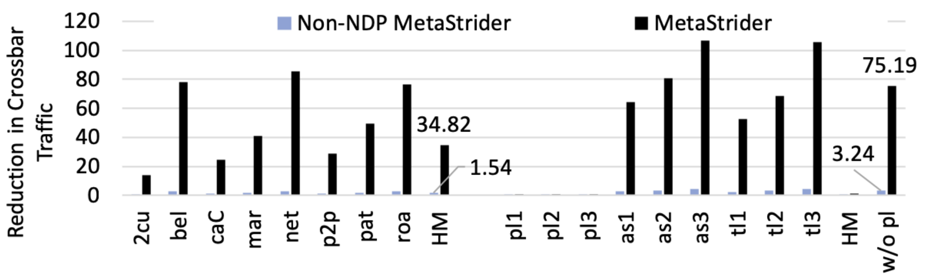


Figure 3.12: Primary benefit of near data processing (NDP) portrayed via the reduction in bytes of traffic on the system interconnect (higher reduction is better). Note that MetaStrider reduces traffic even without NDP.

the penalty of extracting key-ranges via decompression or re-computation (Section 3.3.1). In either approach, an increase in average access time is the result. However, no significant impact ($< 0.5\%$) on kernel speedup was observed when the following representative access times were simulated across all workloads: (i) Idealized, where there is no overhead in accessing the store, (ii) SRAM, and (iii) DRAM, where all the Metadata is in an SRAM/DRAM store respectively.

Real time capability. Since `GlobalReduce()` is a relatively longer latency operation when compared to a `Lookup()`, and that the former is required to guarantee there are no pending off-path duplicates waiting to be reduced, there is a tradeoff between lookup accuracy and performance. When `GlobalReduce()` is omitted, a call to extract the value for a key κ via `Lookup(`κ`)` may yield one of the following cases: (i) Exact match (record with key=κ found and correct value returned), (ii) Partial match (key found but incorrect value, prompting incomplete reduction), and (iii) Missing key (key not found). This is of specific importance to Firehose, where real-time flagging is necessary. In this context, MetaStrider achieves a favorable distribution of 96.5%, 2% and 1.5% respectively. When no tree balancing is done, case (iii) is no longer a possibility. Recall that after a call to `GlobalReduce()`, both (ii) and (iii) would be eliminated. **Benefit of Metadata decoupling and tree balancing**. Decoupling the Metadata from the kv-rows and tree-balancing independently improve DRAM efficiency. However, tree balancing is practical only with decoupling, because of the metadata-heavy nature of re-balancing. As a result,

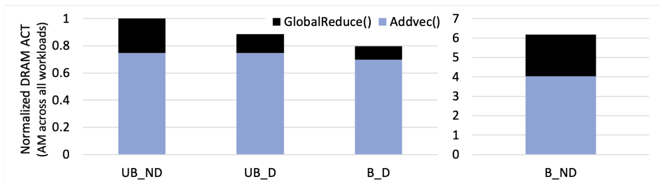


Figure 3.13: Benefit of Metadata decoupling and tree balancing (lower #ACT is better). UB=UnBalanced, B=Balanced. ND=Not Decoupled, D=Decoupled.

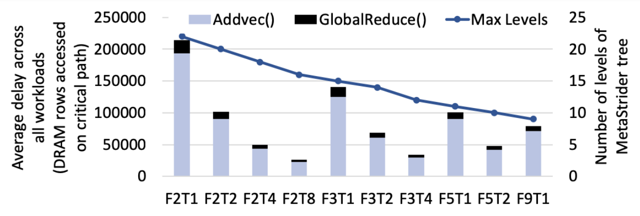


Figure 3.14: Performance of various fanout/partitioning configurations when averaged across all workloads, in terms of DRAM rows accessed (lower is better). MLP = (F - 1) * T, where F=fanout, T=#trees. Performance fundamentally improves with either technique, even in the absence of tree balancing. Furthermore, `GlobalReduce()` overhead is significantly reduced when compared to SuperStrider (Figure 3.3).

MetaStrider deploys a combination of tree balancing and Metadata decoupling. Figure 3.13 shows that a 20% reduction in DRAM ACTs is observed due to AVL-balancing of the tree (when Metadata is decoupled), upon averaging across all workloads.

3.5.5 Scalability Analysis

This section demonstrates quantitatively the tradeoffs described in Section 3.4.

Partitioning vs Fanout. Figure 3.14 evaluates various design points that combine partitioning and fanout, while also depicting the scalability of the performance overhead of `GlobalReduce()`. Although the height of the tree decreases with increasing fanout, note that fragmentation increases as each node in an F-tree spans $(F - 1)$ rows, resulting in an increased average depth per node insertion. Increasing T rather than F is therefore favorable, assuming sufficient front-end core parallelism is available to scatter to each partition.

Figure 3.15 shows that increasing T rather than F is more favorable from both, an energy stand-point and performance stand-point. Furthermore, a load balancing hash for the partitions such as RNS hashing further improves gains. In the figure, note that iso-T

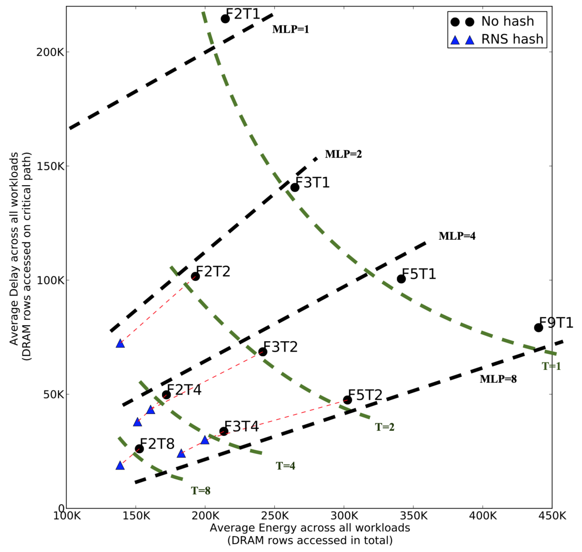
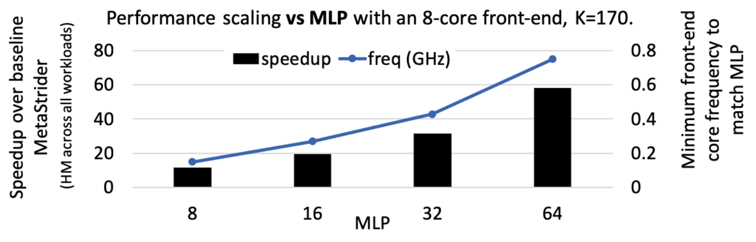


Figure 3.15: A pareto-style tradeoff analysis in DRAM delay and energy for various combinations of tree partitioning and fanout, when averaged across all workloads.

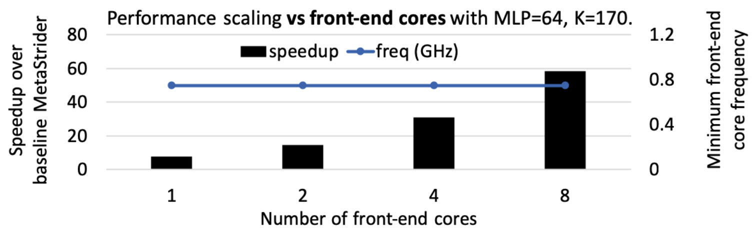
frontiers are indicated with green dashed lines and iso-MLP with black dashed lines.

Pipelining. Ideally, one would therefore allocate a partition for every unit of MLP. However, in a system that is constrained by the number of front-end cores driving MetaStrider, that may not be possible. Instead, pipelining across banks may be used, although perfect MLP utilization would be traded off. Figures 3.16a and 3.16b demonstrate that this is still largely effective in providing near-linear scalability in performance.

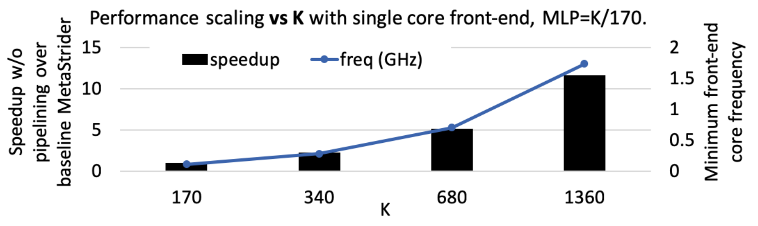
Grouping. As the reader may have guessed, the effect of grouping is similar to that of logically increasing K of the system. This is attractive provided the front-end can provide batches of K-sorted vectors accordingly. Figure 3.16c shows super-linear benefits of increasing K when a correspondingly fast front-end core is used.



(a) Scalability with MLP.



(b) Scalability with cores.



(c) Scalability with K.

Figure 3.16: MetaStrider performance scales near-linearly with available hardware resources using techniques from Section 3.4. The number of partitions (trees) is always equal to the number of front-end cores. A sub-1 GHz core frequency is sufficient to drive MetaStrider, unless a large K is needed.
From Figure 3.15, partitioning is favored over fanout. Therefore, in (a)/(b), partitioning+pipelining is used.
In (c), the impact of row-grouping (alone) is depicted.

CHAPTER 4
SORTCACHE: INTELLIGENT CACHE MANAGEMENT FOR ACCELERATING SPARSE DATA WORKLOADS

4.1 Introduction

Memory architectures traditionally leverage patterned data streams to achieve low latency. However, this predictability is not present for many critical workloads including sparse data applications. These workloads have received considerable attention for several decades [96] because of the unique challenges they pose. As described in Chapter 2, the irregularity of access in sparse applications comes in two flavors: (i) inherent irregularity in the memory-access pattern in the algorithm, such as those in graph analytics [40, 41, 42] and HPC applications [37, 34, 38], and (ii) irregularity imposed by domain experts as a computational performance optimization. A prime example of the latter is optimizations that leverage an algorithm's tolerance to strategically *dropping* some computations without affecting the overall accuracy of results. This is particularly common in today's machine learning workloads, such as in pruned deep convolutional networks [55, 51, 58].

An important category of data-irregular workloads are moderately-sparse workloads, wherein the fraction of non-zeros in the input data stream is typically above 1%. Although these workloads have a high cache hit ratio, they waste cache bandwidth. Modern caches, such as that in Intel's Icelake, are capable of reading an entire cache block in parallel [118]. Furthermore, the capacity and bandwidth of on-chip caches has been on the rise and is

```
for i, j, k = ...:
 partial = a[i] * b[j]
 c[k] = c[k] + partial // Sparse reduction on c with key=k, value=partial
```

Figure 4.1: High-level view of *sparse reduction*, a kernel commonly seen in sparse data applications.

predicted to continue to increase [119]. Therefore, unless all the words within a cache block are useful in a given context, a significant amount of cache bandwidth is destined to be left under-utilized.

Recent research [24, 25, 62, 26, 27] (Chapter 3) has focused on *hyper-sparse* data applications, wherein the fraction of non-zeros in the input data stream is well below 1%. As such, these proposals avoid wasted power and cycles by bypassing the cache hierarchy entirely, and instead focus on optimizing DRAM row performance for irregular accesses. However, utilizing these DRAM-centric accelerators for moderately sparse workloads is inefficient because off-chip accesses are more expensive than on-chip accesses, even if the high on-chip bandwidth is under-utilized. Furthermore, such approaches are difficult to implement in practice because processor vendors and memory system vendors are often separate entities. While memory system vendors are typically commodity-driven (i.e., capacity first with performance second), processor vendors are more centrally focused on application performance.

There are significant advantages to designing an accelerator that augments a general-purpose processor rather than deploying a standalone accelerator. These advantages come in the form of ease of adoptability, integration as well as improved generality. Furthermore, the practice of adopting more and more diverse fixed-function or programmable and specialized hardware into the traditional datapath has been prevalent and successful for decades, ranging from acceleration of floating point computations to vector extensions to tensor augmentations [120]. This chapter makes the observation that **intelligent, processor-centric cache management can improve bandwidth utilization for data-irregular accesses, thereby accelerating moderately-sparse workloads.**

Through an Amdahl's analysis on real hardware, it was found that each of the sparse applications analyzed spend a significant amount of time (49% - 90%) performing the *gather-apply-scatter* transformation, incrementally, on sparse data. In this transformation, the index address of the gathered data is commonly identical to that of the scattered data, meaning that the access pattern of the gather-apply-scatter transformation is equivalent to *sparse reduction*. A high-level view of this sparse reduction is shown in Figure 4.1, which was originally introduced in Figure 2.1 and described in Section 2.2.

Sparse reductions can be viewed as a stream of (key, value) pairs (*kv*-pairs), where the key is equivalent to a matrix index, and the value is its associated non-zero value. An effective method of maximizing cache bandwidth utilization is to *extract* as much sequential locality as possible from this sparse stream of data. This is more commonly done in the slower, block-structured forms of memory using an extract-by-construction paradigm, such as B$^+$-trees or vectorized n-ary forests [121, 122, 27]. This research leverages the prior work to enhance on chip cache subsystems via dynamic sparse-to-dense conversion based on vectorized binary search trees.

A vectorized binary search tree (VBST), as its name suggests, extends a BST such that each node in the tree is a constant-sized, sorted vector of *kv*-pairs. Each node is associated with a *pivot* key, and these pivots in turn form a binary tree. Operations on a VBST are at the granularity of a node. The size of a node is set to be an integral multiple of a cache block (e.g., 64 bytes) to guarantee that VBST operations maximize the internal and external cache bandwidth.

Using VBSTs as a framework, this work proposes **SortCache**, a processor-centric approach to accelerating sparse workloads by introducing accelerators that leverage the on-chip cache subsystem. Its primary benefit is accelerating difficult, moderately-sparse workloads without requiring changes to the processor to memory interface and with minimal programmer intervention. The key contributions of this chapter are summarized as follows:

1. This chapter proposes and demonstrates the performance advantage of adding SortCache as a sparse workload accelerator to on-chip cache subsystems.

2. Unlike prior near-memory approaches, SortCache can accelerate moderately-sparse workloads since it is implemented on chip, closer to the cores.

3. SortCache is shown to be capable of significantly improving performance while also reducing energy consumption across a broad range of graph analytics, high performance computing and deep learning workloads.

Simulation experiments using a diverse set of workloads show that SortCache, when compared to an aggressive baseline, is able to accelerate vector workloads by 75% on average, and unvectorized workloads by an average of $2.2\times$, while also reducing energy consumption by 31% for the former and $9.8\times$ for the latter.

Recall from Chapter 2 the pervasive usage of moderately-sparse reductions in real-world applications. This motivates the design of SortCache, as is elaborated below.

Although all of the moderately-sparse applications introduced in Chapter 2 benefit from the relatively lower latency of the cache hierarchy (vs off-chip DRAM), there is a significant amount of cache bandwidth that goes unused. Modern processor architectures (e.g., Icelake [118]) are able to read at least an entire cache block in parallel to feed their vector function units. In the ideal scenario, when a cache block is read to the function units, all the words contained in the cache block are usable. However, it is found that only a small fraction of words in a cache block load are useful for these applications, requiring multiple such loads per compute instruction on average. This wasted bandwidth requires masking and stalls in vector function units resulting in lower performance and increased energy consumption.

Before describing this further quantitatively, note that the evaluation in this chapter assumes that the *moderately-sparse data of interest is already present in the L1, L2 or L3 (the LLC) caches* and that such data does not get evicted from the LLC in the region of interest. This is supported by the fact that software prefetching and cache tiling are ubiquitous in modern applications [123, 58, 61].

Figure 4.2 shows that only about 10% of the words present in a cache block are utilized when the block is loaded for unvectorized workloads, on average. Explicitly vectorized code (i.e., by an expert programmer [123]) greatly improves the fraction of useful words fetched

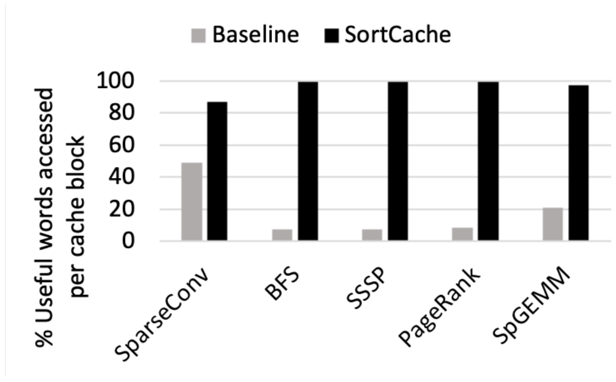


Figure 4.2: Percentage of useful words in a cache block load (higher is better), on average.

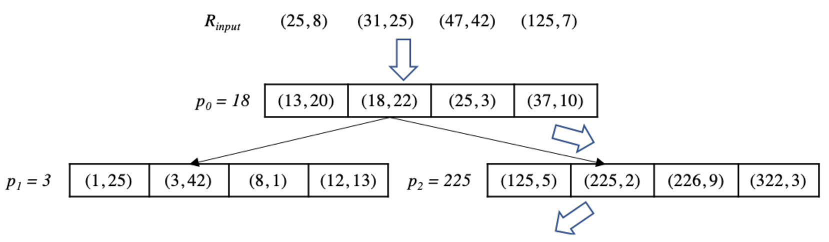
per cache block, as seen with Sparse Convolutions, but this still results in less than 50% of the words being used.

Unfortunately, a majority of sparse applications are not vectorized to start with and continue to leave performance on the table either without explicit vectorization or a mechanism to exploit the caches' internal bandwidth. By paying the smaller cost of smart data reorganization within a cache, SortCache is able to avoid the greater cost of under-utilization, irrespective of whether or not the application is vectorized. **SortCache is the first work capable of maximizing cache bandwidth utilization across such a wide variety of sparse applications, thereby accelerating them.** Furthermore, it is able to do so with minimal programmer intervention.

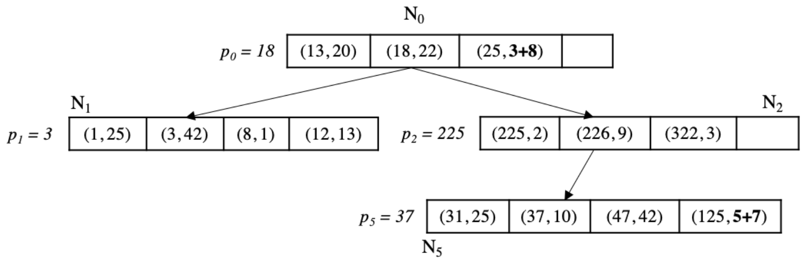
The remainder of this chapter is organized as follows: Section 4.2 provides an overview of VBSTs. Section 4.3 describes in detail the design of a VBST-based SortCache architecture. Section 4.4 and 4.5 present the experimental methodology and empirical evaluations of SortCache, respectively. Finally, related work is summarized in Section 4.6.

4.2 Extracting Sequential Locality via VBSTs

SortCache re-orders sparse data during gather-scatter operations such that sequential locality can be extracted by means of intelligent construction. This mechanism is provided via vectorized binary search trees, which were inspired by the complex n-ary forests proposed



(a) Initial VBST state with dynamic path chosen upon insertion of next R_{input} batch.



(b) VBST state after insertion and in situ reduction of K *kv*-pairs.

Figure 4.3: Example of a VBST with $K = 4$.

in the context of DRAM-based hypersparse accelerators [27]. This section presents the detailed operations that can be performed on VBSTs.

A VBST is defined by four static parameters: K - the maximum number of *kv*-pairs in a VBST node, *sz* - the size in bytes of each *kv*-pair, N_0 - the (address of) root VBST node, and *op* - the reduction (*add* as defined in Chapter 2) operation. Each node, N_i, logically consists of three *tag* fields in addition to the K *kv*-pairs: the pivot, p_i, and the addresses of the left and right sub-VBSTs. Two invariants are enforced in the construction of a VBST, and these are fundamentally critical to scalably extracting sequential locality: (i) *for each N_i, all keys in its left sub-VBST are smaller than p_i, which in turn is smaller than all keys in its right sub-VBST*; and, (ii) *at each N_i, all of its* kv-*pairs are sorted by key*. These are shown in Figure 4.3.

4.2.1 Insertion

To efficiently enforce the required invariants, a VBST is built incrementally over time. First, K unsorted *kv*-pairs (*records*) are supplied by the input as a batch to the VBST. Second, these records are propagated down a dynamic path through the VBST, from root to leaf. En route, reductions are performed in-situ, and duplicate keys are coalesced with their reduced value. In the event where there are insufficient repeated keys, a new leaf node is created. As a result, all the input records end up being written into the VBST in addition to sparse reductions (and de-duplication) performed on records with same keys. The specifics of dynamic path selection, leaf creation and how the invariants are maintained are described via an example below.

Consider the state of the tree as shown in Figure 4.3a. Assume *op* is the $+$ operation and the records in the new batch from the input are $R_{input} = [(25, 8), (31, 25), (47, 42), (125, 7)]$. The records R_0 from the root node N_0 are *merged* with R_{input} to obtain a temporary vector $R_{temp} = [(13, 20), (18, 22), (25, 3 + 8), (31, 25), (37, 10), (47, 42), (125, 7)]$. This merge operation sorts by key the records in R_{input} and R_0, while also performing a reduction operation (according to *op*) on the values of records with matching keys. After reduction, the redundant records are compressed out, resulting in a total length of at most $2K$ for R_{temp}.

Next, R_{temp} is partitioned, by p_0, into two vectors $R_{large} = [(25, 11), (31, 25), (37, 10), (47, 42), (125, 7)]$ and $R_{small} = [(13, 20), (18, 22)]$. Note that R_{large} is of length at least K, and R_{small} is of length at most K. The partitions are re-adjusted if necessary such that R_{large} has length exactly K. In this example, $(25, 11)$ is moved to R_{small} to achieve this. Subsequently, R_{small} is written back to N_0 as R_0. If the remaining keys in R_{large} are greater than or equal to p_0, then the above procedure is repeated with the right sub-VBST (root at N_2) – else, with the left sub-VBST (root at N_1) – with R_{large} as the new "R_{input}" to the sub-VBST. This is repeated until $R_{large} = \phi$, which may require creating a new leaf as part of the last step. In this example, the path taken is $[N_0, N_2, N_5]$. Upon leaf creation, at N_5, p_5 is assigned as the median key of R_5, resulting in the updated state of the VBST as shown in Figure 4.3b.

Insertion of a K-batch visits each level of the VBST at most once, resulting in approximately $lg\ \frac{|R_{stream}|}{K}$ nodes being visited per insertion, where R_{stream} is the number of unique records in the VBST at the time of this insertion. *Observe that as part of the insertion, sparse reduction is also performed, while also maximizing the number of consecutive words accessed.*

A small fraction of duplicate records may exist outside of the dynamic path chosen, but these are resolved via a simple depth-first-search after the input has been consumed. Several applications, including those evaluated in this work, are tolerant to such partial reductions either due to their approximate nature (i.e., deep learning) or their iterative nature. (This post-processing step is not described here as it has already been described in Chapter 3.)

4.2.2 Lookup

A *Lookup* is a gather that is not followed by an apply-scatter. (Note that if a record was gathered for the sole purpose of apply-scatter, as is the common case in the workloads evaluated, then that is more efficiently performed via the insertion procedure above.) There are two variants of lookup seen in applications: *random*, and *ordered*.

Random scalar lookup for the value associated with a key κ is achieved via a VBST walk from the root. If $R_0[0] \leq \kappa \leq R_0[K-1]$, then κ, if it exists, exists in N_0, and a binary search within N_0 may be performed since R_0 is sorted. Else, if $\kappa < R_0[0]$, the scalar lookup procedure can be performed recursively the left sub-VBST. Else, when $\kappa > R_0[K-1]$, a lookup can be performed on the right sub-VBST. Therefore, this visits at most $lg\ N$ nodes, where N is the number of nodes in the tree at the time of lookup. This can be improved further by storing the range and address information of each node in a separate compact data structure (see Section 4.3.1).

Ordered lookup is more straightforward and efficient. From Figure 4.3, it is apparent that an in-order, depth-first-search traversal of the VBST starting at the left-most leaf results in records being read out in increasing key order.

4.2.3 Deletion

Although not necessary for the workloads evaluated in this work, we briefly discuss deletion. Batch deletion is useful when certain records need to be “aged out” for capacity constraints, such as for real-time intrusion detection in cybersecurity. Deleting records can be achieved by performing the insertion procedure above, except with the values in R_{input} set to 0. During the merge step performed on R_{temp}, when a 0-value is detected, the reduction operation can then omit the associated key(s) entirely, or propagate them to a backup VBST via a write-back insertion procedure.

Clearly, the insertion procedure above is the most important and expensive VBST operation. Therefore, the microarchitectural design for SortCache optimizes for insertion. For the remainder of this chapter, VBST or SortCache operations refer to insertion, unless otherwise specified.

4.3 Repurposing Traditional Caches for Sparse Reductions

This section presents how the SortCache design implements a VBST. Figure 4.4 shows a high-level overview of SortCache components from the perspective of a single cache.

4.3.1 Data Layout

Recall that the goal of SortCache is to make effective use of cache bandwidth by maximizing intra-cache-block locality. Since records within a VBST node are contiguously accessed, they should not cross cache block boundaries. Therefore, the records of a VBST node are logically mapped onto an integral multiple of cache blocks. The question of finding the right integral multiple is addressed below in Section 4.3.4.

The VBST itself is partitioned across the cache hierarchy either statically or dynamically. Closer inspection of VBST operations reveals that the higher level nodes of the VBST

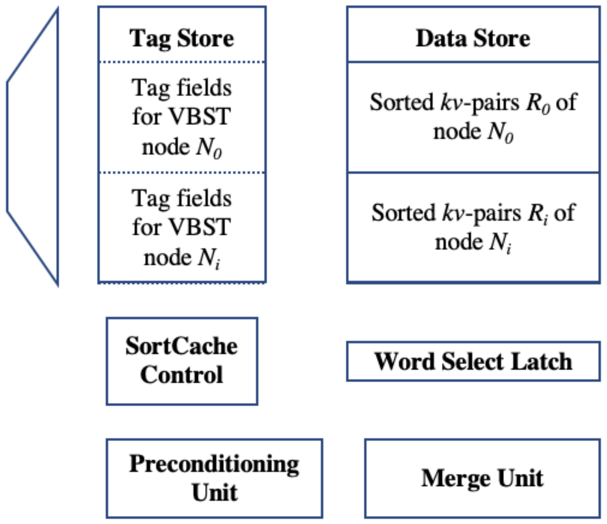

Tag Store
Tag fields for VBST node N_0
Tag fields for VBST node N_i
Data Store
Sorted *kv*-pairs R_0 of node N_0
Sorted *kv*-pairs R_i of node N_i
SortCache Control
Word Select Latch
Preconditioning Unit
Merge Unit

Figure 4.4: An overview of a traditional cache augmented with SortCache hardware. For simplicity, 2 nodes are assumed to fit in the cache, and ways, tag compare, pre-decoders, mats etc. are omitted from the figure.

are more likely to be accessed than the lower (leaf) level nodes. Therefore, nodes closer to the root must be installed higher up in the cache hierarchy. This can be achieved in two ways: (i) *Static*: assign a logical node-index to each node via a top-down topological sort. Given L1, L2, L3 cache capacities available to SortCache as C_1, C_2 and C_3 bytes, respectively, and a node size of sz_{node} bytes (sz_{node} can be computed as $sz_{record} \times K$), then assign nodes $\{N_0, N_1, ..., N_{\frac{C_1}{sz_{node}}-1}\}$ to L1, the next $\frac{C_2}{sz_{node}}$ nodes to L2, and the remaining to L3. Alternately, there is (ii) *Dynamic*: perform demand-based block eviction/install to occur at the granularity of a node. For this, the existing LRU counters are slightly modified to operate at a node-granularity rather than at a block-granularity basis (recall that nodes span an integral multiple of blocks). Given the skewed access distribution of the nodes, the dynamic approach tends to mimic the partitioning strategy described above, while also being able to account for unexpected regularity in the otherwise irregular sparse input stream. Note that node-level LRU counter operation is necessary because it is absolutely essential that all records within a node migrate together. During SortCache operation, due to insitu reduction, holes towards the end of nodes may dynamically appear and disappear. Using regular LRU may therefore result in partial eviction, and tracking these would result in more complexity than using LRU counters that operate at a node granularity would.

The per-node *tag* fields (pivot, addresses of left and right children) can be stored in the cache's tag store fields associated with the nodes. This is due to three reasons: (i) the traditional tag bits are unnecessary for the hardware managed *kv*-pairs, (ii) tag access in parallel with data access, as is common in traditional caches, offers zero performance overhead, and (iii) there is sufficient tag storage in traditional caches, as will be explained in more detail below, offering enough headroom for future SortCache implementations that may need more *tag* than that required by the VBSTs described in this work. For example, consider a typical C_1/C_2/C_3 of 32 KB/256 KB/16 MB (for a total of 16.28125 MB available for the VBST) and relatively small $K = 64$. Note that a smaller K results in a larger number of VBST nodes, requiring more number of bits to encode addresses of children as well as a higher number of *tag* instances– thus making this example a worst-case scenario. Assume $sz_{record} = 4$ bytes, the pivot key requires 2 bytes, $sz_{node} = 4 \times 64 = 256$ bytes, number of nodes $N = \frac{16.28125 \times 1024 \times 1024}{256} = 66688$, which is slightly over 2^{16}. In practice, not all of the cache hierarchy would be available to SortCache as the non-reduction portions of the application would require some caching. Therefore, 16 bits (2 bytes) are sufficient to address all nodes for this example. From an implementation perspective, the child addresses are not stored as node indices, but are encoded according to their 16-bit geometrical coordinate locations L in the cache hierarchy. L is of the form {`cacheId`, `indexId`, `wayOffset`}. With 8-way caches, a 3-bit wide `wayOffset` suffices. The width of `indexId` increases with cache size, i.e., from L1 to L2 to L3. The remaining MSB bits form the `cacheId` field, to help decode whether the node is in L1, L2 or L3. This results in the *tag* fields requiring 2 (pivot) $+2 \times 2$ (child addresses) $= 6$ bytes per node, or $\frac{32 \times 1024}{256} \times 6 = 768$ bytes, $\frac{256 \times 1024}{256} \times 6 = 6$ KB, $\frac{16 \times 1024 \times 1024}{256} \times 6 = 64$ KB for the *tag* fields of L_1, L_2 and L_3 respectively. Assuming 8-way caches, these are much smaller than the available traditional tag storage (2.875 KB, 21.5 KB, 1.15625 MB respectively).

4.3.2 Merge Hardware

In order to maintain the two VBST invariants for efficient operation, efficient merge is a requirement (Section 4.2). Therefore, SortCache relies on explicit hardware support for merging the records of the input batch R_{input}, with those of a given node R_i (R_0 to start with).

Recall that R_i is always sorted, and the intermediate results R_{temp} are also sorted. This means that for the majority of operation, merging takes two sorted vectors as input, which is significantly more efficient than generic sorting. However, R_{input} is unsorted as no such data transformations are assumed to be applied by the main application. Thus, *SortCache incrementally preconditions R_{input} (i.e., sorts an input batch, not the entire input) such that further merging is more efficient.*

One method to sort unsorted data is to break it in two and repeatedly merge it. This allows the same merge hardware used for merging sorted vectors to be used to sort R_{input}, with more iterations. If the merge hardware is replicated, preconditioning on a given input batch can be done in parallel with a preceding series of merge operations (resulting from the previous input batch). Such replication results in lower hardware necessary than having a dedicated sorter, without lost performance. This is because a single preconditioning is typically followed by several downstream merge operations, potentially down to the leaf node, enabling preconditioning and subsequent merging to be pipelined.

Two variants for the underlying merge hardware are now presented, followed by the control logic. While the latter is distributed across the cache controllers, the merge hardware is assumed to be close to the L1 cache controller to save power as it is more likely to access records in L1 than in L2/L3.

Linear Merge

Since the fundamental merge operation is performed on two sorted vectors, it can be done in linear time. For example, say $R_{input} = \{(1, 1), (2, 24), (5, 42), (10, 35)\}$ and $R_0 =$

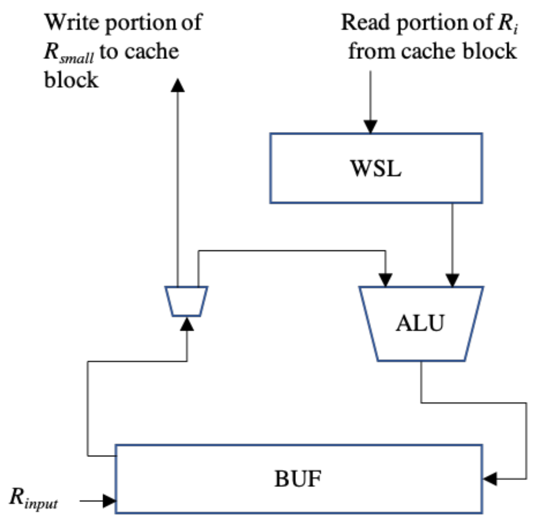

Write portion of R_{small} to cache block
Read portion of R_i from cache block
WSL
ALU
BUF
R_{input}

Figure 4.5: SortCache serial merge and preconditioning hardware.

$\{(2,5),...\}$. First, the keys from (1, 1) and (2, 5) are compared. Since $1 < 2$, (1, 1) is written out to R_{temp} and the keys from (2, 24) and (2, 5) are compared. Next, a reduction is performed and (2, 24 *op* 5) is written out to R_{temp}. This process is repeated, advancing a record from either vector until all records are consumed.

R_{input} is buffered in a $2K$-record sized FIFO (called *BUF*) with two input and one output port. BUF also doubles as the real estate for R_{temp}, with an advancing head pointer to separate the two. Recall from Section 4.3.1 that R_0 is housed in a K-dependent number of contiguous cache blocks. Post the wordline, bitline and sense-amplifier operations, the word select latch (WSL) has a cache block's worth of records, which are read one-by-one by the ALU (i.e, compute unit that performs comparison and reduction *op*) and populated into BUF (see Figure 4.5). Therefore, WSL$_{read}$/BUF$_{read}$, ALU and BUF$_{write}$ form pipeline stages that can operate without stalls for the lifetime of the current cache block, after which the entire process is repeated. At design time, SortCache examines the latencies of each stage and generates a 2-stage or 3-stage pipeline accordingly. At the end of each node-merge step, R_{small} is written back from BUF back to the most recently read cache blocks.

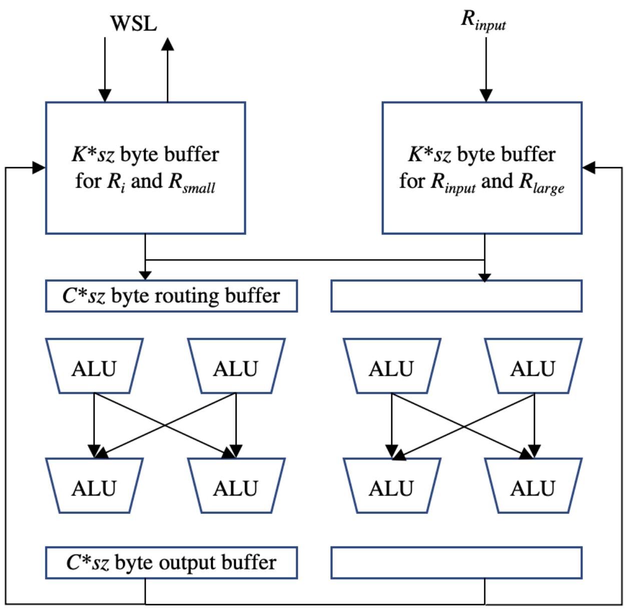

WSL
R_{input}
K*sz byte buffer for R_i and R_{small}
K*sz byte buffer for R_{input} and R_{large}
C*sz byte routing buffer
ALU
ALU
ALU
ALU
ALU
ALU
ALU
ALU
C*sz byte output buffer

Figure 4.6: SortCache parallel merge and preconditioning hardware with $C = 4$ as an example.

Parallel Bitonic Merge

Since the WSL loads a cache block in parallel, rather than serially reading from WSL, reading and merging in parallel using a bitonic merge network is possible. This is a classic power-performance tradeoff between the two approaches.

The bitonic merge network takes two K-record sized sorted vectors and passes them through a C-wide and $lg\ C$-deep bitonic network of ALUs, where $C < K$ (see Figure 4.6). The records are read in $lg\ K$ strides of K, $\frac{K}{2}$,..., 1. For each stride, C records are routed to the $lg\ C$-deep network in a depth-wise pipelined manner.

Preconditioning of R_{input} can be done by repeating the above operation, with 1 stride in the first iteration, 2 strides in the second iteration, ..., $lg\ K$ strides in the final iteration.

Control Logic

The control logic of SortCache is responsible for two major class of operations: (i) SortCache mode macro-level semantics, such as routing records to the appropriate cache (Section 4.3.1), setting/resetting node-level LRU/install/evict logic, way-partitioning and lazy writebacks

```
loop:
 MUL R_value, R_a, R_b
 LD R_tmp, [R_c], offset
 ADD R_tmp, R_tmp, R_value
 ST R_tmp, [R_c], offset
```

Figure 4.7: Pseudo-assembly for the sparse reduction kernel.

```
loop:
 MUL R_value, R_a, R_b
 MOV R_key, offset2
 RED R_key, R_value
```

Figure 4.8: The `RED` instruction instructs SortCache to perform *gather-apply-scatter*. `offset2` is different from `offset` as it has to account for storage of keys alongside the values (rather than in a separate vector such as in CSR). If size of key and value are same, `offset2 = 2*offset1`.

(Section 4.3.3); and, (ii) VBST operation micro-level semantics, such as routing records between cache blocks and merge networks above and dynamically deciding the root-to-leaf path to be taken after each node-merge (Section 4.2).

4.3.3 System Integration

SortCache operation can be triggered using a simple `RED` R_{key}, R_{value} instruction. The sparse reduction kernel introduced in Figure 4.1 can then be transformed from the pseudo-assembly in Figure 4.7, to that in Figure 4.8 by a compiler or programmer. Pragma-style programmer annotations specify the static VBST parameters (K, sz, reduction operator), base pointer and scope of reduction, as shown in Figure 4.9.

Although sparse reductions account for the majority of cache bandwidth usage, applications may need to retain a portion of the cache capacity for other memory accesses.

```
#pragma SortCache INIT K=256, sz=4, op='+'
#pragma SortCache RED base_pointer=c
for i, j, k = ...:
 partial = a[i] * b[j]
 c[k] = c[k] + partial
...
print(c)
#pragma SortCache END base_pointer=c
```

Figure 4.9: Programmer annotations to leverage SortCache.

Automatically determining the general fraction of cache capacity required for regular memory accesses is beyond the scope of this work. However, for the memory footprint and tile sizes in the workloads evaluated in this work, it has been found that reserving the L1, half of L2 and half of L3 for sparse reductions is sufficient to prevent L3 evictions for the VBST, while also not introducing regular memory accesses onto the critical path of the application. Reserving portions of a cache can be done via static way-partitioning or address-based set-partitioning. In this thesis, all 8-ways of L1 and the last 4 ways of 8-way L2/L3 caches are assumed reserved for VBSTs (during SortCache operation). As the sequence of node accesses is available only one node at a time for a VBST, all the cache MSHRs are available to the other portions of the application (outside of the sparse reduction operation).

On encountering the first `RED` instruction, the cache hierarchy is transformed into SortCache mode, wherein the modified node-level LRU counters and install/evict logic are used for the reserved portions of the cache. Lazy flush (dirty writeback or clean invalidate) of L1 and the reserved ways of L2/L3 are also performed when nodes are installed into the cache. At the end of a `RED` region, rather than reverting to traditional cache mode immediately, contents of the reserved portions are retained so that downstream lookup operations can be performed on the VBST directly (Section 4.2.2). If no more operations on the reduced sparse data are necessary, as can be signaled by the `SortCache END` pragma, reservations are lazily invalidated and released for regular cache operation.

4.3.4 Choosing *K*

Recall from Section 4.2 that increasing K, the number of *kv*-pairs in a node, reduces the average VBST depth and therefore increases performance. However, it also increases the amount of data to be merged on a per-node basis as well as the preconditioning (pre-sorting) of the input batch of records for efficient subsequent merging (Section 4.3.2).

Formally, the number of serial comparison operations grows as $O(Klg\ K)$ for preconditioning and $O(K)$ for linear merge at a node. As there are (ideally) $O(lg\ \frac{R_{stream}}{K})$ nodes that need to be visited per input-batch (R_{stream} is the number of unique records already in the tree at time of insertion), the cost to inserting a batch of records may be viewed, to a first order, as $O(Klg\ K + Klog\ \frac{R_{stream}}{K})$. Upon performing a differential analysis over K, and assuming that the pre-conditioning is overlapped with (previous) merge series, it can be seen that a higher R_{stream} favors a higher K for high performance.

However, from an energy stand point, a higher value of K is more detrimental to the log-linear bitonic network when compared to the linear merger network. As such, the evaluations in Section 4.5.5 find that the linear mergers favor a higher K than the bitonic network.

4.4 Evaluation Methodology

4.4.1 Sparse Applications Evaluated

A wide variety of sparse data applications are simulated in this study, with multiple inputs used for each workload category introduced in Chapter 2, SparseConv, BFS, SSSP, PageRank and SpGEMM. These are described here.

SparseConv: A caffe-based [124] implementation of pruned, deep convolutional neural networks, known as SkimCaffe [123, 58], was used as the framework for real-world sparse convolutions. Each of the four sparse CNNs that resulted from the pruned AlexNet [57] topology were considered (the one dense CNN was not included), with inputs from the ImageNet ILSVRC-2012 dataset [125] provided to the network. Note that, among the evaluated workloads, this is the only category where the framework included carefully hand-optimized vectorized code done by domain experts.

BFS, SSSP, PageRank: Graphmat [61], an SpMV based engine, was used as the framework for graph analytics.

SpGEMM: The most work-optimal algorithm for SpGEMM, outer-product [65], was used. For the SpMV and SpGEMM based workloads, the input matrices were selected from the SuiteSparse repository [126]. This was guided by prior sparse research [65, 88, 91, 92]

Table 4.1: Baseline Configuration

Parameter	Value
Front-end Core	Infinite issue width
	Perfect memory disambiguation
Cache Policy	Inclusive, LRU replacement
Block Size	64 bytes
Granularity	1 byte to 64 bytes per access
L1	32 KB, 8-way, 10 MSHR [118]
	2 Read-ports, 1 Write-port
L2	256 KB, 8-way, 20 MSHR
L3	16 MB

and by the goal to demonstrate SortCache on a wide spectrum of domains.

Table 2.1 summarizes several key characteristics of the workloads evaluated. Most notable is the relatively high Amdahl's percentage of sparse reductions across multiple domains. The Amdahl's analysis was performed via native execution on a 2.3GHz, 4-issue AMD Opteron 6276 with 2MB LLC per core.

4.4.2 Simulation Infrastructure and Configurations

The sparse reduction kernel was evaluated using a cycle-accurate, trace-based simulator. Timing, power and area for the caches (baseline and SortCache) and buffers (SortCache) were modeled in detail using CACTI [112]. Those for the computational logic (SortCache) were modeled using 15nm-based RTL synthesis results from Synopsys Design Compiler [127] using the FreePDK15 standard cell library [128].

Recall that SortCache focuses on extracting maximum cache-*hit* bandwidth. To isolate this, the simulator models a front-end core with infinite issue width and perfect memory disambiguation, connected to a typical cache hierarchy. For the *baseline* (Table 4.1), the L1 has 1 write-port and 2 read-ports (true dual porting) with a 64 byte wide interface, as used in modern vector processors such as Intel's Icelake [118]. For SortCache (Table 4.2), these parameters are retained, except that the caches have only 1 read-port (and 1 write-port) each. Specifics of cache partitioning/reservation for SortCache operation has been described in

Table 4.2: SortCache Configuration

Parameter	Value
Cache Policy	Static, Dynamic (Section 4.3.1)
Record Size (*sz*)	Application dependent
Records per Node (*K*)	64 - 8192
Node Size	$K \times sz$ bytes
Bitonic Network Size (*C*)	2 - 64
Reduction *op*	+, min, predicated assignment
Preconditioning and Merger Unit Architectures	Linear-Linear, Linear-Bitonic, Bitonic-Bitonic

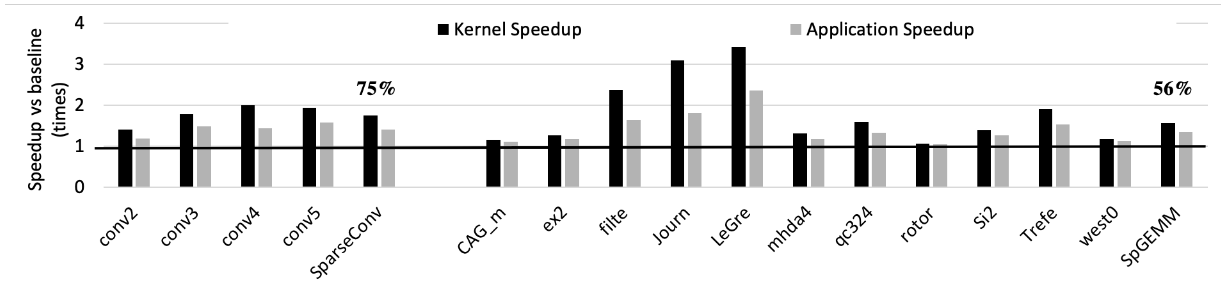


Figure 4.10: Speedup of Sparse Convolutions and SpGEMM vs the baseline.

Section 4.3.3.

To simulate SortCache, the frameworks mentioned above were modified according to Figure 4.9 to generate a dynamic `RED` instruction stream. Both static and dynamic VBST node placement strategies (Section 4.3.1) were evaluated. K was varied from 64 to 8192 (log-scale). (Exploring higher K values would have caused a node to not fit entirely in L1, whereas lower K values would have resulted in trees with too many levels to be practical.) The record size (*sz*) was determined based on the desired key range that to be encoding. Most workloads require 4 bytes in practice.

This research was supported in part through research cyberinfrastructure resources and services provided by the Partnership for an Advanced Computing Environment (PACE) at the Georgia Institute of Technology, Atlanta, Georgia, USA [129].

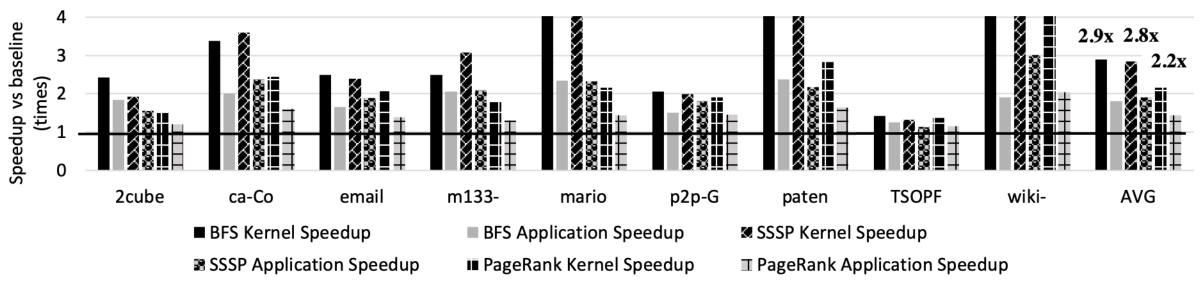


Figure 4.11: Speedup of BFS, SSSP and PageRank vs the baseline.

4.5 Experimental Results

4.5.1 Finding universally-beneficial SortCache configurations

Before presenting results, details of the SortCache configuration that benefits all of the workloads while not being over-fitted for any particular workload, are presented.

First, in terms of the merger approach, the linear merger was sufficient to extract significant performance benefits for BFS, SSSP, PageRank and SpGEMM based workloads. The more powerful bitonic merger was used for SparseConv, as it had to compete against a more aggressive baseline, which, as has been discussed above, benefitted from optimized hand-vectorized code. Thus both mergers are included with SortCache.

Second, both the preconditioning unit and the merger unit were assumed to be identical (linear-with-linear, bitonic-with-bitonic), as increasing design complexity with heterogenous units did not yield any significant benefits.

Third, a high $K = 8192$ was set for the linear merger and a low $K = 128$ was set for the bitonic merger (with $C = 32$) (for more details see Sections 4.5.5, 4.5.6).

Finally, the dynamic node placement strategy (Section 4.3.1) was used as it resulted in the most number of higher-level cache hits (for more details see Section 4.5.7).

4.5.2 Performance Results

Figure 4.10 presents SortCaches speedup over the baseline, both for the sparse reduction kernels and the entire application for SparseConv and SpGEMM. Figure 4.11 presents those for BFS, SSSP and PageRank.

Referring to the results, all of the workloads achieve at least a 15% speedup, with the exception of SpGEMM/rotor (6%). This is because the optimal K for rotor is 2048, which would have yielded a 20% speedup. In general, each of the workloads have their own optimal point for K, depending upon the nature of the input, or more precisely in this case, on the density distribution of the resulting VBST nodes. When reduction (repeated keys) manifests uniformly over time (batches) and space (keys), a large fraction of VBST nodes result in empty space in them, implying that a lower K would have been more efficient.

There are two solutions (whose details are beyond the scope of this work) to improve the performance of such tail workloads: (i) Periodically "defragment" the VBST; and, (ii) Leverage compiler assisted inspector-executor strategies [130] in determining the optimum K for a given phase of the application.

4.5.3 Energy Results

Using the same overall SortCache configuration as above, Figure 4.12 shows the average normalized energy across workloads. The linear merger based design (used for BFS, SSSP, PageRank, SpGEMM) offers an order of magnitude lowering in energy consumption. The bitonic network used for SparseConv is also able to significantly lower energy even though it competes against the hand-optimized baseline.

The overheads due to accessing the *tag* fields, the compute logic and the buffer are insignificant in general (Figure 4.13). Moreover, because of effective use of cache bandwidth (Figure 4.2), significant gains come from lower cache energy. For the bitonic network, there is a strong correlation between its relatively higher energy consumption (as compared to the linear merger) and its heavy internal buffer usage.

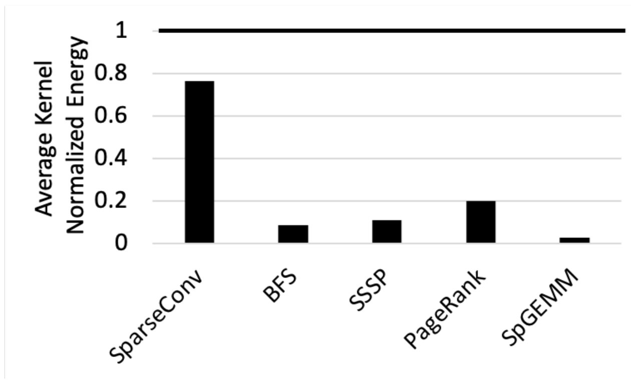


Figure 4.12: SortCache energy normalized to the baseline (lower is better).

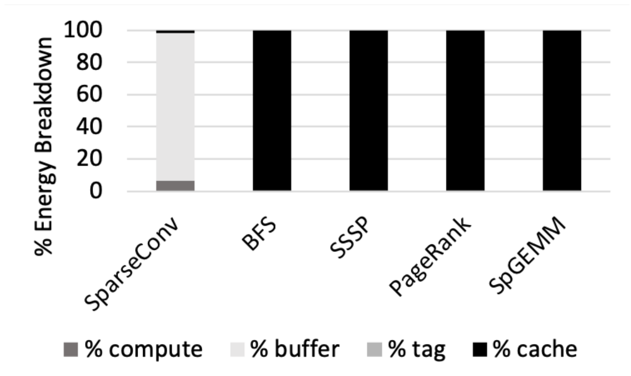


Figure 4.13: Breakdown of energy consumption of each SortCache component.

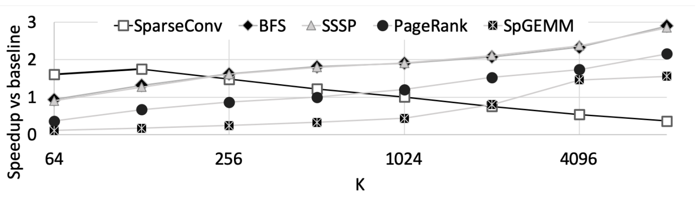


Figure 4.14: Impact of K on performance

4.5.4 Area

The bitonic network requires an area of 0.26 mm^2, of which 0.01 mm^2 is needed for the network of ALUs. This, similar to the energy consumption, speaks to the buffer-heavy nature of this design. The linear merger requires an area of 0.14 mm^2 (which is almost entirely due its BUF structure). As these networks are duplicated to perform pipelined preconditioning, a total area overhead of 0.79 mm^2 is required. This is under 5% of that of the cache hierarchy.

4.5.5 Impact of *K*

As discussed in Section 4.3.4, increasing K tends to improve performance since it leads to fewer nodes accessed per insertion, thanks to a shallower VBST. However, the cost for increased K is a higher preconditioning and node-merge overhead. Given the range of K explored (Section 4.4.2), the benefits of the former outweigh the costs of the latter for the linear merger, as seen with the BFS, SSSP, PageRank and SpGEMM based workloads in Figure 4.14.

However, for a similar balance to be achieved with the bitonic merger, the network throughput must be increased for higher values of K. A higher C improves the parallelism within the network, but stresses the already buffer-heavy design. With $C = 32$, an initial performance improvement is seen when K is increased from 64 to 128 due to a shallower tree, but this rapidly drops as K is increased further due to higher node-merge latency.

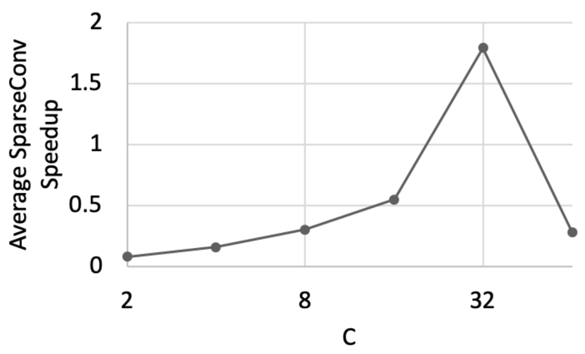


Figure 4.15: Impact of bitonic network size on performance.

4.5.6 Impact of the Bitonic Network Size (C)

Figure 4.15 shows the average performance improvement relative to the baseline for SparseConv, as a function of bitonic network size.

As discussed above, increasing C increases the parallelism available in the bitonic network. Initially, this provides performance benefits (at increased power) since a wider set of ALUs are able to operate in parallel. However, beyond $C = 32$, the performance quickly tapers off because of the routing and output buffers becoming bottlenecks. Another way of viewing these buffers is through the lens of a crossbar-like structure that routes records to and from the C ALUs. For these reasons, the sweet spot of $C = 32$ has been used for SortCache. A more complex design that deploys a hierarchical bitonic network (i.e., a merge tree of smaller bitonic networks) may be a scalable solution, but at higher cost.

4.5.7 Impact of Node Placement and Merger Replication

In deciding where to install a newly-created leaf VBST node in the cache hierarchy, Section 4.3.1 introduced using either a dynamic node-LRU based approach or a static approach where topologically higher nodes were placed higher in the cache hierarchy. A hybrid is also possible where only the topologically higher few nodes are placed in the L1, while L2 and L3 would resort to using a node-level LRU behavior, similar to the dynamic approach.

While the static and hybrid approaches would benefit a few workloads, the dynamic

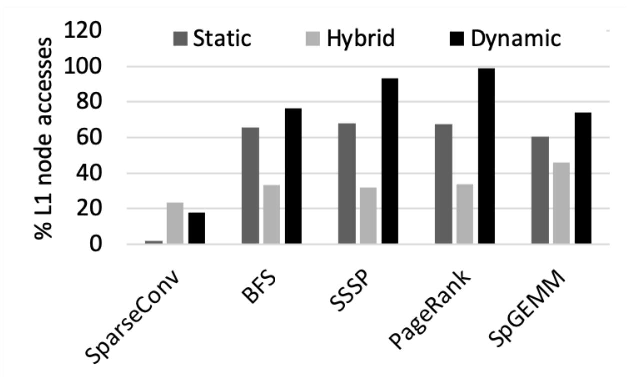


Figure 4.16: Effectiveness of VBST node placement strategy in terms of percentage of accesses to the cache hierarchy being L1 hits (higher is better).

approach is better overall. This is because VBST access paths are non-uniform. Although the static approaches make strong intuitive sense in that the topologically higher nodes are more likely to be accessed, once the access path lengths increase, the lower nodes need to be accessed (and often, for certain input patterns). The dynamic approach is able to promote these "hot" lower nodes in addition to ensuring that a majority of the topologically higher nodes (especially the root node) that are accessed most frequently remain in the L1, as seen in Figure 4.16.

It can further be observed that the L1 is able to satisfy a vast majority of SortCache accesses on average. The exception is SparseConv, which requires smaller node sizes because of its more expensive bitonic merger. As a result, this work replicated neither the linear nor the bitonic mergers. However, if there is headroom in the area budget, replicating these mergers at lower levels of the cache would reduce the energy cost of intra-chip data movement.

4.6 Related Work

Compiler and profiler assisted transformations of data to improve cache locality have been researched for over two decades [131, 132]. Several works have repurposed the cache real estate as a software-managed scratchpad memory, to reduce redundant data transfers in the context of CPUs, GPUs and embedded systems [133, 134, 135, 136]. Hardware-based transformations of cache data layout to improve locality have also been proposed, in the form of reducing conflict and capacity misses using alternate indexing and address transformations in traditional caches [137, 31]. However, these works view data reorganization from the perspective of dense data streams.

On the other hand, transformation-based hardware acceleration of sparse data streams has gathered significant interest in the near-data (DRAM) processing community [27, 24, 26, 62, 138, 25]. Most of these approaches bypass caches and target hyper-sparse applications, and are relatively harder to adopt by virtue of requiring off-chip computation. Two recent notable exceptions are HTA [139] and SPiDRE [75]. These approaches extract cache locality and do not require off-chip computation.

HTA targets data-irregular accesses that arise from hash tables, by making use of a cache-conscious data layout. This is similar to SortCache in that sequential locality is leveraged. However, no global ordering among the cache blocks is necessary as the block index is provided directly by the hashing function, and significant research has gone into designing hashes that minimize collisions in hash table based applications such as databases [140], genomics [141], etc. Unfortunately, the same cannot be said about the sparse workloads of interest in this paper, thus requiring a dynamic approach such as VBSTs. SPiDRE performs LLC prefetching via a programmer-specified rearrange function, therefore improving cache hit ratios and also useful cache bandwidth. Although this improves sparse-gather performance, sparse-scatter is not accelerated. However, SortCache can benefit from SPiDRE as it can potentially accelerate the front-end generation of partial products for SpMDM, SpMV and SpGEMM.

Another class of cache-based acceleration has gained traction in the recent past, in the form of in-cache [76, 77, 78] or near-cache computation[142, 143]. These approaches do not transform the data layout but instead focus on performing useful computation via the cache hierarchy. Such computation either leverages emerging SRAM circuit technology (“bitline computing), or introduces a co-processor to the cache hierarchy. Fundamentally, bitline computing [80, 81] activates multiple word-lines in parallel and infers an *and* (or *nor*) operation by sensing the shared bitline (or its complement). SortCache has the potential to benefit from such parallel activations as it can accelerate the key-compare and value-reduce operations, thus making the preconditioning and merge units more efficient.

Finally, recently proposed research, known as PHI [63] is an orthogonal approach to the sparse reductions problem. In particular, PHI focusses on push-based updates in the context of graph analytics, with an emphasis on reducing traffic to main memory rather than maximizing cache bandwidth utilization. It successfully leverages temporal locality in keys via coalescing, i.e., by treating portions of the cache hierarchy as a write buffer. Traffic to main memory is further lowered by batched updates. However, during their in-cache update phase, which is fundamentally scalar in nature, unless the stream of sparse reductions has significant sequential locality to begin with (which, according to Figure 4.2, is not common unless the program was explicitly vectorized), PHI is unable to effectively utilize the wide cache bandwidth available in modern processors, which is where SortCache excels. Therefore, PHI and SortCache, when used together can benefit from each other. Both approaches reduce synchronization overheads as multiple cores can afford stateless offloads of their sparse scatters. As an alternative to software tiling, SortCache can lean upon PHI to reduce memory traffic. PHI can lean upon SortCache for more efficient in-cache sparse updates.

4.7 Summary of Mechanisms to Accelerate Sparse Reductions

Several mechanisms have been proposed in Chapters 3 and 4 to accelerate sparse data applications. These are summarized and categorized visually in Figure 4.17. First, there are those mechanisms that are fundamentally necessary and central to obtaining performance and energy efficiency. Second, there are several algorithmic, architectural, platform-specific as well as implementation-oriented optimizations that have also been proposed.

At a higher level, a key takeaway from this thesis is that explicit, intelligent and easily-

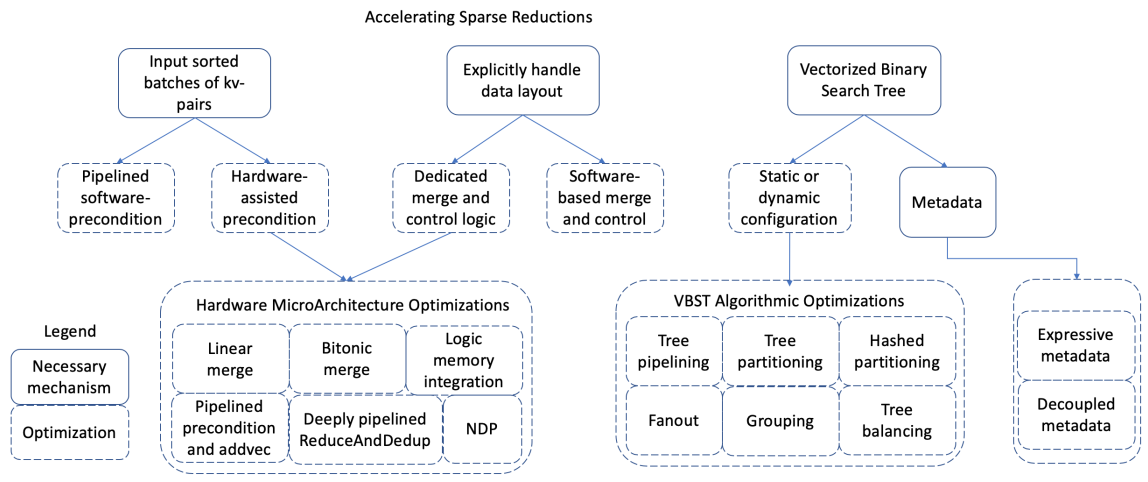

Accelerating Sparse Reductions
Input sorted batches of kv-pairs
Explicitly handle data layout
Vectorized Binary Search Tree
Pipelined software-precondition
Hardware-assisted precondition
Dedicated merge and control logic
Software-based merge and control
Static or dynamic configuration
Metadata
Hardware MicroArchitecture Optimizations
Linear merge
Bitonic merge
Logic memory integration
Pipelined precondition and addvec
Deeply pipelined ReduceAndDedup
NDP
VBST Algorithmic Optimizations
Tree pipelining
Tree partitioning
Hashed partitioning
Fanout
Grouping
Tree balancing
Expressive metadata
Decoupled metadata
Legend
Necessary mechanism
Optimization

Figure 4.17: Overview of necessary mechanisms and optimizations proposed in Chapters 3 and 4 to accelerate sparse data applications.

integrateable hardware-assisted data handling is vital to lowering data movement through the memory hierarchy. As a result, significant improvements to performance and energy efficiency are obtained for sparse data applications, that are otherwise difficult to accelerate in traditional systems.

CHAPTER 5
COMPUTATIONALLY ERROR-TOLERANT POST-MOORE PROCESSING

5.1 Ultra Low Power Device Technologies

Dynamic power scales proportionately with frequency and with the square of operating voltage, therefore, lowering V_{dd} is more beneficial.

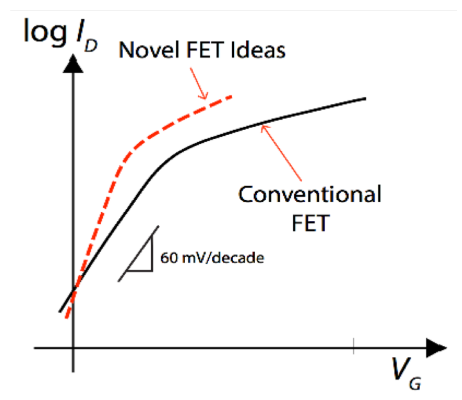
Although the theoretical lower limit for V_{dd} for inverter functionality is $2\frac{kT}{q}$ (or about $36mV$) [12, 13, 14], there are challenges to operating at this level. For barrier modulated transport, a fundamental physical limit exists, known as Boltzmann Tyranny. The following explanation borrows heavily from a recent PhD thesis in this area [11]. To increase the drain current I_D by an order of magnitude at room temperature, the gate voltage V_G needs to be increased by at least $k_B T ln 10 \equiv 60mV$, where k_B and T are the Boltzmann constant and the absolute temperature, respectively. The lower limit of the subthreshold swing S, defined as $\frac{\partial V_G}{\partial log_{10} I_D}$ is, therefore, 60 mV / decade. As a result, to maintain a good on-off ratio of the current, ˜1V needs to be applied at the gate. Despite significant engineering put into transistor design of "steep-sloping" devices, the limit of $2.3\frac{k_B T}{q}$ is a fundamental limit.

In order to overcome this limitation so that V_{dd} can be successfully lowered, two approaches to "re-invent" [144, 16, 145, 146] the transistor have been proposed:

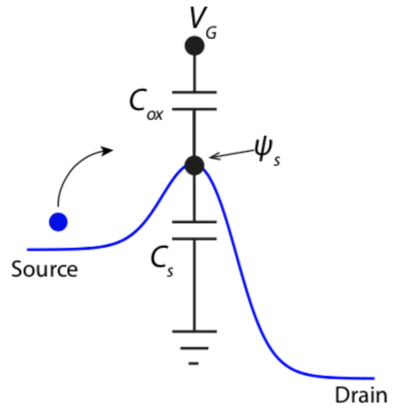
1. Change transport mechanism. Examples include band-to-band tunneling field effect transistors (TFET) [147, 148], impact ionization metal oxide semiconductor transistors (IMOS) [149] and nano-electro mechanical (NEM) switches [150, 151].

2. Change electrostatic gating such that the surface potential of the transistor increases beyond what is possible conventionally. Such active gating relies on the ability to drive the gate oxide to a non-equilibrium state such that its capacitance is negative.

Mathematically, $S = \frac{\partial V_G}{\partial log_{10} I_D} = \frac{\partial V_G}{\partial V_S} \frac{\partial V_S}{\partial log_{10} I_D}$, which can be abstracted to a first order



(**a**) Changing transport mechanism is a promising approach to going below the 60 mV/decade threshold.



(**b**) Negative capacitance can raise the surface potential above that of the applied voltage.

Figure 5.1: Next generation devices concepts [11].

into the expression $(1 + \frac{C_S}{C_{ox}}).(\text{“60”})$. In order to lower S below 60, the two approaches above seek to lower “60” or to cause $C_{ox} < 0$, respectively (Figure 5.1).

These next generation devices are fast switching even at few tens of millivolts, but as a result, are vulnerable to thermal noise perturbations. This translates into *intermittent, stochastic bit errors in logic*. With signal energies approaching the kT noise floor, future architectures will need to treat reliability as a first class citizen, by employing efficient computational error correction.

5.2 Computational Error Correction

5.2.1 Overview

We first classify various approaches to computational error correction into two broad categories:

1. *Temporal Redundancy*. This approach is based on the hypothesis that the probability of transient errors that occur at the same place to have temporal multiplicity is very low. In other words, a soft error occurs infrequently at the same device, and as such, repeated *measurements* in some manner would serve as an indicator to the correct

computation.

2. *Spatial Redundancy*. This approach is based on the hypothesis that the probability of multiple identical computations to all be in error at the same time is very low. In other words, by replicating a computation, any error in a small fraction of the replicas can be *masked* / overpowered by the other correct replicas.

These principles, it turns out, are fundamental to any sort of error correction including computation (ex. arithmetic), storage (ex. memory) and transmission (ex. networking). Some proposals favor spatial redundancy over temporal redundancy, some vice versa, and some employ both, depending upon the target fault model and environment.

Von Neumann [152] was among the first to propose using redundant components to overcome the effects of defective devices. He introduced the now widely used technique of Triple Modular Redundancy (TMR), which essentially uses three devices instead of one and uses a majority voter to infer a correct output. To note here is that such a mechanism can correct a single error (meaning that at least two of the three devices are not in error), or detect most double errors (where at least two devices are in error, and their outputs are non-identical). Even today, variants of such a (single error correct, double error detect) SECDED error model are in use.

Also proposed in his work were R-fold modular redundancy (RMR), which reduces to TMR when R=3; Cascade-TMR, which is essentially a multi-level TMR (ex. 3 sets of TMR modules and an overall voter) and NAND multiplexing. The latter replaces each processing unit with multiplexed units containing N lines for each input and output; the multiplex unit itself has two stages: executive stage (performs the function of the processing unit, in parallel) and restorative stage (reduces degradation caused by the executive stage; acts as a non-linear output amplifier). Years later, Nikolic et al. [153] provide a quantitative comparison of these techniques for fault coverage and area overhead necessary. In addition, they also evaluate a reconfiguration technique that uses Configurable Logic Blocks (CLBs) to create fixed function atomic fault tolerant blocks, and clustering them to achieve global fault tolerance.

While these techniques rank high in terms of robustness, they unfortunately, require a very significant overhead in terms of area and energy. It is intuitive that by trading performance, it should be possible to keep component overhead within reasonable limits to achieve reasonable reliability. Arithmetic codes are designed to do just that. We borrow the following classification of arithmetic codes from Wakerly [154].

1. *Separate vs Non-separate.*

   (a) *Separate:* The encoding of a datum X is the concatenation of X and a check symbol computed from X by the function C. The encoding of X is given by $f(X) =< C(X), X >$. Further, upon arithmetic transformation, the data and check symbols are non-interacting. For example, $f(X+Y) =< C(X) * C(Y), X+Y >$.

   (b) *Non-separate:* The encoding of a datum X is simply $C(X)$, with arithmetic transformation being obtained by a single binary operation on the encoding. For example, $f(X+Y) = C(X) * C(Y)$.

2. *Systematic vs Non-systematic.* For systematic codes, the encoding of a datum X has a subfield which equals X.

3. *Homomorphic vs Non-homomorphic.* For homomorphic codes, the data and check symbols are transformed using the same function.

In general, non-separate codes are generally non-systematic (although exception, i.e., systematic non-separate codes exist [155]), and that separate codes are also generally homomorphic (again, exceptions exist [154], however, multiplication is not supported and issues with implementation exist).

A typical example of a separate code is a residue code. In fact, it has been observed that all separate codes are equivalent to residue codes [156].

The remainder of this section is organized as follows. We first introduce a class of codes known as AN codes, then move onto summarize a large class of various parity predicting / circuit implementation dependent techniques and finally describe residue codes, which form the basis for the first part of this thesis.

5.2.2 AN codes

A typical example of non-separate codes is the AN code [157]. The check function constitutes a simple multiplication by a "check base" A:

$f(X) = AX; f(X+Y) = AX + AY$

As such, the code is valid under addition and subtraction, with possible error detection/-correction. The crux of such an algorithm relies on the observation that the sum/difference is also a multiple of A. Error detection is relatively straightforward as it just has to verify that the sum is divisible by A. However, error correction involves division by A and a series of subsequent subtractions and bit shifts, the count of which depends upon the number in question; resulting in a multi-variable-cycle latency for error correction, rendering it difficult to design efficient computers around. Years later, Liu [158] proposed a multi-bit error correction for the AN code, albeit at a much higher cost.

Failure to support multiplication notwithstanding, AN codes also run the danger of silent data corruptions due to the inherent possibility of undetected errors (for example, any erroneous number can pass as correct if it happens to be a multiple of A). For these two reasons, proposals to augment AN codes have been made. Forin [159] introduced static signatures (B) to augment the encoding as follows:

$f(X) = AX + B_X$, $0 < B_X < A$ can be arbitrarily chosen for each X.

As such, upon addition of numbers X and Y, the sum modulo A should be $B_X + B_Y$. To detect use of potentially stale (although correct) registers, a timestamping mechanism was further augmented, known as ANBD encoding:

$f(X) = AX + B_X + D$, where D indicates *version*.

Needless to say, for both ANB and ANBD coded systems, software support can be leveraged to hasten error detection by static assignments of signatures and pre-computation of their summations for stack variables. Further, recent work [160] extends this idea to employ signatures at the basic block level to verify dynamic control flow (including non-external function calls) wherever possible, by having instructions communicate counter values to a watchdog entity.

Yet other approaches suggest efficient encoding for multiplication and compiler techniques [161, 162]. However, no error correction scheme is proposed and fail-stop is the default mode of operation.

5.2.3 Micro-architectural / ISA independent techniques

A different category of 'codes' exists in that they are circuit dependent. Their main idea is to *predict* parity transformation with arithmetic operations and/or careful addition of spatio-temporal redundancy. This constitutes a relatively large body of work, and the following text strives to provide an intuitive treatment and coverage of such approaches.

Sun et al. [163] combine spatial redundancy and temporal redundancy to propose an error correcting parallel adder. They realize a Kogge Stone prefix tree using two Han-Carlson trees, and simply duplicate the generate/propagate circuitry. For elements off the resulting critical path, temporal redundancy is utilized. As far as adders are concerned, their approach is efficient as it is reported to correct 93.76% of soft errors with an area overhead of 12.23% and a delay overhead of 6.41%. However, no such scheme is presented by them for multipliers.

We now review the cost of several competing approaches in chronological order.

Johnson et al. [164] examine exploitation of spatial, temporal and hybrid techniques for the purposes of detection. They note that duplication with comparison incurs over a 100% area overhead. In certain cases, using alternating logic (using compliments) is more efficient *w.r.t.* hardware cost, however, making a function self-dual may require a 100% increase in area. Finally, for the purposes of temporal redundancy, recomputing with shifted/swapped operands involve negligible area overhead (extra hardware needed only to compute input encoding and output comparison). However, the latter two approaches incur over a 100% increase in latency. While these early implementation seem to be high-cost, especially given that no correction is performed, the ideas are fundamental and are refined over the years.

Hsu et al. [165] propose a temporal redundancy technique that utilizes spatial redundancy in a clever manner. TMR is emulated but hardware overhead is relatively lowered by using 2 x $\frac{n}{3}$ adders/multipliers and using temporal redundancy when an error is detected, to achieve correction. Their approach incurs a hardware overhead of 25% and a delay penalty of 108%.

Nikolaidis [166] propose efficient parity prediction techniques to achieve (detection only) low area overhead of 17% for carry lookahead adders. As noted by the residue based detection work of Pan et al. [167], Nikolaidis et al. propose [168] using differential logic circuits to implement each cell of array-based multipliers, and, also propose [169] output duplicated Booth multipliers, again, for detection alone. The latter was improved upon by Marienfeld et al. [170] to achieve a hardware overhead of 35% for detection in 32 bit multipliers.

Peng et al. [171] develop a mechanism wherein their adder stops upon error detection, and using a deadlock detector, reconfigure the adder. They are able to handle single faults in their 32 bit adders with an 81% area overhead, 140% for 2 faults and 211% for 3 faults. Vasudevan et al. [172] develop error detection in a carry select adder, with an area overhead of 20%.

Rao et al.[173] propose exploiting inherently redundant computation paths in carry generation blocks of carry lookahead adders to identify a faulty block. For the generator and propagator circuitry, time redundancy is used via rotated operands. Ghosh et al. [174] apply temporal redundancy to the Kogge-Stone adder based on the observation that even and odd carries are independent. In their two cycle addition technique, first, one of the correct set of bits (even/odd) are computed and stored at output register. Second, operands are shifted by one bit and the remaining sets of bits (odd/even) are computed and stored.

Rao et al. [175] further propose spatial redundancy for specific parallel prefix adders to achieve fault tolerance. Their design results in area overheads of 85%, 90% and 63% for Brent-Kung, Kogge-Stone and their hybrid implementations respectively.

Valinataj et al. [176] distribute TMR to protect carries alone, the premise being that correct carries are sufficient to generate correct output parities for both addition and multiplication. For a 32 bit carry lookahead adder and a 32 bit wallace tree multiplier, their technique requires an area overhead of 115% and 240% respectively. Krekhov et al. [177] propose parity prediction for the purposes of multi-bit error correction for addition, with roughly a 100% area overhead. Mathew et al. [178] propose parity prediction for multi-bit error detection and correction in Galois Field multipliers with over 100% of area overhead.

Keren et al. [179] observe that not all 2^k outputs of a k-bit output are generally valid outputs for a given combinational circuit. Instead of using redundancy bits, the input to the checker is created using the output bits and the input bits, assuming the function unit is implemented as two independent circuits. Based on the combinational circuits implemented on an FPGA by them, an average overhead of 85% in the number of LUTs was observed.

Dolev et al. [180] seek to transform Hamming codes with arithmetic, by generating codes for the fundamental NAND operation. The idea is to re-generate the code by performing correction on the input and then performing NAND and XOR. The XOR could be replaced with a BCH encoder for multi-bit error correction support.

Other orthogonal techniques

Banerjee et al. [181] outline an ASIC design flow where-in the application designer specifies which modules are critical and which can do with approximate outputs. This is especially relevant in DSP applications, for example, in FIR filters, certain coefficients are more important than the others. Post identification via a probabilistic analysis to determine potential fault locations, series transistors are added to mitigate potential shorts and parallel transistors are added to mitigate potential opens.

Blome et al. [182] propose using a small protected cache of live register values for better coverage than protected register files in that protecting the non-state logic (like read/write logic) of a storage structure is easier for smaller structures. As far as computation is concerned, time delayed shadow latches are used to leverage temporal redundancy.

Another class of computational error correction techniques is that of limiting redundancy to that of detection and using rollback to an error-free checkpoint to recover. Needless to mention, such a mechanism comes with its own set of trade-offs and challenges, some of which are: non-zero error detection latency, recovery latency, degree and stride of checkpoint placement etc.. We refer the interested reader to the techniques presented by Habkhi et al. [183] as a starting point for a discussion of these issues and related work in the area. Of further note is the use of checkpoint based recovery in speculative processors.

5.2.4 Summary of non-residue based codes

Standard error correcting codes (ECC) [184] have already been adopted into modern memory systems. These codes accommodate errors occurring in storage and communication/network traffic, but are not able to protect computational logic. The naive approach to computational error correction is triple modular redundancy (TMR) [152], requiring over a 200% overhead in area and energy for single error correcting/double error detecting (SECDED) capability. Several techniques in the form of arithmetic codes such as AN codes [157, 162, 159, 158, 160, 161], self-checking [164, 179, 170, 166, 169, 168, 172] and self-correcting [163, 165, 171, 180, 174, 177, 178, 175, 173, 176] adders and multipliers have since been devised. Orthogonally, there have been proposals that employ redundancy at a higher granularity, such as timing speculation (wherein error correction capability is limited to circuit timing violations) [185, 186], partial pipeline replication [187] or checkpoint-rollback-recovery such as those in IBM POWER6/7/8 and z10/196 [188, 189, 190, 191] processors, and various Intel Corporation [192] and Sun Microsystems mainframes [193]. While these are

Table 5.1: A (4, 2)-RRNS example with the simplified base set (3, 5, 2, 7, 11, 13). % is the mod operator. Range is 210, with 11 and 13 being the redundant bases. (Reproduced from [29].)

Decimal	% 3	% 5	% 2	% 7	% 11	% 13
13	1	3	1	6	2	0
14	2	4	0	0	3	1
13+14=27	(1+2)% 3=0	2	1	6	5	1
	All columns (residues) function independently of one another.					
	An error in any one of these columns (residues) can be corrected by the remaining columns.					

more efficient than naive TMR, they come with limitations on their error model, their area overheads are still over 100% and/or they incur a significant performance penalty [194] (e.g., owing to the fact that they leverage temporal redundancy in an effort to minimize area overhead).

5.2.5 Residue based codes

There exists a class of computationally error resilient codes based on the residue number system (RNS) [106], that are generally superior to the above techniques in terms of area, energy and latency overheads. The premise of RNS is that a number can be uniquely represented as a tuple of residues, where the residues are the remainder when the number is divided by a set of coprime bases/moduli. Each residue itself may be represented in a weighted radix representation in binary. An example is shown in Table 5.1. Assume the set of bases/moduli to be (3, 5, 2, 7); then, the number 13 can be uniquely mapped to the tuple (1, 3, 1, 6) as a result of the Chinese remainder theorem. Observe that (13 mod 3 = 1), (13 mod 5 = 3) and so on. Now, suppose we wish to add 13 (1, 3, 1, 6) to the number 14 (2, 4, 0, 0); this can be achieved by simply (modulo) adding the residues respectively to obtain 27 (0, 2, 1, 6). Observe that ((1 + 2) mod 3 = 0), ((3 + 4) mod 5 = 2) and so on. Critically, the computation occurs with no carries or interaction between residues.

Formally, let $B = \{m_i \in \mathbb{N} \; for \; i = 1, 2, 3, ..., n\}$ be a set of n co-prime natural numbers, which are referred to as bases or moduli. $M = \prod_{i=1}^{n} m_i$ defines the range of natural numbers that can be bijectively represented by an RNS system that is defined by the set of bases B. Specifically, for x such that $x \in \mathbb{N},\ x < M$, then, x can be represented as the following tuple: $(|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, ..., |x|_{m_n})$, where $|x|_m = x\ mod\ m$. Each term in this n-tuple is referred to as a residue. If necessary the value of x can be regenerated from the tuple of residues using a series of addition and table lookup operations via the Chinese remainder theorem, mixed-radix conversion, or macro-coefficient extraction.

Addition, subtraction and multiplication are closed under RNS, rendering the residues to be mutually independent *wrt* arithmetic. In other words, given $x, y \in \mathbb{N},\ x, y < M$, $|x\ op\ y|_m = ||x|_m\ op\ |y|_m|_m$, where *op* is any add/subtract/multiply operation.

In other words, RNS is closed under subtraction and multiplication operations as well, and they too can operate without interaction between residues. This important property has several useful implications:

- As the residues operate with no carries between them, they can operate in parallel. Furthermore, each residue operation's bit width is a fraction of that of the original operation. This translates into improved computational efficiency, which has been proven to be especially useful in digital signal processing (DSP) [195, 196, 197].
- A very large number can be losslessly represented and operated on via many smaller numbers (residues) in parallel. This property is used by the cryptography (RSA) community [198, 199, 200].
- Any bit error caused by a faulty computational logic element is guaranteed to be localized to within the corresponding residue, without impacting any of the other residues.

The last implication is of particular interest because it allows for a robust, efficient method for computational error correction. When redundant bases/moduli are introduced (Redundant RNS or RRNS) [201], the resulting redundant residues form an error correcting code that transforms itself automatically upon arithmetic operations, rather than having to be recomputed afterwards. If a corrupt residue results during an arithmetic operation, it is possible to infer its value using the remaining correct residues in the result. To continue our running example in Table 5.1, the number 13, being less than the product of the initial set of bases/moduli (3, 5, 2, 7), (i.e, 210), it can be uniquely represented using the 4-tuple (1, 3, 1, 6), via the Chinese remainder theorem. As such, these bases/moduli and residues are referred to as non-redundant. Upon adding two redundant bases/moduli, say (11, 13), the resultant redundant residues are therefore (2, 0). These redundant residues, by definition, are not necessary for representation, but provide a way to recover from errors. Given the (4, 2)-tuples for 13 (1, 3, 1, 6, $2, 0$) and 14 (2, 4, 0, 0, $3, 1$), a storage/transmission/computation error arising in any one of the residues of 13, 14 or their arithmetic manipulation result, can be corrected using the remaining residues. The running example is summarized in Table 5.1.

Formally, the set of moduli now contains n non-redundant and r redundant moduli: $B = \{m_i \in \mathbb{N} \text{ for } i = 1, 2, 3, ..., n, n+1, ..., n+r\}$. The reason these extra bases are redundant is because any natural number smaller than M $(= \prod_{i=1}^{n} m_i)$ can still be represented uniquely by its n non-redundant residues. Recall that the introduction of r redundant residues renders a computationally resilient *error code* because of the fact that all residues are transformed in an identical manner under arithmetic operations. For x such that $x \in \mathbb{N},\ x < M$, its RRNS representation is as follows: $(|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, ..., |x|_{m_n}, |x|_{m_{n+1}}, ..., |x|_{m_{n+r}})$, i.e., containing n non-redundant residues as well as r redundant residues.

We refer the interested reader to [201] for details of the multi-bit error correction operation, which involves addition, multiplication and table lookup operations. This correction capability increases with r, tolerating upto $\frac{r}{2}$ errant residues [202]. There are proposals to perform fractional multiplication [201] and to represent floating point numbers [203] using RNS. The key idea to extend this to RRNS is to protect the exponent and mantissa separately, as they transform differently upon arithmetic operations. For fault tolerance to work, certain conditions must be satisfied by B [201]. Given these, published work suggests that the most efficient conversion to *binary* algorithm (for $n = 4$, independent of r) nominally costs 8 cycles and is possible via macro-coefficient extraction [201]. More efficient conversion algorithms for RNS exist, such as those that use Mersenne primes (or other specific structural properties) as bases [204, 205, 206], but no known RRNS algorithms exist. A detailed treatment of these concepts is beyond the scope of this thesis.

The range for an RRNS representation is the product of its non-redundant moduli. Therefore, by choosing a non-redundant base/modulus set to be (199, 233, 194, 239) it becomes possible to represent a 32-bit integer in general in an RNS format, and extending it with a redundant set of (251, 509) allows for an RRNS representation. Such a (4, 2)-RRNS has the following advantages, over and above what RNS provides to us:

- Computational error correction can be achieved with a little over 50% area overhead.
- The SECDED granularity is that of a residue; meaning that such an RRNS is capable of *correcting multi-bit errors as long as they occur within a single residue*, or alternately, is capable of *detecting multi-bit errors as long as they occur within at most 2 residues*.
- Being closed under arithmetic, the correct value is *preserved* across a chain of dependent operations. Therefore, it is not necessary to incur the overhead of an RRNS error correction/detection after every operation.
- Due to the above, RRNS lends us robust computational error correction with relatively insignificant performance penalty, as is evidenced by this thesis.

Furthermore, there has been a significant body of research on RRNS that strives to make it more efficient algorithmically for error resilience [207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 202, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233] and division [214, 227, 234, 235, 236, 200, 237, 238, 239]. Although typically limited to detection, residue based logic protection (including that for floating point units and vectorized units) has widespread used in several commercial high-end server processors [188, 193, 190, 192, 191]. Clearly, the RRNS approach of computational error correction is ranked among the highest in terms of error correction capability and efficiency.

An efficient RRNS-based architecture is a viable compiler target for contemporary general-purpose as well as scientific computing applications, *in addition to being able to work efficiently with novel devices that operate close to the kT noise floor*. We strongly believe that single-thread processor performance scaling that has been stalled since the mid-2000s can be restarted via RRNS.

5.3 The CREEPY approach

5.3.1 Overview

Given new device concepts that enable device operation at signal energies close to the kT noise floor [16, 18, 21, 20, 19, 22], CREEPY (Computationally Redundant, Energy Efficient Processing for Y'all) aims to achieve lower energy consumption by lowering V_{dd} in such a manner that the intermittent errors that thereby arise are corrected efficiently.

We take note of the compute preserving properties of RRNS above and propose building a Turing-complete computer around this idea.

A CREEPY core consists of 6 *subcores*, an Instruction Register (IR) and a Residue Interaction Unit (RIU), as depicted in Figure 5.2.

Each subcore consists of an adder, a multiplier, a portion of the distributed register file and a portion of the distributed data cache. The bit-width of these components is same as that of their corresponding residue (8-bit or 9-bit in this example). Each subcore is fault-isolated from the other because it is designed to operate on a single residue of data (analogous to a bit-slice processor, with bolsters). Post a successful instruction fetch (the instruction cache stores instructions in binary, and is ECC-protected), the ECC-checked instruction is dispatched onto the 6 subcores, which then proceed to operate on their corresponding slice of data. For example, adding two registers is done on a per residue basis; the register file is itself distributed across the 6 subcores. Similarly, the data cache is also distributed across the 6 subcores and stores RRNS protected data. The RIU is then responsible to perform any operations that involve more than a single residue.

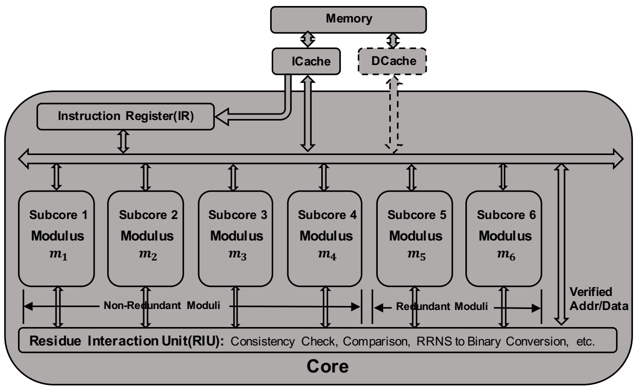


Figure 5.2: The CREEPY Core with the reference RRNS system. The register file and data cache are distributed across the subcores.

Because conversions to and from binary are expensive and rather unnecessary for RRNS data, a CREEPY core operates entirely on RRNS data and literals. An upshot of this is that control-path errors manifest themselves as data errors, meaning that they can be handled simply by handling the data error. For example, if there is an error in bypass logic in a subcore, or, if a faulty decoder in one of the subcores causes it to perform a multiplication instead of an addition, the resultant residue for that subcore would have an erroneous value, but can be recovered from the remaining 5 residues that were a result of the correct addition operation.

Although operating entirely on RRNS data and literals avoids the significant overheads of converting to/from binary, representing a memory address (for the purposes of PC, LD and ST) in an RRNS format naively may cause significant degradation in locality and changes memory access patterns, which is fundamental to memory systems performance. This issue is handled in Chapter 3, where we propose bit-manipulation techniques as well as a compiler based approach, with little to no overhead.

CREEPY employs standard ECC-protected main memory because of ECC's compactness and efficiency when it comes to protecting stored data. However, standard ECC isn't amenable to computational fault tolerance and therefore, the representation of data is in RNS form (as opposed to binary). The memory controller checks ECC on a processor load and generates the two redundant residues before loading the resultant RRNS data into the last level cache. Similarly, it generates ECC upon a processor store (and the redundant residues are not stored into the main memory). The exact choice of ECC is not relevant to this work; any of the existing schemes [184] may be used.

5.3.2 ISA

The description of CREEPY ISA is laid out in a manner similar to that of the MIPS ISA, for explanatory purposes. To simplify instruction fetch and decode, all instructions are of fixed length; 32 bits. The ISA expects 32 registers (R0-R31), with R0 hard-wired to zero, R30 being the link register and R31 storing the default next PC ($= PC + 4$). In our micro-architecture, each register is 49 bits long (*i.e.,* it contains the RRNS redundant residues as well) and is sliced on a per-modulus (sub-core) basis. The data cache is also implemented in a similar manner, as it stores data in an RRNS format.

1. **R-Format (ADD/SUB/MUL)**

   These instructions assume that the destination operand as well as both source operands are registers.

Opcode	Src Reg1	Src Reg2	Dest Reg	Reserved
6b	5b	5b	5b	11b

2. **I-Format (ADDI/SUBI/MULI)**

   For instructions that require compiler generated immediate literals, two new instructions (that always occur in succession without exception) are defined. Telescopic op-codes are employed to facilitate implementation of such *set* instructions. The fundamental need for the *set* instruction arises from the fact that literals are 49 bit RRNS values and would not otherwise simply fit within a 32 bit field (next to an immediate instruction, for example).

Set123 sets the the first 3 residues of the immediate value into the first 3 sub-core slices of the destination register and *Set456* sets the remaining 3 residues of the immediate value into the other three sub-core slices of the destination register.

Opcode	Dest	Reserved	Residue3	Residue2	Residue1
11[2b]	5b	0[1b]	8b	8b	8b

Opcode	Dest	Residue6	Residue5	Residue4
11[2b]	5b	9b	8b	8b

For an example, consider the immediate instruction *Addi R1, R2, 0x020202020202*. A CREEPY program would implement this instruction as follows:

(a) Set123 R3, 020202

(b) Set456 R3, 020202

(c) Add R1, R3, R2

3. **Branch**

Opcode	Reg1	Reg2	Reg3	Link	Reserved
6b	5b	5b	5b	1b	10b

Recall that R0 = 0, R31 = PC + 4 and that R30 is the link register. A CREEPY branch follows one of the following semantics:

(a) Reg1 = R0 and Reg3 = R0 and Link = 0: An unconditional branch that always jumps to the address in Reg2.

(b) Link = 0: A conditional branch that jumps to the address in Reg2 (base) + Reg3 (offset) if Reg1 is 0. This is otherwise known as a *beqz* instruction.

(c) Link = 1: A branch and link instruction to enable sub-routine calls and returns. The default next PC is stored into the link register and the program jumps to the address in Reg2.

4. **Load/Store**

Opcode	Reg1	Reg2	Reg3	Reserved
6b	5b	5b	5b	11b

Reg3 is the destination for a load and is the source register for a store. The source/destination address for a load/store is given by Reg1 (base) + Reg2 (offset). Note that the memory address is hereby stored in an RRNS format. Recall from Section 3 that efficiently handling RRNS addresses without conversion to binary is critical to application performance. Tradeoffs and methodologies in this space are handled by in Chapter 3.

5. **RRNS Check**

Opcode	Reg1	Reserved
6b	5b	21b

Reg1 is the register that needs to be checked. Once an error is detected, the system would try to correct it by utilizing RRNS error correction logic in the RIU. Candidate usage scenarios are discussed in Section 5.4.1 and evaluated in Section 3.5. Helper instructions such as *mov*, *ret* etc. also exist, but are omitted from this description for brevity.

5.3.3 Error Model

First, a distinction among fault, error and failure is made as follows:

Fault. A single bit flips, but is not stuck-at, *i.e.,* only intermittent / transient faults are considered. Causes may range from unreliable devices to low supply voltage to particle strikes to random noise and any combination therein.

Error. One or more faults in a single residue that show up during a consistency check.

Failure. Error uncorrectable and no recovery mechanism, or error undetectable.

Faults may lead to errors which may lead to failures. *We can guarantee the system is reliable if at most one error per core occurs between two RIU checks*. Multiple bit flips are rare but this phenomenon occurs if a circuit in the carry chain fails [188]. In our design, carry chains are limited to a residue as there are no carries between residues. Therefore, any resulting multi-bit errors would be localized to a single residue, which we can correct. If this RRNS system needs to detect and correct multiple error residues, an extra checkpoint and rollback mechanism is necessary. However, based on the discussion above, the case of multiple residues in error is extremely rare, and is therefore left to future work.

Redundancy in time, *i.e.*, check at cycle x, check again at cycle y, check again at cycle z, and vote, does not apply to this model as it is possible that the three checks suffer 3 independent 1 bit faults, rendering voting useless. The *transient* clause in the model rules out stuck-at faults. An implication of this is that one cannot achieve reliability by merely trading performance alone. Additional resources in terms of spatial redundancy are necessary, which is exactly what has been designed.

Different components of the core are protected via specialized means that target each component. The guiding principle is to design a system that uses the more efficient of RRNS/ECC based redundancy based on the range and nature of data being protected. Where both techniques are deemed insufficient to prevent the fault from metastasizing into an error, and eventually into a failure, the more conventional (and expensive) method: Triple Modular Redundancy (TMR), is employed. An alternative is to prevent the fault from occurring in the first place by using high V_{dd} (and/or circuit hardening). Choosing optimally between the latter expensive techniques is beyond the scope of this document but the assumption is that the RIU uses a high V_{dd} / hardened circuitry. It is further assumed that error in control signals manifest themselves as errors in data (for example, a control error causing one of the subcores to operate on the wrong opcode will be caught as a data error); however, one can potentially further improve the control signals' integrity by using either TMR or intelligent state assignment, and that the RIU uses a high V_{dd} / hardened circuitry.

5.4 Design Tradeoffs for an RRNS core

5.4.1 RRNS Check Insertion Strategies

Given that the CREEPY microarchitecture supports the error model outlined above, it is necessary to carefully insert RRNS_check instructions as they have a direct impact on the performance-energy-reliability metrics of the core. In this section, we outline the following check insertion schemes.

Periodic check

Insert a single check instruction after every n instructions. When $n = \infty$, this is an unchecked core and when $n = 1$, every instruction is checked. Note that lowering the value of n increases the check insertion frequency, raising performance overhead. While increased check insertion frequency typically provides increased reliability, one must be wary of the fact that the check instruction itself is of non-zero latency, meaning that, the longer the core spends in consistency checking, the longer it leaves its state vulnerable for errors to *creep* in. On the other hand, not checking every instruction also increases the probability of errors manifesting into multiple residues, leading to core failure.

Pipelined check

Insert a pipelined check that checks n instructions after every n instructions. This approach has the performance advantage of amortizing the latency of RRNS_check via pipelining as well as the reliability advantage of being able to increase state coverage of consistency check.

StateTable guided adaptive check

We define a bookkeeping entity known as $StateTable$ (described in the next Section) to maintain temporal information of the vulnerability of processor state. Whenever the probability of a register exceeds a certain threshold, an RRNS_check is inserted for that register. Naturally, this can be extended to insert pipelined checks if more than one register is in need of a check. This $StateTable$ itself is assumed to be an error-free entity that can either be implemented in software or hardware. Like the other two schemes, this insertion scheme can be implemented by the compiler or by the runtime (hardware or software); however, it is likely that utilizing a runtime component for this purpose would yield greater accuracy, which translates to improved efficiency and reliability, although subject to the overhead the $StateTable$ itself introduces.

Irrespective of the check strategy, we acknowledge that the following need to be error-free for correct execution; however, for the purposes of this simulation, we ignore their overheads/implications on control flow by assuming periodic checkpointing for potential rollbacks: (1) Effective address of each memory access (RRNS check), (2) Instruction contents (standard ECC check), and, (3) Main memory contents (standard ECC check). For a low-overhead checkpoint mechanism candidate, one can use an incremental checkpoint scheme to save energy and reduce storage overhead, when compared to using full checkpoints alone. The incremental checkpoints only record the modified entries from the last checkpoint (the last checkpoint could either be a full checkpoint or an incremental checkpoint). Once the rollback operation is necessary, the system can then use the last full checkpoint and the subsequent incremental checkpoints to recovery the machine state. A detailed trade-off analysis of the size, frequency, reliability and energy of such a scheme is beyond the scope of this thesis.

5.4.2 Multi-Domain Voltage Supply

The error distribution for each domain of a CREEPY core, *viz.*, computational logic, SRAM cells and RIU logic, are different. In an SRAM device, any fault occurring in one of its transistors gets latched, thereby resulting in an error. To contrast, glitches in logic transistors get masked if the glitch does not occur close to the clock edge. Also, to avoid having to 'check a check instruction', we assume the RIU logic is error-free protected via TMR, hardened logic, and/or higher signal energies, with the latter sufficient to model the energy effects of the former. Given that the vulnerability of these domains increases from computational logic to SRAM cells to RIU logic, it is inefficient to assume a uniformly high signal energy across these domains. We model this phenomena by independent voltage rails for each of these domains. Shimazaki [240] and Rusu [241] proposed some multi-voltage domain designs. The voltage domains referred to in CREEPY are coarse-grained (module-based), rendering the implementation feasible.

Orthogonally, index-sum multiplication has been proposed in the past [242] to achieve multiplication via simple addition and table lookup operations, thereby rendering it more efficient than traditional binary multiplication, provided the size of the LUT is not too large. The principle is analogous to using a *logarithm* operation, i.e., a multiplication can be achieved via a table lookup, addition and a reverse table lookup. The size of the LUT for a 32-bit input is very large, however, RNS properties allows it to be sliced into significantly smaller tables. The results presented later in this chapter therefore include the index-sum multiplier as well. However, the details of this technique are beyond the scope of this thesis, and can be obtained instead directly from our publication [30]. Similarly, specific details of the RIU algorithms, such as single error detection and correction algorithm, signed number representation, overflow detection, comparison and correction factors for arithmetic are omitted from this thesis and the interested reader may view our publication instead.

5.5 Stochastic Simulation Model

To measure the performance-energy-reliability trade-off of a CREEPY core, a stochastic fault injection mechanism is augmented into a cycle-accurate in-order trace-based simulator. This work abstracts the notion of using next-generation devices operating at low signal energies (E_s) and the resulting interaction with the kT noise floor into P_e, the probability of an error occurring in a transistor state in any given cycle. E_s, provided as an input to the simulation, is a measure of the signal energy at the input of a transistor; P_e is the probability of a fault occurring at the output of a transistor in any given cycle. The relationship of E_s and P_e can be defined by the following relation: $P_e = exp(\frac{-E_s}{kT})$ [243]. From Section 5.4.2, these inputs are vectors as they denote the signal energies and error probabilities for each voltage domain, however, for explanatory purposes, the text presents them as scalars for the remainder of this section. Also input to the simulator is the check insertion strategy, as discussed in Section 5.4.1. Because this work evaluates a very different number system, an unpipelined microarchitecture was simulated with no branch prediction and a 2-level memory hierarchy (LLC-DRAM, with latencies of 12 cycles and 100 cycles for LLC hit and miss respectively) to restrict the primary focus of this chapter.

We first introduce a series of error events and their probabilities.

P_e Probability of an error occurring in a transistor state in any given cycle. This is provided as an input to the simulation, as just discussed.

P_{add} Probability of at least a single error in an adder (each sub-core has an adder). If there are N_{add} transistors in an adder, the probability of each of these transistors being free of error is $(1 - P_e)^{N_{add}}$. Therefore, P_{add}=1-$(1 - P_e)^{N_{add}}$. Similarly, P_{sub} and P_{mul} are calculated. For multi-cycle operations, this definition holds as long as the state of each transistor is used exactly once for the operation. This is true for the said operators. Note that this is a conservative (pessimistic) estimate in our evaluation because this work ignores any error masking that may potentially occur.

P_{R_i} Probability of at least 1 error being present in a slice (sub-core/residue) of register R_i since its last write. To compute this, a $StateTable$ is devised, the i^{th} entry of which holds the tuple (P, $cycle$), where, P is the probability of R_i having atleast 1 error being present in the corresponding residue upon its most recent update at cycle $cycle$. This $StateTable$ is updated for each register write.

For example, consider the register R_0. 1) At cycle 0, the default value of R_0 tuple is (P=0, cycle=0). 2) At cycle 10, assume that there is an ADD instruction: ADD R0, R1, R2, and that it is the first instruction writing to R_0. The tuple value is then updated to (Error_Probability_ADD, 10). It is necessary to update the P value here because the error probability of this ADD instruction should be taken into account. P value would then be set back to 0 once an RRNS check is inserted for that register and no error is detected, and then set the current system cycle value to the cycle field. This way, the P field in the $StateTable$ always reflects the probability of that register of having at least 1 error being present in one of its residues, given its most recent update at the cycle field.

Assuming an SRAM implementation of 8-bit wide R_i, the number of transistors is $8 \times 6 = 48$. The probability of R_i being error free is subject to two probabilities: (1) probability of an error-free write, ($P_1 = 1 - StateTable[R_i].P$) and, (2) probability of no error creeping into it since its last write ($P_2 = (1 - P_e')^{48(c - StateTable[R_i].cycle)}$), where, c is the current cycle and P_e' is the probability of an error occurring in the state of an SRAM transistor. Due to the nature of an SRAM device, any fault occurring in one of its transistors gets latched, resulting in a higher probability of an error (when compared with glitches in logic transistors getting masked if the glitch does not occur close to the clock edge). As such, it is assumed that $P_e' = 100P_e$. Putting it all together, $P_{R_i} = 1 - P_1 * P_2$.

$P_{LOAD\ X}$ Probability of at least 1 error being present in the loaded data of address X. This is analogous to P_{R_i}, with the extended $StateTable$ storing an entry for each cache line. As this work assumes a perfect off-chip (ECC protected) main memory, cache miss repairs are initialized with a zero probability in error, and cache replacement victims' entries are evicted from the $StateTable$. Finally, $P_{LOAD\ X}$ encapsulates the probability of an error in the implicit computation of the address X itself (from its base and offset) during the execution of the load, in addition to the probability of an error in the loaded data from the cache line.

P_{SC} Probability of at least 1 error occurring in a sub-core from the last time it was checked. To illustrate, consider the following add instruction: $ADD\ R_3, R_2, R_1$. Then, at the end of instruction, $P_{SC} = 1 - (1 - P_{add})(1 - P_{R_2})(1 - P_{R1})$.

P_C Probability of exactly 1 error occurring in a CREEPY core from the last time it was checked. This translates to exactly 1 sub-core being in error (where the sub-core error itself may be of multi-bit form; RRNS can tolerate multi-bit flips within a single residue). Therefore, $P_C = 6C_1 \times P_{SC}(1 - P_{SC})^5$, where the combinatorial choose operator nC_r enumerates the number of ways in which r items can be chosen from n distinct items.

P_C^0 Probability of no error in a CREEPY core from the last time it was checked. $P_C^0 = 6C_0 \times (1 - P_{SC})^6 = (1 - P_{SC})^6$.

P_C^{fail} Probability of a CREEPY core failing at any given cycle, since the last time it was checked. The current version of the CREEPY micro-architecture is unable to correct more than 1 error occurring in the core, and assumes a recovery mechanism such as checkpointing is in place. As such, this work deems ≥ 2 errors in the core as amounting to a failure. Therefore, $P_C^{fail} = \sum_{2 \leq r \leq 6} 6C_r \times P_{SC}^r (1 - P_{SC})^{6-r} = 1 - P_C^0 - P_C$.

Note that the computation of these error probabilities is done after every instruction (irrespective of the check insertion strategy) for the purposes of bookkeeping such as $StateTable$ update and to estimate the probability of a failure $P_{C,i}^{fail}$ at each time step t_i. This work uses a typically used reliability metric, Mean Time Between Failure (MTBF) [244], which can be defined as follows: $MTBF = \frac{Total\ Cycles}{CPU\ Frequency \times \sum_i P_{C,i}^{fail}}$. The subscript i in $P_{C,i}^{fail}$ represents the i^{th} instruction of the instruction stream. MTBF also corresponds to mean time to checkpoint recovery.

5.6 Results from RRNS Core Simulations

5.6.1 Signal Energy Limits

From an independent set of simulations of a non-error-correcting core operating on binary data, it is found that the minimal signal energy required for ensuring its reliable operation is $48kT$. In the first design of an error correcting RRNS core [29], a single voltage domain was assumed across computational logic, SRAM cells and RIU logic. Together with a traditional multiplier (i.e., without index-sum), a pipelined check insertion strategy (with a frequency of 5 instructions) and a 16MB LLC, the result is that one can tolerate gate signal energies of $42 - 43kT$, as shown in Figure 5.3.

However, given the dissimilarity in error distribution across computation, SRAM and RIU (Section 5.4.2), independent voltage domains are considered for these. For simplicity, the RIU gate signal energy is conservatively set to be $48kT$ (i.e., same as that required for a non-error correcting binary core), although it can be potentially lowered as its functionality is a subset of that of a binary core. It is found that the relative impact of energy savings in the RIU is rather limited (Section 5.6.4), and therefore restrict the RIU gate signal energy to $48kT$ in our evaluations.

We abstract these voltage domains as a triplet; for example, $30 - 43 - 48$ denotes the gate signal energy for computational logic to be $30kT$, SRAM cells to be $43kT$, and RIU logic to be $48kT$. For the purposes of this evaluation, a target MTBF of $1E + 10$ seconds (over 300 years) is assumed and it is found that the gate signal energy for computational logic can be lowered all the way to $28 - 31kT$, depending upon the benchmark, as shown in Table 5.2.

Given these minimum signal energies, the performance, efficiency and reliability of various core configurations are evaluated in the following sub-sections. The core configurations presented are as follows:

Binary A non-error-correcting core operating on binary data. This is the baseline and

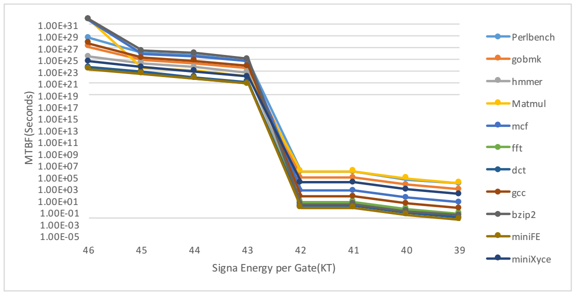


Figure 5.3: Reliability when a single voltage domain is used.

Table 5.2: Sensitivity of MTBF (seconds) to benchmarks and gate signal energies of computational logic, SRAM cells and RIU logic. For example, 30-43-48 denotes the gate signal energy for computational logic to be $30kT$, SRAM cells to be $43kT$, and RIU logic to be $48kT$.

Benchmarks	**36-43-48**	**35-43-48**	**34-43-48**	**33-43-48**	**32-43-48**	**31-43-48**	**30-43-48**	**29-43-48**	**28-43-48**	**27-43-48**
perlbench	1.70E+17	2.29E+16	3.10E+15	4.20E+14	5.69E+13	7.70E+12	1.04E+12	1.41E+11	**1.91E+10***	2.62E+09
gobmk	1.07E+17	1.44E+16	1.95E+15	2.64E+14	3.58E+13	4.85E+12	6.55E+11	8.87E+10	**1.22E+10***	1.97E+09
hmmer	4.08E+16	5.53E+15	7.48E+14	1.01E+14	1.37E+13	1.85E+12	2.51E+11	**3.40E+10***	6.23E+09	9.69E+08
matmul	1.08E+16	1.47E+15	1.99E+14	2.69E+13	3.64E+12	4.93E+11	6.66E+10	**9.02E+09***	1.22E+09	2.76E+08
mcf	1.81E+17	2.44E+16	3.31E+15	4.47E+14	6.06E+13	8.20E+12	1.11E+12	1.50E+11	**2.03E+10***	2.80E+09
fft	7.63E+14	1.03E+14	1.40E+13	1.89E+12	2.56E+11	**3.47E+10***	4.69E+09	6.35E+08	1.86E+08	2.32E+07
dct	1.33E+15	1.80E+14	2.44E+13	3.30E+12	4.47E+11	6.05E+10	**8.18E+09***	1.11E+09	3.11E+08	4.02E+07
gcc	1.41E+17	1.91E+16	2.59E+15	3.50E+14	4.75E+13	6.42E+12	8.68E+11	1.18E+11	**1.60E+10***	2.37E+09
bzip2	1.73E+17	2.34E+16	3.17E+15	4.29E+14	5.81E+13	7.86E+12	1.06E+12	1.44E+11	**1.95E+10***	2.65E+09
miniFE	6.94E+14	9.39E+13	1.27E+13	1.72E+12	2.33E+11	**3.15E+10***	4.26E+09	5.77E+08	1.69E+08	2.11E+07
miniXyce	1.09E+16	1.47E+15	1.99E+14	2.69E+13	3.64E+12	4.93E+11	6.66E+10	**9.02E+09***	2.04E+09	3.06E+08

* These signal energies are used for the remainder of this work, as they render reasonable reliability.

requires signal energies of at least $48kT$ in order to achieve reasonable reliability.

RNS A non-error-correcting core operating on RNS data. In other words, an RRNS core without redundant subcores and error correction capabilities.

RRNS_pipe5 An error correcting RRNS core with a pipelined check insertion strategy (with a frequency of 5 instructions), as was determined as the most optimal strategy in the first RRNS core design [29].

Index-sum_pipe5 Similar to RRNS_pipe5, except that the traditional multiplier is replaced with an index-sum multiplier.

RRNS_Adapt_1e-9 An error correcting RRNS core with an adaptive check insertion strategy (with an error probability threshold of $1e-9$. It was found that $1e-9$ was the optimal threshold obtained via simulation for target MTBF/signal energy).

Index-sum_Adapt_1e-9 Similar to RRNS_Adapt_1e-9 that uses an index-sum multiplier.

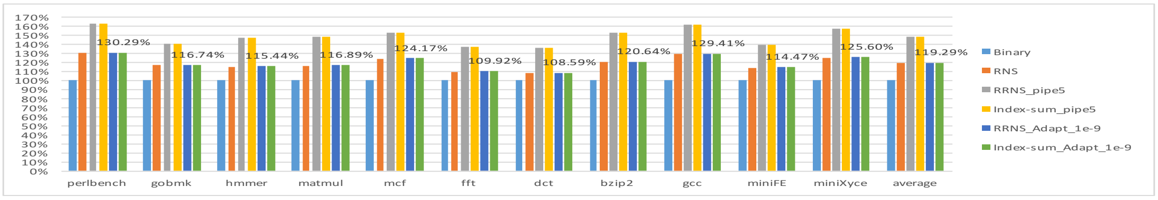


Figure 5.4: Performance of various core configurations, normalized to an non-error-correcting binary core

5.6.2 Performance

Figure 5.4 presents the performance of various core configurations listed in Section 5.6.1, normalized to that of a non-error-correcting binary core.

There is an inherent performance degradation in running binary-optimized code on an (R)RNS-based core because position-based bit manipulation techniques are expensive in (R)RNS, however, this is limited to about 20% on average. Introducing error correction may further degrade performance if naive or static check insertion strategies are used. The overhead due to error correction is amortized when the check insertion strategy is adaptive instead.

5.6.3 Energy

The primary concern of CREEPY core design is reducing the core energy overhead. Figure 5.5 shows the normalized energy consumption of the aforementioned configurations.

The non-error-correcting binary core requires high gate signal energies in order to be reliable. Given the low-bit-width and carry-free nature of RNS arithmetic, RNS based cores are inherently more energy efficient than their binary counterparts. When *efficient* error correction is introduced, further energy savings can be achieved as the supply voltage can be turned down while still maintaining reliable functionality. Adaptive check insertion strategy ensures that the overhead of error correction is minimal. Finally, using index-sum multipliers enables further energy savings as they are more efficient than traditional multipliers (savings of over $3\times$ for multiplication intensive benchmarks and over $2.3\times$ on average).

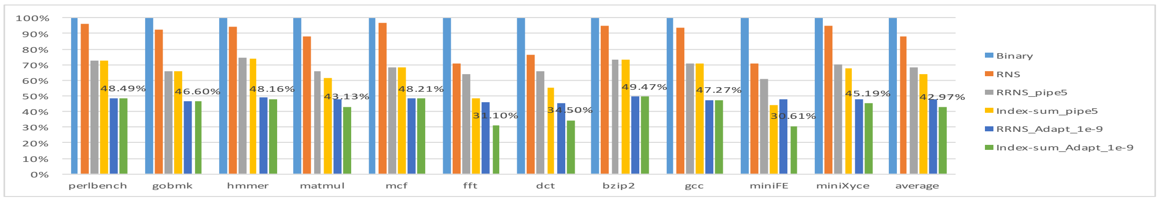


Figure 5.5: Energy Comparison for Different Strategies

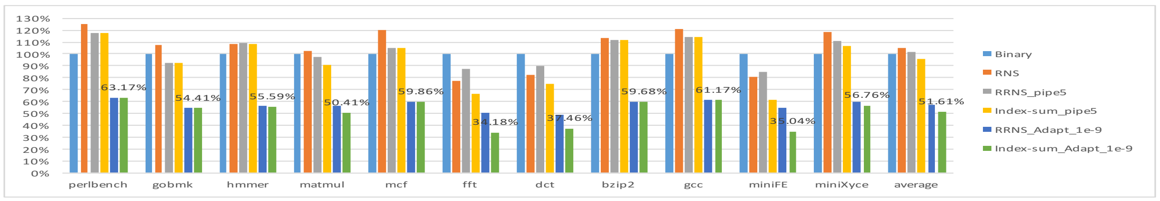


Figure 5.6: EDP Comparison for Different Strategies

Figure 5.6 shows the Energy Delay Product (EDP) of these core configurations, normalized to that of a non-error-correcting binary core. With the exception of arithmetic intensive workloads, RNS cores typically have a higher EDP than binary cores. However, via efficient error correction, our RRNS cores show significantly improved EDP. Specifically, by utilizing our best optimization scheme (index-sum multiplier and adaptive check insertion), one obtains EDP benefits of about $2\times$ on average, or about $3\times$ for multiplication intensive workloads.

5.6.4 Potential of RIU Optimizations

As described in Sections 5.4.2 and 5.6.1, this work conservatively chooses the gate signal energy for RIU logic to be that necessary for reliable operation of a Turing complete non-error-correcting binary core, i.e., $48kT$. One of the reasons for this is to side-step the issue of 'checking the checker'. However, if one were to deploy self-checking logic or some other optimizations in the RIU, it may no longer be necessary to use a high voltage supply for the RIU domain. In this limits study, 3 possibilities of the gate signal energy to RIU logic are evaluated: *Binary* - $48kT$, *Computation* - same as that of RRNS subcore computational logic, *Zero* - $0kT$. From an Amdahl's law perspective, it is found that optimizing RIU logic has limited impact on core energy, as shown in Figure 5.7, thanks to our judicious RIU

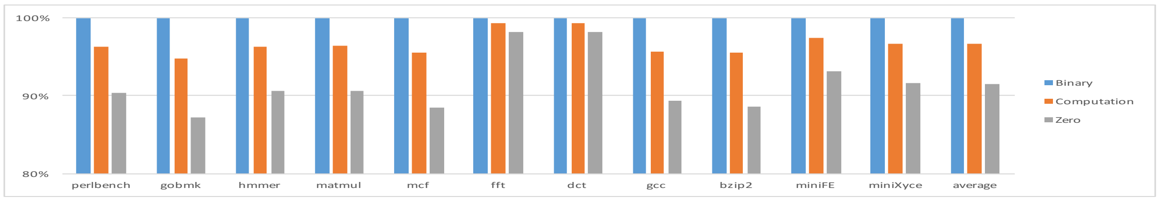


Figure 5.7: The Potential of RIU Energy Optimization

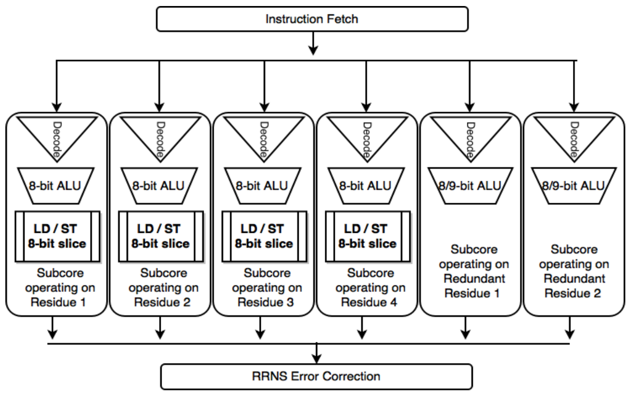

Instruction Fetch
Decode
8-bit ALU
LD / ST 8-bit slice
Subcore operating on Residue 1
Subcore operating on Residue 2
Subcore operating on Residue 3
Subcore operating on Residue 4
8/9-bit ALU
Subcore operating on Redundant Residue 1
Subcore operating on Redundant Residue 2
RRNS Error Correction

Figure 5.8: Generic high level schematic of an RRNS error correcting compute core with 4 non-redundant *sub*-cores and 2 redundant *sub*-cores.

usage via adaptive check insertion.

5.7 Review of the anatomy of an RRNS core microarchitecture

For improved readability, this section reviews the details of the RRNS core introduced above, from the perspective of memory system design.

Figure 5.8 depicts a generic schematic of an RRNS core with 4 non-redundant *sub*-cores and 2 redundant ones. This schematic is similar to prior RRNS core microarchitecture proposals [30, 201]. Each *sub*-core is associated with exactly one base modulus and thereby operates on its own residue, with the register file and highest level data cache being distributed into the 6 subcores on a per-residue basis.

Error model. Recall that such a core operates entirely on RRNS data and literals, meaning that there are no unnecessary (and expensive) conversions to and from *binary*. Therefore, all memory addresses (PC, LD/ST) are in RRNS form, including any pointer arithmetic. Another upshot of operating entirely on RRNS data is that control-path errors manifest themselves as data errors, meaning that they can be handled simply by handling the data error. For example, if there is an error in bypass logic in a subcore, or, if a faulty decoder in one of the subcores causes it to perform a multiplication instead of an addition, the resultant residue for that subcore would have an erroneous value, but can be recovered from the remaining 5 residues that were a result of the correct addition operation. Finally, as instructions themselves don't undergo modification, ECC is sufficient to protect them. The architecture proposed in Chapter 2 is able to ensure reliable operation as long as at most one subcore is in error (possibly multi-bit) between two error correction operations. Circuit-hardening or using high V_{dd} for the error correction logic is proposed to ensure its reliable operation. Because of latching effects, SRAM transistors have different (more prone to errors) reliability characteristics when compared to logic, and as such, are modeled with $100\times$ the error probability of logic transistors.

Overheads. In terms of area, such an RRNS error correcting core requires $2\times$ the area of a traditional non-error-resilient core. However, a large fraction of this overhead is from LUTs necessary in the error correction operation; a (4, 1)-RRNS core that simply detects errors is in fact 34% smaller in area when compared to a traditional core. In an independent study that implements the error model above (but ignores memory addressing inefficiencies), we found that the overhead due to slow comparison operations, boolean operations as well as RRNS error correction operations resulted in an overhead of just about 20% in runtime when compared to a traditional, non-error-correcting core over general purpose (SPEC2006) as well as computationally intensive workloads (such as FFT, matrix multiplication etc.). However, RRNS enables lowering of signal energy to few tens of millivolts, and results in about a $2\times$ improvement in energy-delay-product, in spite of these overheads. The engineering details of this study are provided earlier in this chapter.

Abstractions for the memory interface. The presence of a Load/Store unit in the redundant *sub*-cores is implementation dependent. For the purposes of this thesis, we assume that the core is responsible for generating *valid* memory requests in the Memory Address Register (MAR). This is possible by either (a) inserting an RRNS consistency check

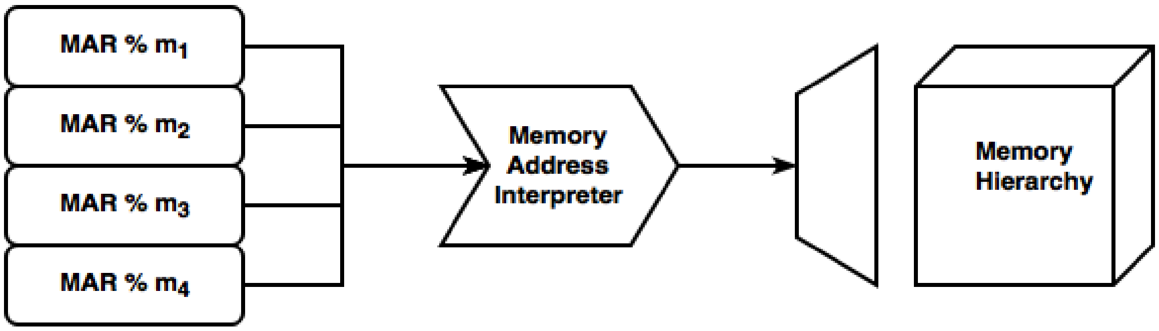


Figure 5.9: An (R)RNS compute core natively generates memory addresses in the 4-residue tuple format. This is depicted by MAR $\%$ m_i in the figure, where m_i is the i^{th} base modulus, $\%$ is the modulo operator, and MAR could be one of Program Counter (PC), Load or Store address.

before a memory access, or (b) by enabling a checkpointing mechanism for rollback/recovery in case a memory request to an illegal location is generated. For the latter, a segmentation scheme can be used to flag ”wrongly” computed addresses as illegal, thereby triggering checkpoint recovery. As we show in Section 5.9.2, minor perturbations in the native *rns_concat* representation of a memory address causes its value to fluctuate wildly, thereby allowing for such segmentation to work. Note that the consistency check of (a) or the segment check of (b) can be done in parallel with a memory access, and are therefore not on the critical path of the program. Therefore, we omit the Load/Store units in the redundant subcores. Finally, the addressing logic in the memory hierarchy (such as a decoder or an address translator) is assumed to either utilize devices that do not stochastically flip due to thermal noise, or employ self checking logic [245, 246]. Relaxing either assumption reveals a set of implementation dependent tradeoffs and are beyond the scope of this thesis. *Without loss of generality, this thesis assumes that no explicit error correction is required for such addressing logic in the memory hierarchy*. It follows that the redundant residues can be dropped from the MAR, and that a 32-bit address can be logically represented as a 4-tuple RNS number.

5.8 Need for Memory System Design for Computationally-Resilient Post-Moore Processors

While RRNS is clearly attractive for designing ultra-low-power, computationally error resilient microarchitectures, prior work on RRNS has abstracted away the memory hierarchy for simplicity. Thus, an RRNS microarchitecture hits a performance bottleneck when connected to non-ideal, real world memory systems. Memory is addressed in binary in conventional processors, whereas an RRNS compute core natively generates memory addresses as a tuple of residues. There are two naive approaches to this memory interface problem:

Binary. Convert the tuple-of-residues format to binary and address memory in binary as usual. This approach imposes a severe latency and energy penalty on every instruction fetch, load and store operation. This overhead is due to the multi-step nature of each RNS to binary conversion, which, nominally costs 8 cycles in the form of add and table lookup operations. According to our results, this slowdown is $3\times$ on average for in-order cores and $2\times$ for out-of-order (OoO) cores.

Rns_concat. Another straightforward, although naive approach is to concatenate the native tuple-of-residues and use the result as the memory address. Unfortunately, this technique destroys spatial locality of sequential memory accesses, rendering caches largely ineffective and causing application slowdowns of over $3\times$ on average for in-order cores and $4\times$ for OoO cores.

There are other overheads associated with RRNS, such as non-trivial comparison and boolean operations. However, the overheads due to memory addressing inefficiencies are significantly higher. The memory hierarchy is accessed for *each* instruction (PC) in addition to memory instructions (LD/ST). In contrast, comparison and boolean op instructions are less frequent, even if a targeted code generator does not minimize such ops. Furthermore, even with a naive implementation, the penalty for such ops is less than that of a cache miss. In an independent study (that ignores memory addressing inefficiencies), we found that the impact of slow comparison, boolean operations as well as consistency checking operations due to RRNS makes programs run just about 20% slower than a traditional core. Yet, due to RRNS, we realize significant energy savings, rendering energy-delay-product benefits of about $2\times$ when compared to traditional non-error-correcting cores, in spite of these overheads.

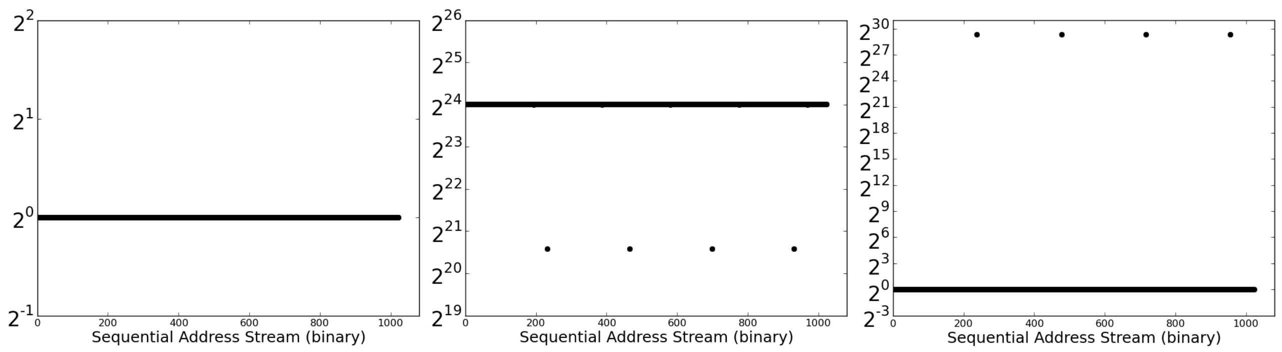
In spite of its strong potential to restart single thread performance scaling, an RRNS processor is not competitive with traditional designs due to memory addressing inefficiencies outlined above. The energy savings would be overshadowed by the decrease in memory system performance. This chapter presents the first study of its kind to our knowledge to focus on the RRNS memory access problem.

5.9 Memory Addressing Schemes

As noted in Section 5.7, the memory address generated by an RRNS compute core in its native form is in a tuple-of-residues format, where the address itself may be to instructions or data. As depicted in Figure 5.9, such a 4-tuple must be properly interpreted before the memory hierarchy can be accessed. This section presents a basis set of possible *interpretations* of an address.

5.9.1 Interpretation Schemes

Binary. The approach assumed by prior work was to convert the 4-tuple of residues to its binary representation and use this as a memory address. From this point on, the system can access the memory hierarchy as conventional computers do. However, this conversion from RNS to *binary* doesn't come for free, and the cost must be paid for each memory access, both cache hits and cache misses. This overhead is a multi-cycle access time increase and an associated increase in energy consumption. This is because conversion from RNS to *binary* is non-trivial: it is a multi-step operation, involving addition and table-lookup operations,



(a) Delta between consecutive addresses: *binary* scheme. **(b)** Delta between consecutive addresses: *rns_concat* scheme. **(c)** Delta between consecutive addresses: *rns_sub* scheme.

Figure 5.10: Spatial locality analysis via visual comparison of the 3 addressing schemes for a sequential stream of addresses by computing the *deltas* of *interpreted* addresses of consecutive addresses of the sequential stream.

and costs 8 CPU cycles nominally. Our experiments indicate that applications suffer from a slowdown of $3\times$ on average for inorder cores and $2\times$ for OoO cores upon using such a naive conversion approach.

$$\{r_1, r_2, r_3, r_4\} \implies binary(\{r_1, r_2, r_3, r_4\})$$

This conversion typically is not a one-time cost as addresses may repeat themselves (locality). As a solution, this work proposes to cache the conversion itself in a Conversion Lookaside Buffer (**CLB**), in a manner similar to how a TLB caches translations.

Rns_concat. On the other extreme is a lightweight interpreter that merely concatenates the 4-tuple, with the resulting 32bit number being treated as the address to the memory hierarchy. More concretely, the 4-residue tuple $\{r_1, r_2, r_3, r_4\}$ is treated as the bitstream $r_1r_2r_3r_4$.

$$\{r_1, r_2, r_3, r_4\} \implies r_1r_2r_3r_4$$

Rns_sub is a scheme similar to *rns_concat*, but the lowest residue is subtracted from the 3 other residues in an attempt to preserve locality.

$$\{r_1, r_2, r_3, r_4\} \implies rns\_concat((r_1 - r_4)\%m_1, \\ (r_2 - r_4)\%m_2,\ (r_3 - r_4)\%m_3,\ r_4)$$

where, the RNS base moduli are given by m_i. This scheme is significantly less expensive than *binary* and is slightly more expensive than *rns_concat*.

5.9.2 Sequential Address Analysis Example

Figure 5.10 shows a comparison of how the 3 addressing schemes discussed so far, *binary*, *rns_concat* and *rns_sub*, remap a stream of sequential accesses into the memory address space. On the X-axis is the input value to the Memory Address Interpreter; on the Y-axis is the difference (*delta*) between two consecutive *interpreted* memory addresses.

First, notice that the *delta* for *binary* is always exactly 1; converting the tuple-of-residues back to the binary number they represent results in sequential memory addresses. However, *rns_concat* remaps accesses according to the following function: if the address X $\equiv\{r_1$, r_2, r_3, $r_4\}$, then $X + 1$ $\equiv\{r_1 + 1$, $r_2 + 1$, $r_3 + 1$, $r_4 + 1\}$, thereby resulting in a *delta* of 0x01010101 as each residue is 8-bits wide. This is the *delta* value that is seen in the figure as a straight horizontal line slightly above 2^{24}. However, each time a residue overflows, the above constancy claim for *delta* breaks down, generating the discontinuities shown in the plot.

Non-unit delta values will cause consecutive accesses to touch *different* cache lines and memory pages, destroying spatial locality in the access stream. To mitigate this constant *delta* offset seen in *rns_concat*, this work defines *rns_sub* in an effort to preserve locality. Applying *rns_sub* to X and 1, observe the following pattern (ignoring modulo overflows):

$$X \equiv \{r_1 - r_4, r_2 - r_4, r_3 - r_4, r_4\};\ \ 1 \equiv \{0, 0, 0, 1\}$$

$$\implies X + 1 \equiv \{r_1 - r_4, r_2 - r_4, r_3 - r_4, r_4 + 1\}$$

Therefore, *rns_sub* yields a *delta* value of exactly 1 in the common case, similar to the *binary* scenario. In general, the *rns_sub* representation of any number $n < m_4$ is $\{0, 0, 0, n\}$, meaning that difference between $X + n$ and X in the *rns_sub* representation is exactly n in the common case. This means that *rns_sub* is capable of preserving the spatial locality that cache and DRAM memory systems need to deliver performance.

While the detailed evaluation of a prefetcher is beyond the scope of this work, intuitively, prefetchers that work well with *binary* would work well with *rns_sub* as well. With *rns_concat*, however, a stride of 1 must be re-interpreted as a stride of 0x01010101, which is the *delta* between consecutive addresses, significantly altering the operation of most prefetchers.

Although our example analysis in this section is based upon a sequential address stream, the insights are applicable to arbitrary streams that exhibit spatial locality, as is demonstrated via simulation results detailed below.

5.10 Memory System Design Tradeoffs

Utilizing a basis set of address interpretation schemes outlined in Section 5.9, one can now construct and analyze a design space consisting of memory access granularity, Memory Address Interpreter (MAI) configuration and DRAM address interleaving dimensions.

5.10.1 Memory Access Granularity

As discussed in Section 5.9, with *rns_concat* based schemes, consecutive addresses do not map onto contiguous memory locations. This causes spatial locality in caches to suffer, prompting us to salvage sequential locality by increasing the memory access granularity to that of a cache line. Under this paradigm, each memory access now refers to a 64-byte chunk of memory rather than a single byte. Clearly, this requires support from the software stack, and this work proposes an ISA extension, named as the SELECT instruction, to help.

The SELECT instruction takes as arguments a base address (represented as *rns_concat*), and a separate offset represented in binary to perform a word lookup within a cache line. This hybrid binary-*rns_concat* representation renders the best of binary on one hand, i.e., sequential locality as intended by the programmer, and that of *rns_concat* on the other hand, i.e., no runtime conversion overhead. A detailed example of how such a compiler transformation may be effected is presented below for a word size of 8 bytes, although other word sizes can also be supported.

Representing a portion of the address in binary may seem counter-productive to the reliability of the system, but note that this offset is static (set at compile time) and therefore does not warrant computational error correction, meaning that the ECC protection that comes naturally to instructions (Section 5.7) is sufficient to protect this offset as well.

Introducing such a representation comes with a set of limitations in software. First, if each cache line access is to be accompanied by a SELECT instruction, a static code bloat of about 15% occurs. In practice, one can expect this bloat to decrease significantly due to spatial locality; once a cache line is loaded, multiple SELECTs can be performed without re-issuing the cache line access. Static code bloat can also be reduced by designing compiler optimizations that further leverage spatial/temporal locality in instruction layout and scheduling. Second, an increased memory access granularity renders arbitrary pointer arithmetic and branch targets tricky to disambiguate at compile time, unless they become aligned to cache-line boundaries. Finally, for general purpose system integration, such a hybrid representation may be required to be extended into the software runtime to avoid an explosion of pages. Alternately, TLB and paging performance may also be improved by employing super pages [247, 248], especially in emerging storage class memories [249] (with segmentation support for security).

The SELECT instruction requires tight integration with the software stack, the detailed implementation of which is beyond the scope of this thesis. However, here is an example loop transformation that a compiler may implement. Consider an implementation with a 64 byte cache line size, a *double* of size 8 bytes such that the following *struct* has a size of 16 bytes, and the following snippet of code, where $M < N$ are arbitrary natural numbers that are not necessarily compile-time constants.

```
typedef struct { double x; double y; } Foo;
Foo foos[N];
double sum = 0;
for (i = 0; i < M; ++i) {
  sum += foos[i].x;
  sum += foos[i].y;
}
```

A pseudo-assembly code of the loop body in a *binary* computer would be as follows:

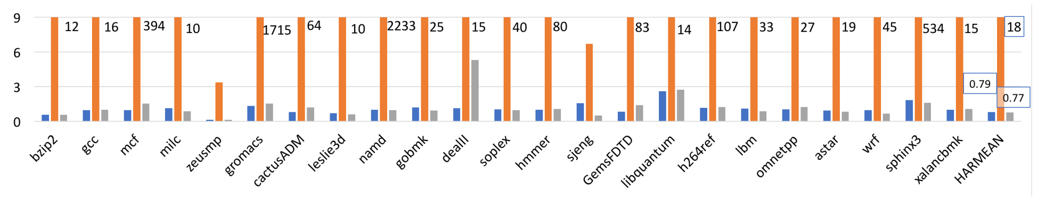
```
LD X, (foos + i*16)     // Read foos[i].x
LD Y, (foos + i*16 + 1*8)  // Read foos[i].y
ADD S, S, X
ADD S, S, Y
```

When SELECT instruction is used, the code transforms to:

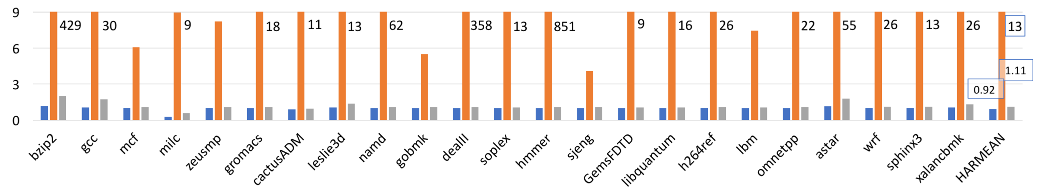
```
uint8_t nOffset = sizeof(CACHELINE)/sizeof(Foo); // 64/16=4
// ++i is in RRNS ; ++n is in binary
for (rns_t i = 0; i < M / nOffset; ++i) {
  LD A64, RNS(foos + i*sizeof(CACHELINE))
  for (int_t n = 0; n < nOffset; ++n) {
    SELECT X, A64, sizeof(Foo)*n
    SELECT Y, A64, sizeof(Foo)*n + 1*8
    ADD S, S, X
    ADD S, S, Y
  }
}
```

5.10.2 Memory Address Interpreter (MAI)

Figure 5.9 presents a simplified view of the memory system. With typical memory systems comprising of L1, L2, L3 caches and a DRAM, the Memory Address Interpreter does not necessarily have to be a singleton unit at the highest level of cache. With a non-inclusive cache hierarchy, the following exhaustive set of legal Memory Address Interpreter



(a) L1 MPKI normalized to binary



(b) L3 MPKI normalized to binary

Figure 5.11: Misses Per Kilo Instructions (MPKI – lower is better) of representative RNS based schemes, normalized to a binary scheme. **SUB** successfully retains spatial locality present in the lower order bits of **IBIN**. **CRNS** salvages sequential locality lost by **NRNS**, with the help of compiler support.

configurations are explored:

1. **Ideal**

IBIN This is the ideal configuration where conversion to binary is without any overhead, or equivalently, a non-error-correcting core that uses a conventional weighted binary representation.

2. **Naive**

NL1 This is a naive conversion approach where every memory access is converted to binary before accessing the L1 cache, thereby incurring a conversion overhead of 8 cycles and its associated energy consumption (adders and lookup tables).

NRNS This is a naive conversion approach where the tuple of residues in the MAR is simply concatenated (*rns_concat*).

NMEM This is a naive conversion approach where *rns_concat* is used through the cache hierarchy and a conversion to binary is effected at the memory controller, thereby incurring an overhead of 24 core cycles (this is because the clock domain of the memory controller is nominally about three times slower than that of the core).

3. **Compiler.** These approaches require compiler support:

CRNS This uses compiler support to realize a memory access granularity of 64 bytes (cache line) and simply concatenates the tuple of address residues before accessing L1. In other words, this is **NRNS** when the SELECT instruction is used.

CX **X**$\in${**L2**, **L3**, **MEM**}. These are similar to **CRNS**, except that a conversion to binary is effected before accessing the L2/L3/main memory respectively.

4. **Rns_sub**

SUB This employs the single cycle overhead conversion of *rns_sub* before the L1 cache is accessed.

SUBM This is similar to **SUB** except that a binary conversion is effected before a main memory access.

5. **CLB**

CLBX **X**$\in${**L1_N**, **MEM_N**}. This is similar to **NL1** or **NMEM**, except that an **N**-entry CLB is used in an attempt to hide the performance overhead of the conversion to binary, although at a potentially higher energy cost. Upon a CLB hit, a single cycle conversion is rendered, however, a CLB miss renders an access time equal to that of a binary conversion as usual. Furthermore, **CCLBMEM_N** is similar to **CMEM** but with an **N**-entry CLB at the memory controller.

CLBY **Y** is of the form **S_N**. This is similar to **SUBM**, except that the conversion to binary at the memory controller is augmented with an **N**-entry CLB.

Space of valid configurations. Those listed above are deemed representative of an exhaustive brute force sweep. For example, it doesn't make sense to perform a conversion to RNS once a binary conversion or an *rns_sub* has already been effected. Also, mixing *rns_sub* at L1 and conversion to binary at L2/L3 imposes constraints on possible cache configurations[1], hence, this work excludes them from our evaluation (although the careful reader may reason about these tradeoffs from the analysis and evaluation presented).

Cache coherence. Cache coherence state is typically maintained at the granularity of a cache line. With the exception of **NRNS**, **NMEM** and **CLBMEM_N**, all the other configurations proposed preserve locality within a cache line at the very least, when compared to **IBIN**. To elaborate, **Compiler** approaches are specifically designed to preserve sequential locality within a cache line. **NL1** and **CLB** based approaches that effectively convert an address to binary prior to L1 trivially preserve locality within a cache line. **Rns_sub** based approaches preserve locality for m_4 consecutive bytes in general (Section 5.9.2); since all the moduli (and m_4, in particular) are greater than 63, therefore, **Rns_sub** preserves cache line locality as well. The conclusion is that all efficient and well performing configurations proposed are also amenable to traditional, unmodified cache coherence protocols.

Summary. Intuitively, the first observation is that the **Compiler** approaches would benefit from the best of locality-preserving properties of *binary* and the no-overhead nature of *rns_concat*, however, at the cost of requiring changes to and tight integration with the software stack. Next, **Rns_sub** approaches would benefit from both its locality-preserving, as well as memory level parallelism inducing properties, at low cost. Finally, **CLB** based approaches would mimic **IBIN**, however, at a significant energy overhead. The simulation results below demonstrate in detail that the relative performance is **Ideal** $>$ **Compiler** $>$ **CLB** $>$ **Rns_sub** $>>$ **Naive**, with **Rns_sub** rendering the most energy efficient architecture along with the added advantage of requiring no support from the software stack.

[1]For example, given the cache configuration in our evaluation, the addresses X (0x4fab84d8) and Y (0x4fab84fc) when subject to *rns_sub* at L1 and conversion to binary at L2, map onto the same L1 index (36) but different L2 indices (165, 166). This makes enforcing a generic non-inclusive property unnecessarily complex.

5.10.3 DRAM Address Interleaving

The MAI of Section 5.10.2 directly impacts DRAM address interleaving. While an industry standard memory controller policy is typically undisclosed, this work span the following interleaving policies gathered from published research:

- **Row**: The LSB bits of the memory address decide the Column index, followed by the Row index, Bank, Rank and the MSB bits decide the Channel (ChRaBaRoCo).
- **Channel**: This address interleaving is devised with the aim of boosting memory level parallelism when a *binary* coded memory address is presented (RoBaRaCoCh).
- **MinOp**: This address interleaving splits the column indices such that exactly 4 consecutive cache lines (assuming a *binary* coded memory address) may map to a single row, thereby striking a balance between row buffer hit rate and memory level parallelism [250].
- **XOR_***: These apply a permutation-based interleaving scheme (for reasons similar to the above) to each of the interleavings presented above [108, 107]. The essence of this is to set the bank index by XORing the bank bits with a selection of higher order bits.

5.11 Results from Memory System Simulation

5.11.1 Evaluation Methodology

We use pin [251] to generate a dynamic instruction trace that records the program counter of each instruction, as well as any load/store address. Our pintool instruments all 32 bit x86 gcc (-O2) optimized binaries from the SPEC 2006 [252] benchmark suite, with test inputs. These traces drive a ramulator [253] based simulator, complete with an L1, L2, L3 non-inclusive cache hierarchy and DRAM main memory. The baseline configuration is presented in Table 5.3.

We enhance the simulator to support the design space presented above, and use McPat [254]/CACTI [255] to model energy overheads due to the Memory Address Interpreter.

Table 5.3: Baseline Configuration

Parameter	Dimensions
Core	Inorder / OoO
OoO Fetch/Retire width/ROB size	4/4/128
L1 size/associativity	32kB/8-way
L2 size/associativity	256kB/8-way
L3 size/associativity	2MB/8-way
Load to use latency L1/L2/L3	4/4+12/4+12+31 cycles
MSHR per cache	16
Caching policy	Non-inclusive/LRU
Core-Memory frequency ratio	3:1
DRAM JEDEC Standard	DDR4 (1channel) / LPDDR4 (2ch)
DRAM address interleaving	Section 5.10.3
DRAM policy	FRFCFS_prioritizeHit Open page
Memory Address Interpreter	Section 5.10.2
CLB size	128/1024 entries
CLB policy	Fully associative/LRU
CLB hit / miss latency	1 cycle/binary conversion

Recall from Section 5.7 that memory addressing logic such as the MAI is assumed to be error-free and can therefore be modeled with conventional MOSFET-based circuits. For the purposes of the power and area model for the MAI, a 32nm/300K/0.9V/2GHz configuration is assumed.

5.11.2 Cache

First, the following text demonstrates the intuition formed in the theoretical analysis of Section 5.9 regarding the impact RNS addressing schemes have on locality. Figure 5.11 shows the cache misses per kilo instructions (MPKI) of RNS schemes, when normalized to binary. *Binary*, captured by **IBIN**, is the normalizing baseline; *rns_concat* is captured by **NRNS**, with the effect of introducing the SELECT instruction captured via **CRNS**, and **SUB** captures behavior of *rns_sub*. The other MAI configurations from Section 5.10.2 do not introduce any new addressing schemes.

As expected from Section 5.9, **NRNS** suffers a significant (18×/13× for L1/L3) loss in spatial locality, whereas **CRNS** helps salvage this. **SUB** was designed to retain the spatial locality that is present in the lower order bits of **IBIN**, and successfully, is more or less on par with it. The interplay of addressing and temporal locality is rendered responsible for cases where RNS schemes show superior cache performance to binary. Results from cache

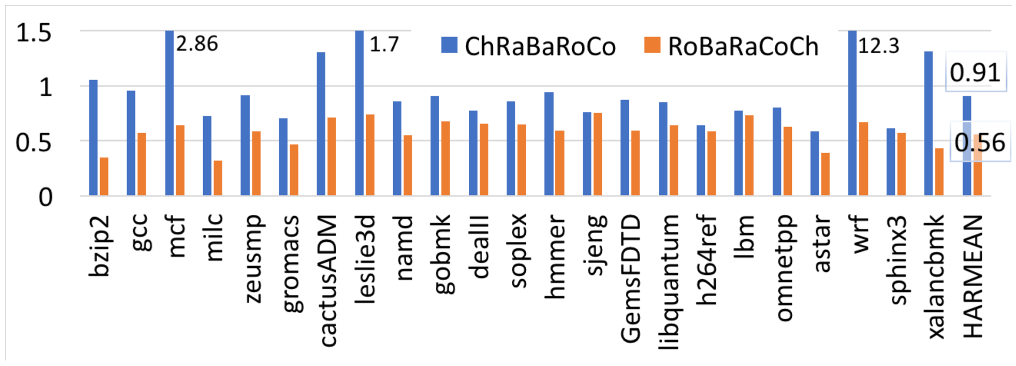


Figure 5.12: DRAM row buffer hit rate (higher is better) of **SUB**, normalized to **IBIN**, under two example address interleavings. Row locality is completely lost in **CRNS** irrespective of interleaving, and is therefore omitted from this plot. Row locality for **SUB** is relatively less impacted, but prefers the first interleaving, by design.

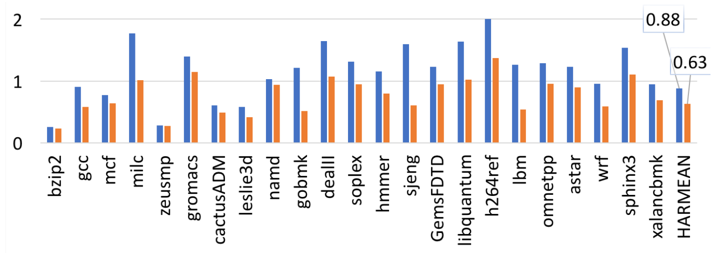
configuration sweeps w.r.t. size/associativity/replacement policy are as expected. Therefore, these are omitted as it adds no new insight to the architecture community.

5.11.3 DRAM

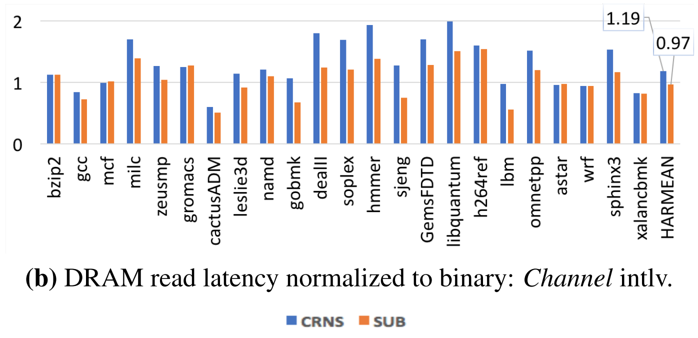
There are two aspects to DRAM performance: row buffer hit rate and memory level parallelism exploited.

By design, one expects **SUB** to show somewhat similar row buffer hit rate to **IBIN** for an interleaving policy that respects the locality preserving design principle of **SUB**, such as *row* interleaving. Figure 5.12 shows its hit rate, normalized to that of binary, for two example interleaving policies. **CRNS**, however, preserves spatial locality at a smaller granularity, thereby exhibiting very low row buffer hit rate. For brevity, this chapter does not present detailed results for **NRNS** because of its incredibly poor cache performance, and also omits depiction of **CRNS** from Figure 5.12 because of its near zero hit rate. The interleaving policy has a significant impact on row buffer locality.

On the other hand, because RNS addressing *naturally* permutes an address, a higher degree of memory level parallelism is expected, especially for linear interleaving policies such as *row* interleaving. Figure 5.13 demonstrates this, as one observes that the average



(a) DRAM read latency normalized to binary: *Row* interleaving.



(b) DRAM read latency normalized to binary: *Channel* intlv.

Figure 5.13: DRAM read latency (lower is better) of representative RNS based schemes, normalized to a binary scheme, under two example address interleavings. RNS based schemes naturally extract more memory level parallelism. Together with row locality characteristics (Figure 5.12), read latency of RNS based schemes are about on-par with or better than binary based schemes.

read latency is significantly improved inspite of an inferior row buffer hit rate for RNS based schemes. This is one of the reasons why several interleaving policies (Section 5.10.3) are deployed for binary based addressing systems, i.e., permuting the address bits reduces bank conflicts and therefore boosts performance (other reasons not related to performance improvement for such permutation are to avoid row-hammer [256] and security issues).

The results presented in this section are with an inorder processor, but similar trends are seen for OoO as well. Also, DRAM scheduling and paging policies have an insignificant impact on performance when compared to the impact due to address interleaving. Therefore this section omits their results for brevity.

5.11.4 Overall Runtime

For completeness, the exhaustive set of 18 MAI configurations from Section 5.10.2 across all 6 DRAM address interleavings from Section 5.10.3 are simulated, and for presentation

Table 5.4: Most favored DRAM interleaving policy for an MAI configuration. In general, RNS based configurations (such as **NRNS**) benefit from increased memory level parallelism especially when linear address interleaving such as *row* interleaving are used and binary based configurations (such as **IBIN**) benefit from permuted and *XOR* interleavings. Favorable interleavings choice is also affected by row buffer locality and memory pressure differences (due to interplay between MAI configuration, cache performance and processor configuration). If several equally performant choices are available, an arbitrary selection is made.

	MAI Configuration	Inorder	OoO
Ideal	IBIN	XOR_MinOp	XOR_MinOp
Naive	NL1	XOR_Channel	
	NRNS	Row	Row
	NMEM		
Rns_sub	SUB	XOR_MinOp	XOR_MinOp
	SUBM	Row	Row
Compiler	CRNS	XOR_MinOp	XOR_MinOp
	CL2	Row	Row
	CL3	XOR_MinOp	XOR_MinOp
	CMEM		
Compiler / CLB	CCLBMEM_128		
	CCLBMEM_1024		
CLB	CLBL1_128		
	CLBL1_1024	XOR_Channel	
	CLBMEM_128	Row	Row
	CLBMEM_1024		
Rns_sub / CLB	CLBS_128	XOR_MinOp	XOR_MinOp
	CLBS_1024		

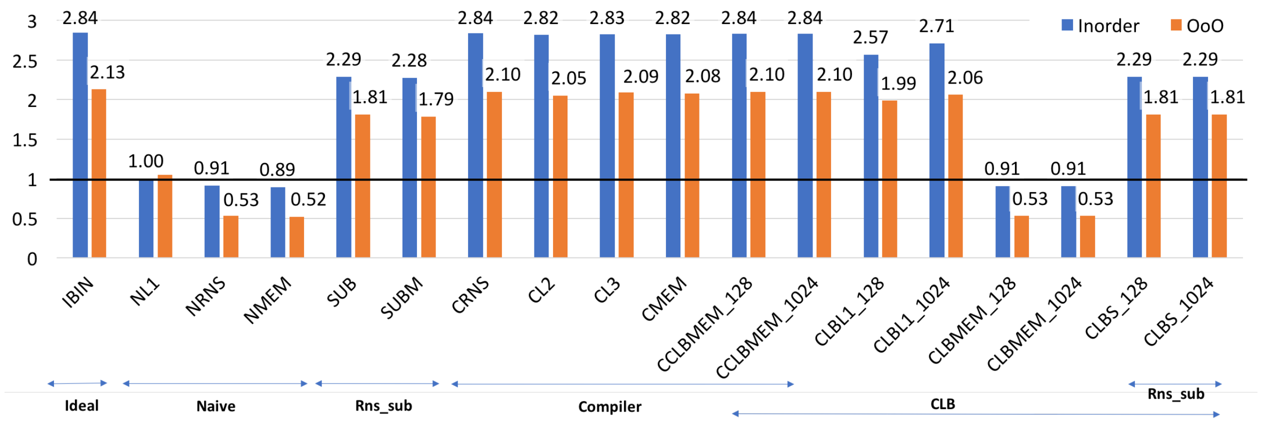
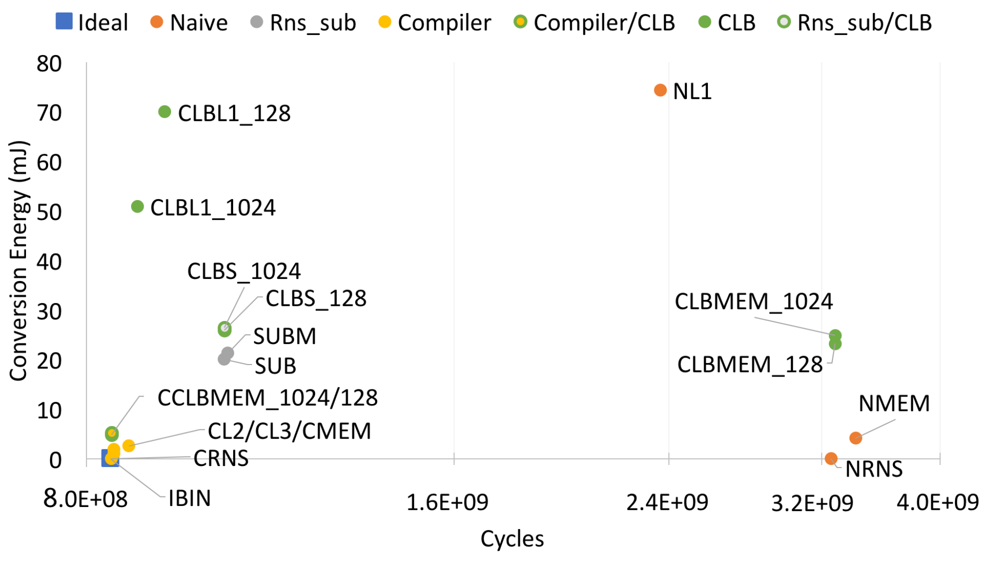
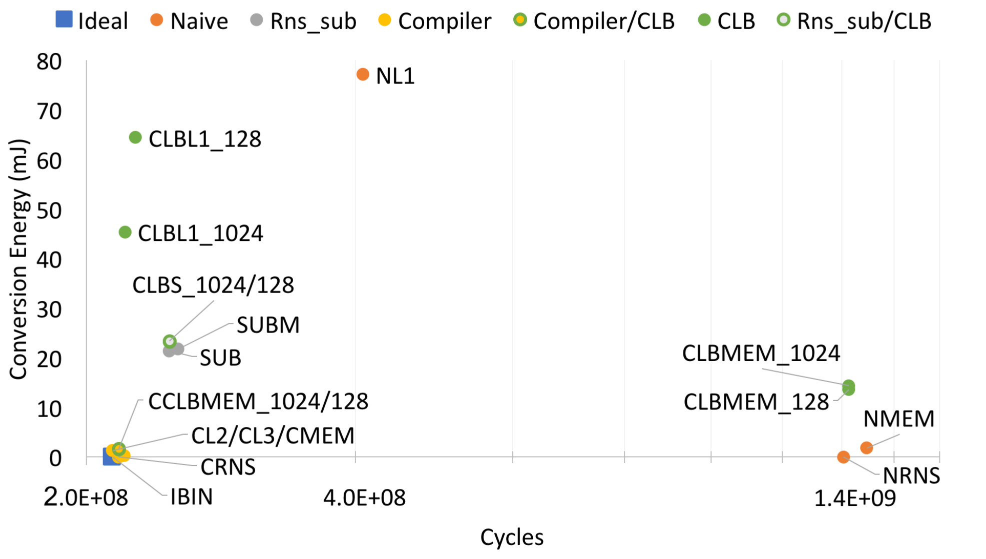


Figure 5.14: Speedup (higher is better) when compared to a naive approach of converting to binary upon each L1 access (**NL1** with *row* interleaving). Inorder cores are understandably more sensitive to conversion latency than OoO cores. For both inorder/OoO cores, the relative performance of each cluster is **Ideal** > **Compiler** > **CLB** > **Rns_sub** >> **Naive**.



(a) Inorder processor



(b) OoO processor

Figure 5.15: Overhead in conversion energy (mJ) vs runtime; lower is better for both axes (similar to a pareto chart). **SUB** is the most efficient microarchitecture configuration that requires no support from the software stack.

purposes summarize the performance for each configuration with its most favored DRAM interleaving policy (Table 5.4) in Figure 5.14. In deriving the most favored interleaving, no distinction is made between workloads, i.e., the interleaving policy is varied only with MAI configuration and is workload independent.

The performances are normalized against a fixed baseline (**NL1** with *row* interleaving) to be able to compare across configurations and interleavings, and the resulting speedups are presented. Only the harmonic mean across the benchmark suite are presented.

[**Ideal**] For example, a conventional non-error-correcting binary core **IBIN** performs $2.84\times/2.13\times$ faster on average when compared to the baseline (for inorder/OoO processor respectively, with a DDR4 DRAM), and favors XOR_MinOp as its preferred address interleaving.

[**Naive**] None of the alternate address interleavings make a noticeable improvement to the performance of the baseline **NL1**. The 8 cycle conversion latency on every access to the memory hierarchy is left unhidden, allowing it to dominate the performance trends. **NRNS/NMEM** incur significant slowdowns because of their poor cache performance.

[**Rns_sub**] **SUB** shows significantly improved performance over the naive approaches. **SUBM** follows closely behind, due to slight decrease in exploited memory level parallelism, coupled with its conversion latency at the memory controller.

[**Compiler**] The approaches that leverage the SELECT instruction effectively approach **IBIN** by combining the best of binary (cache performance) and RNS (no conversion overhead).

[**CLB**] Placing a 128 entry CLB before L1 allows for performance superior to **SUB**, and a 1024 entry CLB seems sufficient to approach the performance of **IBIN**. However, placing a CLB at the memory controller is rendered useless unless either the SELECT instruction is used or *rns_sub* is used to prevent an explosion of cache misses.

From an Amdahl's law perspective, the amount of overhead and variation in memory access patterns introduced by various MAI configurations has more weight than just the

DRAM interleaving policy, which is to be expected. If one were to choose an unfavorable DRAM interleaving instead, the performance on average for each of the non-naive configurations vary by less than 1%, although detailed results are omitted for brevity.

LPDDR4 is a likely candidate to be used in conjunction with low power microarchitectures, but comes at a cost of increased row activation latency. Nevertheless, the insights presented in this work hold even when an LPDDR4 DRAM is used.

5.11.5 Energy Overhead

Figure 5.15 presents the performance of each of the MAI configurations, when put in perspective of how much *additional* energy they consume. For each MAI configuration, their most favored DDR4-DRAM interleaving is chosen (Table 5.4) and the mean cycle count and conversion energy overhead (mJ) across the benchmark suite are presented.

[**Ideal**] For example, at the bottom left is **IBIN**, taking the least number of cycles and without any conversion overhead.

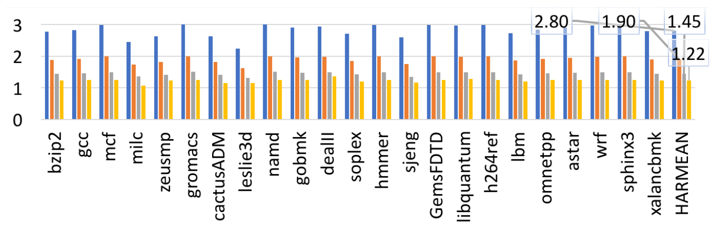
[**Naive**] **NL1**, as has been mentioned throughout this chapter, suffers from poor performance and high energy consumption. **NMEM/NRNS** have little or no increase in energy consumption due to conversion alone, but their performance suffers greatly.

[**Rns_sub**] **SUB/SUBM** are the most *efficient* microarchitectures that do not require support from the software stack for them to work.

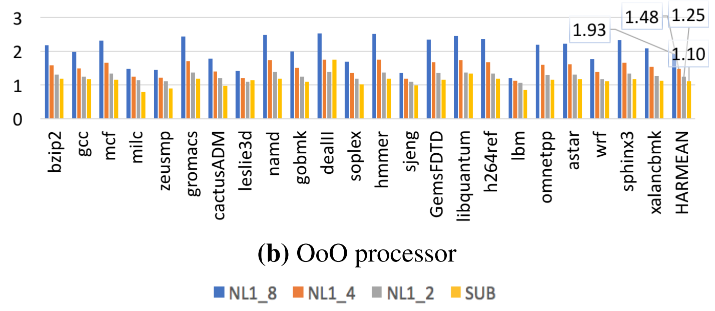
[**Compiler**] While these approach **IBIN**, they are subject to software compatibility and integration issues as discussed in Section 5.10.1.

[**CLB**] While placing a CLB at the L1 is more performant than **SUB**, it comes at a higher conversion energy overhead. Placing a CLB at the memory controller must be accompanied by either an *rns_sub* addressed cache or a cache line addressed (SELECT instruction) cache.

Deriving a CLB configuration is similar to that of a TLB- or cache-like structure. For example, while increasing the CLB size increases access power, it may reduce the number of CLB misses, which also translates into energy savings on CLB miss repairs (ex: **CLBL1_128**



(a) Inorder processor



(b) OoO processor

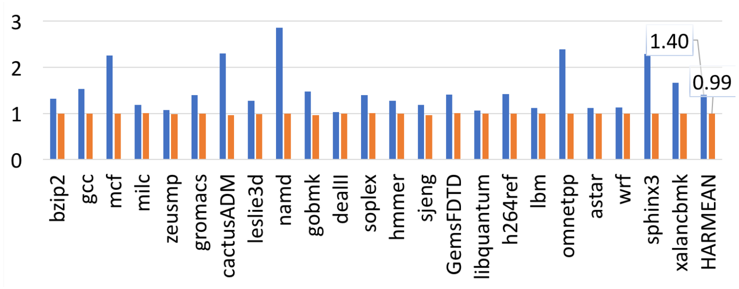
Figure 5.16: Slowdown of hypothetically faster binary convertors (4 cycles and 2 cycles as opposed to the best known candidate: 8 cycles) when compared to **IBIN**. **NL1** thats used everywhere in this document is annotated as **NL1_8** for clarity. Our most efficient microarchitecture technique that uses **SUB** is still faster than these hypothetically faster convertors. It is also more energy efficient than these, however, a quantitative comparison is not possible in the absence of concrete algorithms for faster binary conversion.

vs **CLBL1_1024**). For brevity, this work limit a CLB configuration sweep to just these two for demonstrative purposes; the reader should be able to easily infer points of tradeoff for other CLB configurations.

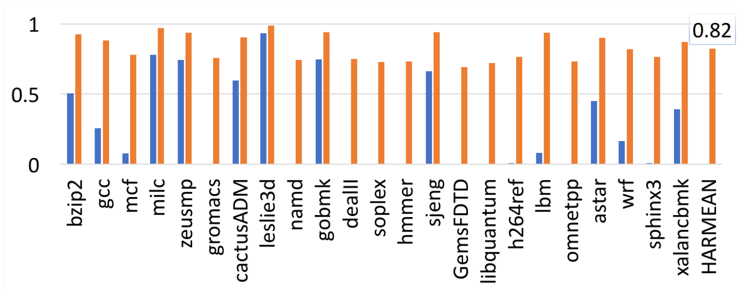
MAI Area. Table 5.5 presents the area requirements of few key configurations. It is concluded that the energy trends discussed above apply to area as well.

Table 5.5: Area requirement in mm^2.

NRNS	SUB	CLB_128	CLB_1024	NL1
0	0.075	0.091	0.104	0.160



(a) TLB hit rate normalized to binary (higher is better)



(b) Fraction of pages that do not cause a hard disk access, normalized to binary (higher is better)

■ CRNS ■ SUB

Figure 5.17: Virtual memory performance of **CRNS** and **SUB** when compared to **IBIN**. **CRNS** has superior TLB hit rate owing to its higher memory access granularity, however, an explosion is seen in the number of physical pages it touches, owing to its limited spatial locality granularity. As expected, both TLB and page pressure of **SUB** approach that of **IBIN**.

5.11.6 Sensitivity to Conversion Latency

Throughout this chapter, an 8 cycle algorithm has been assumed for converting an RNS tuple to a binary number (**NL1**), as it is the state of the art given that the bases must be amenable to RRNS error correction. Nevertheless, this work also evaluates the relative performance of hypothetical conversion algorithms that take half as many or even a quarter of the cycles. For clarity, these are differentiated via subscripts (**NL1_8**, **NL1_4**, **NL1_2**). Figure 5.16 puts their performance in perspective of two representative MAI configurations: **IBIN** and **SUB**. Recall that **SUB** presents the most efficient microarchitecture that does not impose restrictions on software. Therefore, for this analysis, **IBIN** is chosen as the baseline. **SUB** and **NL1_8** are also presented to put the performance of these hypothetical convertors into perspective. Relative performance of the remaining MAI configurations can be easily inferred from the previous result sections.

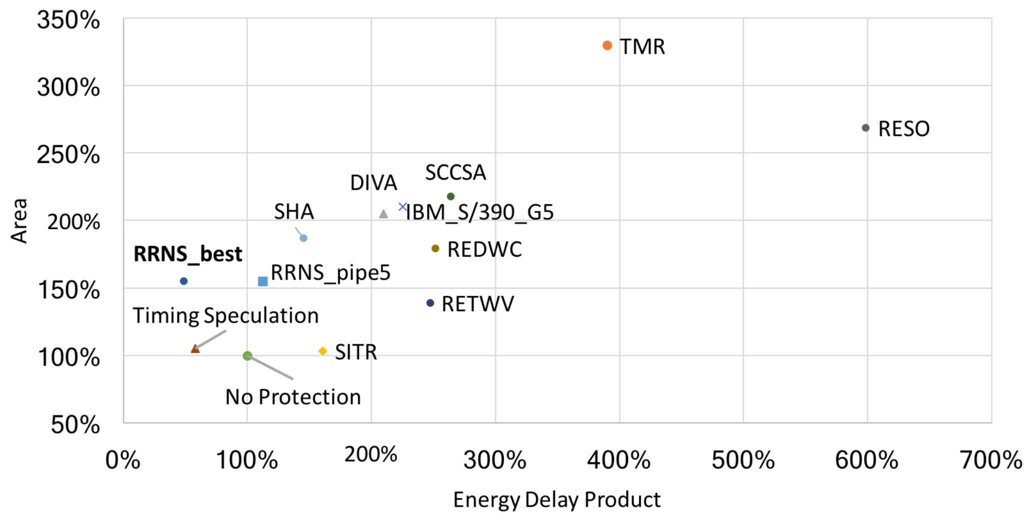
Not only are these hypothetical faster convertors less performant than the schemes presented in this work, but would also be less energy efficient. While a quantitative comparison is not possible in the absence of concrete faster algorithms, one can expect their energy overhead to be rather close to **NL1_8** (and certainly much more than **SUB**) from a functional standpoint of the process of converting to binary.

5.11.7 Virtual Memory

Certain RRNS based architectures may find it necessary to integrate with a virtual memory (VM) subsystem. There are two aspects to VM performance: TLB hit rate and page pressure. Under an idempotent virtual-to-physical transformation, this work uses the reuse distance [257, 258] mechanism to estimate the TLB hit rate in a 512kB structure and to estimate page pressure using an 8GB DRAM, assuming a page size of 4kB (as is common on Linux systems). Page pressure is quantified by measuring the number of pages that have already been brought into main memory as well as the number of pages that need to access secondary memory (such as a non-volatile disk), as they may either have not yet been allocated a physical page frame or had been evicted to make way for newer pages.

Figure 5.17 highlights these for representative RNS schemes, when compared to binary. Given its 64 byte access granularity, **CRNS** exhibits high TLB performance thanks to sequential locality in programs. However, this granularity is rather small at the page level, causing an explosion in the number of pages it accesses. As described in Section 5.10.1, **CRNS** would have to be extended into the runtime system (or a different page granularity should be used) in order to attain low page pressure.

SUB, on the other hand, performs very similar to **IBIN** on both counts. The reason for its relatively higher page pressure when compared to binary is that 4kB pages have an offset of 12 bits, whereas the residues in this work are (or specifically the fourth residue) 8 bits wide, thereby leading to a gap in locality between *rns_sub* and *binary*. Choosing a wider m_4



RESO [259], REDWC [164], RETWV [165], SCCSA [260], SHA [171], DIVA [187], SITR [261], Timing Speculation [185, 186]

Figure 5.18: First order comparison of area overhead and energy-delay product (EDP) of various mechanisms for computational error correction, depicting the superiority of RRNS. Computational error correction techniques use a combination of spatial and temporal redundancy techniques. While temporal redundancy allows for a low area overhead, they suffer from a significant performance penalty. Timing speculation techniques seem more efficient than RRNS, however, their error model assumes all bit errors manifest as circuit timing errors, which is not sufficient to work with ultra low energy logic devices.

would result in an interesting tradeoff as it may change the power-performance-reliability characteristics of the RRNS core, but from a memory systems point of view, it would enhance the spatial locality properties of *rns_sub* which directly translates to improved cache performance, DRAM row buffer hit rate and virtual memory performance.

5.12 Summary of Related Work for Chapter 5

Computational error correction. Figure 5.18 and Section 5.2 summarize various techniques in comparison with RRNS. RRNS is generally considered superior in terms of capability and efficiency for computational error resilience. State of the art adoption and research in the industry also claim the superiority of residue based resilience [188, 190].

Approaches that employ timing speculation [185, 186] may seem superior to RRNS at first glance. However, the error model that can be supported by an RRNS error correcting architecture is orthogonal to theirs, if not broader. For example, razor [185] uses conventional transistors, therefore lowering V_{dd} lowers MOSFET switching speed significantly, resulting in a large frequency drop, which could cause setup time violations that they handle via a delayed latch mechanism. They assume that any error manifests itself as a timing error. Similarly, decor [186] uses a delayed commit approach (with rollback support) to handle violations in timing margins. However, with emerging devices (Section 5.1), V_{dd} can be lowered to few tens of millivolts with a relatively lower frequency loss, meaning that operating at the resultant thermal noise floor leads to *stochastic, intermittent* bit flips, which cannot always be captured as circuit timing errors. Unlike such approaches, RRNS error correcting architectures can not only tolerate such errors in the data path, but also in the control path between memory accesses.

In terms of being able to tolerate control path errors, approaches such as DIVA [187] that replicate parts of the pipeline are capable. Their design provides recovery by having a simple core recalculate results of an out-of-order core. In this approach, the simple core is assumed to be error-free. This is similar to a ”double-modular-redundancy” approach with a rad-hard node, implying a relatively high overhead. Furthermore, if the rad-hard simple core is instead prone to error, checkpoint and re-execute methods would need to be employed, similar to the IBM POWER6/7/8 and z10/z196 processors [188, 189, 190, 191] and various Intel Corporation [192] and Sun Microsystems based mainframes [193]. On the other hand, RRNS is able to tolerate errors in its redundant as well as non-redundant computations.

Finally, a vast majority of these related work are at the circuit level and can be augmented into RRNS-based architectures for potentially enhanced reliability characteristics.

Prime number based indexing. Kharbutli et al. [137] propose using prime numbers for cache indexing to reduce conflict misses. However, such indexing introduces fragmentation that can only be amortized by higher cache capacities. They therefore recommend using their technique only for larger caches (such as L2/L3). Furthermore, their technique isn’t applicable to DRAM addressing.

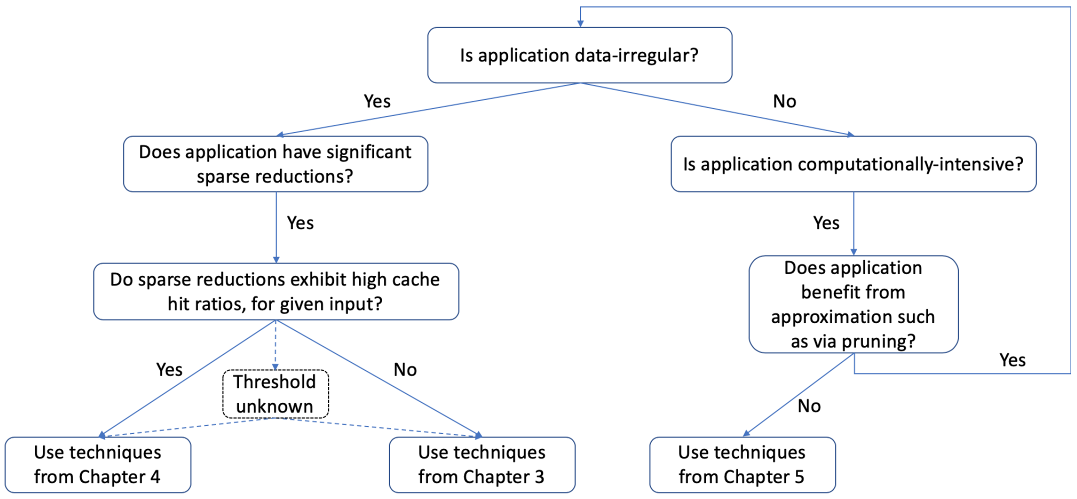
5.13 Summary of Contributions of Chapter 5

This section summarizes the contributions of the third and final part of this thesis, as explained in Chapter 5.

1. Develops microarchitecture, ISA and RRNS centered algorithms towards a Computationally-Redundant, Energy-Efficient core design.
2. Designs RRNS-check-insertion heuristics to optimize performance/energy/reliability tradeoffs.
3. Derives an estimated lower limit on signal energies via stochastic fault injecting simulation.
4. Extends an RRNS microarchitecture - one that is capable of reliably functioning with ultra-low energy logic devices that operate near the kT thermal noise floor at tens of millivolts - to support efficient memory hierarchy access.
5. Proposes and analyzes an efficient, novel translation scheme (*rns_sub*) with locality properties similar to that of *binary*, but with a fraction of its performance/energy overhead.
6. Reduces the performance impact of the naive *binary* approach by using a TLB-like structure called the *Conversion Lookaside Buffer* (CLB) to cache conversions.
7. Proposes a technique to improve spatial locality in the native, zero-overhead translation scheme (*rns_concat*) via a hybrid compiler approach and a modified programming model.
8. Constructs a design space from the schemes proposed and from the analysis of the resultant memory access pattern behavior, along with a detailed cost-benefit analysis.

CHAPTER 6
CONCLUSION

The landscape of modern computer architecture has changed significantly over the past few years. This is driven by a combination of two forces. On one hand, emerging applications are pulling compute and memory system demand in unprecedented directions. On the other hand, technology scaling has not benefitted recent computing systems in the same manner that it did a couple decades ago.



Is application data-irregular?
Yes
No
Does application have significant sparse reductions?
Is application computationally-intensive?
Yes
Yes
Do sparse reductions exhibit high cache hit ratios, for given input?
Does application benefit from approximation such as via pruning?
Yes
Threshold unknown
No
Yes
No
Use techniques from Chapter 4
Use techniques from Chapter 3
Use techniques from Chapter 5

Figure 6.1: Scope and overview of thesis.

At the resulting intersection of an increasingly important set of data-irregular applications, and bottlenecks induced by Moore's law-era physics, lies the reason this thesis exists. This work finds that mitigating these challenges requires non-traditional solutions that tackle various forms of memory access irregularities. To this end, this thesis proposes architectures that improve energy efficiency and performance by intelligently reducing data movement through the memory hierarchy for applications that exhibit data-irregularities due to sparse accesses or due to computationally error-tolerant post-Moore processing. Energy efficiency and performance of data irregular applications can be improved via near memory and near processor sparse data stream acceleration and address remapping.

A visual overview of the organization of the scope of applicability of the techniques presented in this thesis is summarized in Figure 6.1.

Fundamentally, I hope this thesis serves as proof and as an inspiration to designers of future efficient computing systems to innovate simultaneously across multiple levels of abstractions - from traditional and emerging device technologies all the way to applications from a spread of domains - without losing the benefits and rigor of specialization in a chosen field of interest.

REFERENCES

[1] K. K. Chang, "Understanding and improving the latency of dram-based memory systems," *ArXiv preprint arXiv:1712.08304*, 2017.

[2] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, IEEE, 2015, pp. 158–169.

[3] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, IEEE, 2013, pp. 56–65.

[4] P. Kogge, "Memory intensive computing, the 3rd wall, and the need for innovation in architecture," *MEMSYS*, 2017.

[5] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[6] A. McMenamin, "The end of dennard scaling," 2013.

[7] H.-H. Lee, Y. Wu, and G. Tyson, "Quantifying instruction-level parallelism limits on an epic architecture," in *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on*, IEEE, 2000, pp. 21–27.

[8] J. González and A. González, "Limits of instruction level parallelism with data speculation," in *Proc. of the VECPAR Conf*, Citeseer, 1998, pp. 585–598.

[9] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 19, 1991, pp. 276–286.

[10] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "Cpu db: Recording microprocessor history," *Communications of the ACM*, vol. 55, no. 4, pp. 55–63, 2012.

[11] A. I. Khan, *Negative capacitance for ultra-low power computing*. University of California, Berkeley, 2015.

[12] R. Keyes, “Miniaturization of electronics and its limits,” *IBM Journal of Research and Development*, vol. 32, no. 1, pp. 84–88, 1988.

[13] R. M. Swanson and J. D. Meindl, “Ion-implanted complementary mos transistors in low-voltage circuits,” *IEEE Journal of Solid-State Circuits*, vol. 7, no. 2, pp. 146–153, 1972.

[14] J. Von Neumann, A. W. Burks, *et al.*, “Theory of self-reproducing automata,” *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 3–14, 1966.

[15] B. H. Calhoun, A. Wang, and A. Chandrakasan, “Modeling and sizing for minimum energy operation in subthreshold circuits,” *IEEE Journal of Solid-State Circuits*, vol. 40, no. 9, pp. 1778–1786, 2005.

[16] T. N. Theis and P. M. Solomon, “In quest of the “next switch”: Prospects for greatly reduced power dissipation in a successor to the silicon field-effect transistor,” *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2005–2014, 2010.

[17] D. E. Nikonov and I. A. Young, “Overview of beyond-cmos devices and a uniform methodology for their benchmarking,” *Proceedings of the IEEE*, vol. 101, no. 12, pp. 2498–2533, 2013.

[18] T. N. Theis, “(keynote) in quest of a fast, low-voltage digital switch,” *ECS Transactions, 45(6), 3-11*, 2012.

[19] A. I. Khan, K. Chatterjee, J. P. Duarte, Z. Lu, A. Sachid, S. Khandelwal, R. Ramesh, C. Hu, and S. Salahuddin, “Negative capacitance in short-channel finfets externally connected to an epitaxial ferroelectric capacitor,” *IEEE Electron Device Letters*, vol. 37, no. 1, pp. 111–114, 2016.

[20] A. I. Khan and S. Salahuddin, “4 extending cmos with negative capacitance,” *CMOS and Beyond: Logic Switches for Terascale Integrated Circuits*, pp. 56–76, 2015.

[21] A. I. Khan, C. W. Yeung, C. Hu, and S. Salahuddin, “Ferroelectric negative capacitance mosfet: Capacitance tuning &amp; antiferroelectric operation,” in *Electron Devices Meeting (IEDM), 2011 IEEE International*, IEEE, 2011, pp. 11–3.

[22] S. Salahuddin and S. Datta, “Can the subthreshold swing in a classical fet be lowered below 60 mv/decade?” In *Electron Devices Meeting, 2008. IEDM 2008. IEEE International*, IEEE, 2008, pp. 1–4.

[23] S. K. Samal, S. Khandelwal, A. I. Khan, S. Salahuddin, C. Hu, and S. K. Lim, “Full chip power benefits with negative capacitance fets,” in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, IEEE, 2017, pp. 1–6.

[24] E. P. DeBenedictis, J. Cook, S. Srikanth, and T. M. Conte, “Superstrider associative array architecture: Approved for unlimited unclassified release: Sand2017-7089 c,” in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, IEEE, 2017, pp. 1–7.

[25] A. Jain, S. Srikanth, E. DeBenedictis, and T. Krishna, “Merge network for a non-von neumann accumulate accelerator in a 3d chip,” in *IEEE International Conference on Rebooting Computing (ICRC)*, 2018.

[26] S. Srikanth, T. M. Conte, E. P. DeBenedictis, and J. Cook, “The superstrider architecture: Integrating logic and memory towards non-von neumann computing,” in *Rebooting Computing (ICRC), 2017 IEEE International Conference on*, IEEE, 2017, pp. 1–8.

[27] S. Srikanth, A. Jain, J. M. Lennon, T. M. Conte, E. Debenedictis, and J. Cook, “Metastrider: Architectures for scalable memory-centric reduction of sparse data streams,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, p. 35, 2019.

[28] S. Agarwal, J. Cook, E. DeBenedictis, M. P. Frank, G. Cauwenberghs, S. Srikanth, B. Deng, E. R. Hein, P. G. Rabbat, and T. M. Conte, “Energy efficiency limits of logic and memory,” in *Rebooting Computing (ICRC), IEEE International Conference on*, IEEE, 2016, pp. 1–8.

[29] B. Deng, S. Srikanth, E. R. Hein, P. G. Rabbat, T. M. Conte, E. DeBenedictis, and J. Cook, “Computationally-redundant energy-efficient processing for y’all (creepy),” in *Rebooting Computing (ICRC), IEEE International Conference on*, IEEE, 2016, pp. 1–8.

[30] B. Deng, S. Srikanth, E. R. Hein, T. M. Conte, E. Debenedictis, J. Cook, and M. P. Frank, “Extending moore’s law via computationally error-tolerant computing,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 1, p. 8, 2018.

[31] S. Srikanth, P. G. Rabbat, E. R. Hein, B. Deng, T. M. Conte, E. DeBenedictis, J. Cook, and M. P. Frank, “Memory system design for ultra low power, computationally error resilient processor microarchitectures,” in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, IEEE, 2018, pp. 696–709.

[32] E. Nurvitadhi, A. Mishra, and D. Marr, “A sparse matrix vector multiply accelerator for support vector machine,” in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2015 International Conference on*, IEEE, 2015, pp. 109–116.

[33] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr, “Fine-grained accelerators for sparse machine learning workloads,” in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, IEEE, 2017, pp. 635–640.

[34] I. S. Duff, M. A. Heroux, and R. Pozo, “An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, no. 2, pp. 239–267, 2002.

[35] I. Yamazaki and X. S. Li, “On techniques to improve robustness and scalability of a parallel hybrid linear solver,” in *International Conference on High Performance Computing for Computational Science*, Springer, 2010, pp. 421–434.

[36] N. Bell, S. Dalton, and L. N. Olson, “Exposing fine-grained parallelism in algebraic multigrid methods,” *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[37] V. Hapla, D. Horák, and M. Merta, “Use of direct solvers in tfeti massively parallel implementation,” in *International Workshop on Applied Parallel Computing*, Springer, 2012, pp. 192–205.

[38] S. Itoh, P. Ordejón, and R. M. Martin, “Order-n tight-binding molecular dynamics on parallel computers,” *Computer physics communications*, vol. 88, no. 2-3, pp. 173–185, 1995.

[39] W. Kohn, “Density functional and density matrix method scaling linearly with the number of atoms,” *Physical Review Letters*, vol. 76, no. 17, p. 3168, 1996.

[40] J. R. Gilbert, S. Reinhardt, and V. B. Shah, “High-performance graph algorithms from parallel sparse matrices,” in *International Workshop on Applied Parallel Computing*, Springer, 2006, pp. 260–269.

[41] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[42] T. M. Chan, “More algorithms for all-pairs shortest paths in weighted graphs,” *SIAM Journal on Computing*, vol. 39, no. 5, pp. 2075–2089, 2010.

[43] M. O. Rabin and V. V. Vazirani, “Maximum matchings in general graphs through randomization,” *Journal of Algorithms*, vol. 10, no. 4, pp. 557–567, 1989.

[44] A. Buluç and J. R. Gilbert, “The combinatorial blas: Design, implementation, and applications,” *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[45] L. Roditty and U. Zwick, "Improved dynamic reachability algorithms for directed graphs," *SIAM Journal on Computing*, vol. 37, no. 5, pp. 1455–1471, 2008.

[46] V. B. Shah, *An interactive system for combinatorial scientific computing with an emphasis on programmer productivity*. University of California, Santa Barbara, 2007.

[47] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, IEEE, 2015, pp. 804–811.

[48] R. Yuster and U. Zwick, "Detecting short directed cycles using rectangular matrix multiplication and dynamic programming," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, 2004, pp. 254–260.

[49] K. Anderson and S. Plimpton, "Firehose streaming benchmarks," Sandia National Laboratory, Tech. Rep., 2015.

[50] P. M. Kogge and S. K. Kuntz, "A case for migrating execution for irregular applications," in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, ACM, 2017, p. 6.

[51] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *ArXiv preprint arXiv:1510.00149*, 2015.

[52] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[53] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 2074–2082.

[54] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the granularity of sparsity in convolutional neural networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2017.

[55] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 32, 2017.

[56] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," *ArXiv preprint arXiv:1802.06367*, 2018.

[57] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[58] J. Park, S. Li, W. Wen, P. T. P. Tang, H. Li, Y. Chen, and P. Dubey, "Faster cnns with direct sparse convolutions and guided pruning," *ArXiv preprint arXiv:1608.01409*, 2016.

[59] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[60] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 93–104.

[61] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[62] S.-W. Jun, A. Wright, S. Zhang, S. Xu, *et al.*, "Grafboost: Using accelerated flash storage for external graph analytics," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018.

[63] A. Mukkara, N. Beckmann, and D. Sanchez, "Phi: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1009–1022.

[64] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "High-performance sparse matrix-matrix products on intel knl and multicore architectures," *ArXiv preprint arXiv:1804.01698*, 2018.

[65] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, IEEE, 2018, pp. 724–736.

[66] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*. Oxford University Press, 2017.

[67] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, IEEE, 2008, pp. 1–11.

[68] R. W. Vuduc and J. W. Demmel, *Automatic performance tuning of sparse matrix kernels*. University of California, Berkeley, 2003, vol. 1.

[69] U. Borštnik, J. VandeVondele, V. Weber, and J. Hutter, "Sparse matrix multiplication: The distributed block-compressed sparse row library," *Parallel Computing*, vol. 40, no. 5-6, pp. 47–58, 2014.

[70] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in matlab: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[71] *Gcc std::map*, `https://gcc.gnu.org/onlinedocs/libstdc/libstdc-html-USERS-3.4/stl__map_8h-source.html`.

[72] M. Deveci, C. Trott, and S. Rajamanickam, "Performance-portable sparse matrix-matrix multiplication for many-core architectures," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, IEEE, 2017, pp. 693–702.

[73] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," USENIX, 2012.

[74] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ACM, 2013, pp. 472–488.

[75] A. Barredo, J. C. Beard, and M. Moretó, "Poster: Spidre: Accelerating sparse memory access patterns," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019, pp. 483–484.

[76] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 481–492.

[77] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 383–396.

[78] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, Phoenix, Arizona: ACM, 2019, pp. 397–410, ISBN: 978-1-4503-6669-4.

[79] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *IEEE micro*, vol. 17, no. 2, pp. 34–44, 1997.

[80] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, 2016.

[81] M. Kang, E. P. Kim, M.-s. Keel, and N. R. Shanbhag, "Energy-efficient and high throughput sparse distributed memory architecture," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2015, pp. 2505–2508.

[82] I. S. Duff and J. K. Reid, "Some design features of a sparse matrix code," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 1, pp. 18–35, 1979.

[83] F. G. Gustavson, "Some basic techniques for solving sparse systems of linear equations," in *Sparse matrices and their applications*, Springer, 1972, pp. 41–52.

[84] *Https://en.cppreference.com/w/cpp/container/unordered_map*, 2018.

[85] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *International Conference on High Performance Computing*, Springer, 2015, pp. 48–57.

[86] K. Akbudak and C. Aykanat, "Exploiting locality in sparse matrix-matrix multiplication on many-core architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2258–2271, 2017.

[87] M. Intel, "Intel math kernel library," 2007.

[88] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance evaluation of sparse matrix multiplication kernels on intel xeon phi," in *International Conference on Parallel Processing and Applied Mathematics*, Springer, 2013, pp. 559–570.

[89] K. Rupp, F. Rudolf, and J. Weinbub, "Viennacl-a high level linear algebra library for gpus and multi-core cpus," in *Intl. Workshop on GPUs and Scientific Applications*, 2010, pp. 51–56.

[90] P. D. Sulatycke and K. Ghose, "Caching-efficient multithreaded fast multiplication of sparse matrices," in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, IEEE, 1998, pp. 117–123.

[91] S. Dalton, L. Olson, and N. Bell, “Optimizing sparse matrix—matrix multiplication for the gpu,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 4, p. 25, 2015.

[92] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, and U. Naumann, “Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging,” *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. C54–C71, 2015.

[93] C. NVIDIA, “Cusparse library,” *NVIDIA Corporation, Santa Clara, California*, 2014.

[94] W. Liu and B. Vinter, “An efficient gpu general sparse matrix-matrix multiplication for irregular data,” in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, IEEE, 2014, pp. 370–381.

[95] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli, “Sparse matrix-matrix multiplication on modern architectures,” in *High Performance Computing (HiPC), 2012 19th International Conference on*, IEEE, 2012, pp. 1–10.

[96] F. G. Gustavson, “Two fast algorithms for sparse matrices: Multiplication and permuted transposition,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[97] C. Y. Lin, N. Wong, and H. K.-H. So, “Design space exploration for sparse matrix-matrix multiplication on fpgas,” *International Journal of Circuit Theory and Applications*, vol. 41, no. 2, pp. 205–219, 2013.

[98] F. Sadi, L. Fileggi, and F. Franchetti, “Algorithm and hardware co-optimized solution for large spmv problems,” in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, IEEE, 2017, pp. 1–7.

[99] *High bandwidth memory (hbm) dram*, `https://www.jedec.org/standards-documents/results/HBM`.

[100] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning, “In-memory intelligence,” *IEEE Micro*, vol. 37, no. 4, pp. 30–38, 2017.

[101] G. Georgy Adelson-Velsky and E. Landis, “An algorithm for the organization of information,” in *Proceedings of the USSR Academy of Sciences*, 1962.

[102] O. Mutlu and L. Subramanian, “Research problems and opportunities in memory systems,” *Supercomputing frontiers and innovations*, vol. 1, no. 3, pp. 19–55, 2015.

[103] S. Srikanth, L. Subramanian, S. Subramoney, T. M. Conte, and H. Wang, "Tackling memory access latency through dram row management," in *Proceedings of the International Symposium on Memory Systems*, ACM, 2018, pp. 137–147.

[104] *Https://math.nist.gov/matrixmarket/.*

[105] V. N. Rao and V. Kumar, "Parallel depth first search. part i. implementation," *International Journal of Parallel Programming*, vol. 16, no. 6, pp. 479–499, 1987.

[106] H. L. Garner, "The residue number system," *IRE Transactions on Electronic Computers*, no. 2, pp. 140–147, 1959.

[107] B. R. Rau, "Pseudo-randomly interleaved memory," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 19, 1991, pp. 74–83.

[108] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, ACM, 2000, pp. 32–41.

[109] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[110] S. Roy, M. Choudhury, R. Puri, and D. Z. Pan, "Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 10, pp. 1517–1530, 2014.

[111] V Manikandan, V. Muralikrishna, J Ajayan, and V. M. R. Deen, "Static carry skip adder designed using 22-nm strained silicon cmos technology operating under wide range of temperatures," *International Journal of Engineering and Technical Research*, vol. 8, no. 2,

[112] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, 14:1–14:25, Jun. 2017.

[113] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, "Architecting an energy-efficient dram system for gpus," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 73–84.

[114] H. Kwon and T. Krishna, “Opensmart: Single-cycle multi-hop noc generator in bsv and chisel,” in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 195–204.

[115] *Https://www.gamersnexus.net/news-pc/2972-amd-vega-frontier-edition-tear-down-die-size-and-more*.

[116] *Https://www.amd.com/documents/high-bandwidth-memory-hbm.pdf*.

[117] *Https://www.anandtech.com/show/9969/jedec-publishes-hbm2-specification*.

[118] Intel, “Intel 64 and IA-32 architectures optimization reference manual,” Intel, Tech. Rep., 2019.

[119] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D’Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, “Haswell: The fourth-generation intel core processor,” *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[120] NVIDIA, 2019.

[121] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin, “Fractal prefetching b+-trees: Optimizing both cache and disk performance,” in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, ACM, 2002, pp. 157–168.

[122] Y. Li, B. He, Q. Luo, and K. Yi, “Tree indexing on flash disks,” in *2009 IEEE 25th International Conference on Data Engineering*, IEEE, 2009, pp. 1303–1306.

[123] (). Https://github.com/intellabs/skimcaffe, (visited on 2018).

[124] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe,” *Proceedings of the ACM International Conference on Multimedia - MM ’14*, 2014.

[125] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[126] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[127] S. D. Compiler. (). Https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test.html, (visited on 2019).

[128] (). Https://www.eda.ncsu.edu/wiki/freepdk15:contents, (visited on 2017).

[129] PACE, *Partnership for an Advanced Computing Environment (PACE)*, 2017.

[130] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in *ACM SIGPLAN Notices*, ACM, vol. 50, 2015, pp. 521–532.

[131] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," in *ACM SIGPLAN Notices*, ACM, vol. 34, 1999, pp. 229–241.

[132] M. Kandemir, J Ramanujam, and A. Choudhary, "Improving cache locality by a combination of loop and data transformations," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 159–167, 1999.

[133] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No. 02TH8627)*, IEEE, 2002, pp. 73–78.

[134] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have your scratchpad and cache it too," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 43, 2015, pp. 707–719.

[135] M. Moazeni, A. Bui, and M. Sarrafzadeh, "A memory optimization technique for software-managed scratchpad memory in gpus," in *2009 IEEE 7th Symposium on Application Specific Processors*, IEEE, 2009, pp. 43–49.

[136] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "Wcet centric data allocation to scratchpad memory," in *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, IEEE, 2005, 10–pp.

[137] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," in *Software, IEE Proceedings-*, IEEE, 2004, pp. 288–299.

[138] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware," in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, 2013, pp. 1–6.

[139] G. Zhang and D. Sanchez, "Leveraging caches to accelerate hash tables and memoization," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52, Columbus, OH, USA: ACM, 2019, pp. 440–452, ISBN: 978-1-4503-6938-1.

[140] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2013, pp. 468–479.

[141] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in dna sequences using a bloom filter," *BMC bioinformatics*, vol. 12, no. 1, p. 333, 2011.

[142] P. A. La Fratta and P. M. Kogge, "Design enhancements for in-cache computations," in *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, vol. 1, 2009.

[143] F. Duarte and S. Wong, "Cache-based memory copy hardware accelerator for multi-core systems," *IEEE Transactions on Computers*, vol. 59, no. 11, pp. 1494–1507, 2010.

[144] T. N. Theis and P. M. Solomon, "It's time to reinvent the transistor!" *Science*, vol. 327, no. 5973, pp. 1600–1601, 2010.

[145] R. K. Cavin, V. V. Zhirnov, J. A. Hutchby, and G. I. Bourianoff, "Energy barriers, demons, and minimum energy operation of electronic devices," *Fluctuation and Noise letters*, vol. 5, no. 04, pp. C29–C38, 2005.

[146] V. V. Zhirnov, R. K. Cavin, J. A. Hutchby, and G. I. Bourianoff, "Limits to binary logic switch scaling-a gedanken model," *Proceedings of the IEEE*, vol. 91, no. 11, pp. 1934–1939, 2003.

[147] S. Banerjee, W Richardson, J Coleman, and A. Chatterjee, "A new three-terminal tunnel device," *IEEE Electron Device Letters*, vol. 8, no. 8, pp. 347–349, 1987.

[148] C. Hu, D. Chou, P. Patel, and A. Bowonder, "Green transistor-av dd scaling path for future low power ics," in *VLSI Technology, Systems and Applications, 2008. VLSI-TSA 2008. International Symposium on*, IEEE, 2008, pp. 14–15.

[149] K. Gopalakrishnan, P. B. Griffin, and J. D. Plummer, "Impact ionization mos (i-mos)-part i: Device and circuit simulations," *IEEE Transactions on electron devices*, vol. 52, no. 1, pp. 69–76, 2005.

[150] H. Kam and T.-J. K. Liu, "Pull-in and release voltage design for nanoelectromechanical field-effect transistors," *IEEE transactions on Electron Devices*, vol. 56, no. 12, pp. 3072–3082, 2009.

[151] M. Enachescu, M. Lefter, A. Bazigos, A. M. Ionescu, and S. D. Cotofana, "Ultra low power nemfet based logic," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, IEEE, 2013, pp. 566–569.

[152] J. Von Neumann, “Probabilistic logics and the synthesis of reliable organisms from unreliable components,” *Automata studies*, vol. 34, pp. 43–98, 1956.

[153] K Nikolic, A Sadek, and M Forshaw, “Architectures for reliable computing with unreliable nanodevices,” in *Nanotechnology, 2001. IEEE-NANO 2001. Proceedings of the 2001 1st IEEE Conference on*, IEEE, 2001, pp. 254–259.

[154] J. F. Wakerly, *Error detecting codes, self-checking circuits and applications*. North Holland, 1978, vol. 3.

[155] H. L. Garner, “Error codes for arithmetic operations,” *IEEE Transactions on Electronic Computers*, no. 5, pp. 763–770, 1966.

[156] W. W. Peterson, “On checking an adder,” *IBM Journal of Research and Development*, vol. 2, no. 2, pp. 166–168, 1958.

[157] D. T. Brown, “Error detecting and correcting binary codes for arithmetic operations,” *IRE Transactions on Electronic Computers*, no. 3, pp. 333–337, 1960.

[158] C.-K. Liu, “Error-correcting-codes in computer arithmetic,” DTIC Document, Tech. Rep., 1972.

[159] P. Forin, “Vital coded microprocessor principles and application for various transit systems,” *IFAC Control, Computers, Communications*, pp. 79–84, 1989.

[160] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer, “Anb-and anbdmem-encoding: Detecting hardware errors in software,” in *International Conference on Computer Safety, Reliability, and Security*, Springer, 2010, pp. 169–182.

[161] U. Wappler and C. Fetzer, “Hardware failure virtualization via software encoded processing,” in *Industrial Informatics, 2007 5th IEEE International Conference on*, IEEE, vol. 2, 2007, pp. 977–982.

[162] C. Fetzer, U. Schiffel, and M. Süßkraut, “An-encoding compiler: Building safety-critical systems with commodity hardware,” in *International Conference on Computer Safety, Reliability, and Security*, Springer, 2009, pp. 283–296.

[163] Y. Sun, M. Zhang, S. Li, and Y. Zhao, “Cost effective soft error mitigation for parallel adders by exploiting inherent redundancy,” in *IC Design and Technology (ICICDT), 2010 IEEE International Conference on*, IEEE, 2010, pp. 224–227.

[164] B. W. Johnson, J. H. Aylor, and H. H. Hana, “Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit vlsi adder,” *Journal of solid-state circuits*, vol. 23, no. 1, pp. 208–215, 1988.

[165] Y.-M. Hsu and E. Swartzlander, "Time redundant error correcting adders and multipliers," in *Defect and Fault Tolerance in VLSI Systems, Proceedings., International Workshop on*, IEEE, 1992, pp. 247–256.

[166] M. Nicolaidis, "Carry checking/parity prediction adders and alus," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 1, pp. 121–128, 2003.

[167] A. Pan, J. W. Tschanz, and S. Kundu, "A low cost scheme for reducing silent data corruption in large arithmetic circuits," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*, IEEE, 2008, pp. 343–351.

[168] M. Nicolaidis and H. Bederr, "Efficient implementations of self-checking multiply and divide arrays," in *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings.*, IEEE, 1994, pp. 574–579.

[169] M. Nicolaidis and R. Duarte, "Design of fault-secure parity-prediction booth multipliers," in *Design, Automation and Test in Europe, 1998., Proceedings*, IEEE, 1998, pp. 7–14.

[170] D. Marienfeld, E. S. Sogomonyan, V. Ocheretnij, and M Gossel, "New self-checking output-duplicated booth multiplier with high fault coverage for soft errors," in *Test Symposium, 2005. Proceedings. 14th Asian*, IEEE, 2005, pp. 76–81.

[171] S. Peng and R. Manohar, "Fault tolerant asynchronous adder through dynamic self-reconfiguration," in *Computer Design: VLSI in Computers and Processors, ICCD. Proceedings.International Conference on*, IEEE, 2005, pp. 171–178.

[172] D. P. Vasudevan and P. K. Lala, "A technique for modular design of self-checking carry-select adder," in *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on*, IEEE, 2005, pp. 325–333.

[173] W. Rao, A. Orailoglu, and R. Karri, "Fault identification in reconfigurable carry lookahead adders targeting nanoelectronic fabrics," in *Test Symposium, 2006. ETS'06. Eleventh IEEE European*, IEEE, 2006, pp. 63–68.

[174] S. Ghosh, P. Ndai, and K. Roy, "A novel low overhead fault tolerant kogge-stone adder using adaptive clocking," in *Design, Automation and Test in Europe, 2008. DATE'08*, IEEE, 2008, pp. 366–371.

[175] W. Rao and A. Orailoglu, "Towards fault tolerant parallel prefix adders in nanoelectronic systems," in *Design, Automation and Test in Europe, 2008. DATE'08*, IEEE, 2008, pp. 360–365.

[176] M. Valinataj and S. Safari, "Fault tolerant arithmetic operations with multiple error detection and correction," in *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT'07. 22nd IEEE International Symposium on*, IEEE, 2007, pp. 188–196.

[177] E. Krekhov, A.-r. A. Pavlov, A. Pavlov, P. Pavlov, D. Smirnov, A. Tsar'kov, P. Chistopol'skii, A. Shandrikov, B. Sharikov, and D. Yakimov, "A method of monitoring execution of arithmetic operations on computers in computerized monitoring and measuring systems," *Measurement Techniques*, vol. 51, no. 3, pp. 237–241, 2008.

[178] J Mathew, S Banerjee, P Mahesh, D. Pradhan, A. Jabir, and S. Mohanty, "Multiple bit error detection and correction in gf arithmetic circuits," in *Electronic System Design (ISED), 2010 International Symposium on*, IEEE, 2010, pp. 101–106.

[179] O. Keren, I. Levin, V. Ostrovsky, and B. Abramov, "Arbitrary error detection in combinational circuits by using partitioning," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*, IEEE, 2008, pp. 361–369.

[180] S. Dolev, S. Frenkel, D. E. Tamir, and V. Sinelnikov, "Preserving hamming distance in arithmetic and logical operations," *Journal of Electronic Testing*, vol. 29, no. 6, pp. 903–907, 2013.

[181] N. Banerjee, C. Augustine, and K. Roy, "Fault-tolerance with graceful degradation in quality: A design methodology and its application to digital signal processing systems," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*, IEEE, 2008, pp. 323–331.

[182] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke, "Cost-efficient soft error protection for embedded microprocessors," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, ACM, 2006, pp. 421–431.

[183] H. Tabkhi, S. G. Miremadi, and A. Ejlali, "An asymmetric checkpointing and rollback error recovery scheme for embedded processors," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*, IEEE, 2008, pp. 445–453.

[184] F. J. MacWilliams and N. J. A. Sloane, *The theory of error-correcting codes*. Elsevier, 1977.

[185] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, *et al.*, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, IEEE, 2003, pp. 7–18.

[186] M. S. Gupta, K. K. Rangan, M. D. Smith, G.-Y. Wei, and D. Brooks, "Decor: A delayed commit and rollback mechanism for handling inductive noise in processors," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, IEEE, 2008, pp. 381–392.

[187] T. M. Austin, "Diva: A reliable substrate for deep submicron microarchitecture design," in *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, IEEE, 1999, pp. 196–207.

[188] D. Lipetz and E. Schwarz, "Self checking in current floating-point units," in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, IEEE, 2011, pp. 73–76.

[189] IBM, "Ibm power system e880 server, an ibm power8 technology-based system, addresses the requirements of an industry-leading enterprise class system," 2014.

[190] S. Carlough, A. Collura, S. Mueller, and M. Kroener, "The ibm zenterprise-196 decimal floating-point accelerator," in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, IEEE, 2011, pp. 139–146.

[191] M. J. Boersma and J. Haess, *Residue-based error detection for a processor execution unit that supports vector operations*, US Patent 8,984,039, 2015.

[192] Z. Sperber, O. Levy, M. Mishaeli, and R. Gabor, *Recoverable parity and residue error*, US Patent 8,909,988, 2014.

[193] S. Iacobovici, *End-to-end residue based protection of an execution pipeline*, US Patent 7,555,692, 2009.

[194] S. Srikanth, B. Deng, and T. M. Conte, "A brief survey of non-residue based computational error correction," *ArXiv preprint arXiv:1611.03099*, 2016.

[195] R. Chokshi, K. S. Berezowski, A. Shrivastava, and S. J. Piestrak, "Exploiting residue number system for power-efficient digital signal processing in embedded processors," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, ACM, 2009, pp. 19–28.

[196] J Ramirez, A Garcia, S Lopez-Buedo, and A Lloris, "Rns-enabled digital signal processor design," *Electronics Letters*, vol. 38, no. 6, pp. 266–268, 2002.

[197] E. D. Di Claudio, F. Piazza, and G. Orlandi, “Fast combinatorial rns processors for dsp applications,” *IEEE transactions on computers*, vol. 44, no. 5, pp. 624–633, 1995.

[198] J.-C. Bajard and L. Imbert, “A full rns implementation of rsa,” *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 769–774, 2004.

[199] S.-M. Yen, S. Kim, S. Lim, and S.-J. Moon, “Rsa speedup with chinese remainder theorem immune against hardware fault cryptanalysis,” *IEEE Transactions on computers*, vol. 52, no. 4, pp. 461–472, 2003.

[200] C. Y. Hung and B. Parhami, “Fast rns division algorithms for fixed divisors with application to rsa encryption,” *Information Processing Letters*, vol. 51, no. 4, pp. 163–169, 1994.

[201] R. W. Watson and C. W. Hastings, “Self-checked computation using residue arithmetic,” *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1920–1931, 1966.

[202] V. T. Goh and M. U. Siddiqi, “Multiple error detection and correction based on redundant residue number systems,” *IEEE Transactions on Communications*, vol. 56, no. 3, 2008.

[203] J.-S. Chiang and M. Lu, “Floating-point numbers in residue number systems,” *Computers &amp; Mathematics with Applications*, vol. 22, no. 10, pp. 127–140, 1991.

[204] B. Cao, C.-H. Chang, and T. Srikanthan, “A residue-to-binary converter for a new five-moduli set,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 5, pp. 1041–1049, 2007.

[205] P. V. A. Mohan, “Rns-to-binary converter for a new three-moduli set,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 9, pp. 775–779, 2007.

[206] G. Cardarilli, M Re, and R Lojacono, “Rns-to-binary conversion for efficient vlsi implementation,” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 45, no. 6, pp. 667–669, 1998.

[207] T. F. Tay and C.-H. Chang, “A non-iterative multiple residue digit error detection and correction algorithm in rrns,” *IEEE transactions on computers*, vol. 65, no. 2, pp. 396–408, 2016.

[208] J.-C. Bajard, J. Eynard, and N. Merkiche, “Multi-fault attack detection for rns cryptographic architecture,” in *Computer Arithmetic (ARITH), 2016 IEEE 23nd Symposium on*, IEEE, 2016, pp. 16–23.

[209] H. Xiao, H. K. Garg, J. Hu, and G. Xiao, "New error control algorithms for residue number system codes," *ETRI Journal*, vol. 38, no. 2, pp. 326–336, 2016.

[210] L. Xiao and X.-G. Xia, "Error correction in polynomial remainder codes with non-pairwise coprime moduli and robust chinese remainder theorem for polynomials," *IEEE Transactions on Communications*, vol. 63, no. 3, pp. 605–616, 2015.

[211] C.-H. Chang, A. S. Molahosseini, A. A. E. Zarandi, and T. F. Tay, "Residue number systems: A new paradigm to datapath optimization for low-power and high-performance digital signal processing applications," *IEEE circuits and systems magazine*, vol. 15, no. 4, pp. 26–44, 2015.

[212] T. F. Tay and C.-H. Chang, "A new algorithm for single residue digit error correction in redundant residue number system," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, IEEE, 2014, pp. 1748–1751.

[213] P. Yin and L. Li, "A new algorithm for single error correction in rrns," in *Communications, Circuits and Systems (ICCCAS), 2013 International Conference on*, IEEE, vol. 2, 2013, pp. 178–181.

[214] H.-Y. Lo and T.-W. Lin, "Parallel algorithms for residue scaling and error correction in residue arithmetic," *Wireless Engineering and Technology*, vol. 4, no. 04, p. 198, 2013.

[215] A. Sengupta and B. Natarajan, "Performance of systematic rrns based space-time block codes with probability-aware adaptive demapping," *IEEE Transactions on Wireless Communications*, vol. 12, no. 5, pp. 2458–2469, 2013.

[216] N. Z. Haron and S. Hamdioui, "Redundant residue number system code for fault-tolerant hybrid memories," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 7, no. 1, p. 4, 2011.

[217] Y. Tang, E. Boutillon, C. Jégo, and M. Jézéquel, "A new single-error correction scheme based on self-diagnosis residue number arithmetic," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, IEEE, 2010, pp. 27–33.

[218] A. Sweidan and A. A. Hiasat, "On the theory of error control based on moduli with common factors," *Reliable computing*, vol. 7, no. 3, pp. 209–218, 2001.

[219] O. Goldreich, D. Ron, and M. Sudan, "Chinese remaindering with errors," in *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, ACM, 1999, pp. 225–234.

[220] R. S. Katti, “A new residue arithmetic error correction scheme,” *IEEE transactions on computers*, vol. 45, no. 1, pp. 13–19, 1996.

[221] H. Krishna, B. Krishna, K.-Y. Lin, and J.-D. Sun, *Computational Number Theory and Digital Signal Processing: Fast Algorithms and Error Control Techniques*. CRC Press, 1994, vol. 6.

[222] E. D. Di Claudio, G. Orlandi, and F. Piazza, “A systolic redundant residue arithmetic error correction circuit,” *IEEE Transactions on Computers*, vol. 42, no. 4, pp. 427–432, 1993.

[223] H. Krishna, K.-Y. Lin, and J.-D. Sun, “A coding theory approach to error control in redundant residue number systems. i. theory and single error correction,” *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 1, pp. 8–17, 1992.

[224] J.-D. Sun and H. Krishna, “A coding theory approach to error control in redundant residue number systems. ii. multiple error detection and correction,” *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 1, pp. 18–34, 1992.

[225] J.-D. Sun, H. Krishna, and K. Lin, “A superfast algorithm for single-error correction in rrns and hardware implementation,” in *Circuits and Systems, 1992. ISCAS’92. Proceedings., 1992 IEEE International Symposium on*, IEEE, vol. 2, 1992, pp. 795–798.

[226] G. A. Orton, L. E. Peppard, and S. E. Tavares, “New fault tolerant techniques for residue number systems,” *IEEE transactions on computers*, vol. 41, no. 11, pp. 1453–1464, 1992.

[227] C.-C. Su and H.-Y. Lo, “An algorithm for scaling and single residue error correction in residue number systems,” *IEEE Transactions on Computers*, vol. 39, no. 8, pp. 1053–1064, 1990.

[228] V. Ramachandran, “Single residue error correction in residue number systems,” *IEEE transactions on computers*, vol. 32, no. 5, pp. 504–507, 1983.

[229] M Etzel and W Jenkins, “Redundant residue number systems for error detection and correction in digital filters,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 5, pp. 538–545, 1980.

[230] F. Barsi and P. Maestrini, “Error detection and correction by product codes in residue number systems,” *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 915–924, 1974.

[231] S.-S. Yau and Y.-C. Liu, "Error correction in redundant residue number systems," *IEEE Transactions on Computers*, vol. 100, no. 1, pp. 5–11, 1973.

[232] T. R. Rao, "Biresidue error-correcting codes for computer arithmetic," *IEEE Transactions on computers*, vol. 100, no. 5, pp. 398–402, 1970.

[233] N. S. Szabo and R. I. Tanaka, *Residue arithmetic and its applications to computer technology*. McGraw-Hill, 1967.

[234] S Talahmeh and P Siy, "Arithmetic division in rns using galois field gf (p)," *Computers &amp; Mathematics with Applications*, vol. 39, no. 5-6, pp. 227–238, 2000.

[235] A. A. Hiasat and H. Abdel-Aty-Zohdy, "Semi-custom vlsi design and implementation of a new efficient rns division algorithm," *The Computer Journal*, vol. 42, no. 3, pp. 232–240, 1999.

[236] J.-C. Bajard, L.-S. Didier, and J.-M. Muller, "A new euclidean division algorithm for residue number systems," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 19, no. 2, pp. 167–178, 1998.

[237] M. Lu and J.-S. Chiang, "A novel division algorithm for the residue number system," *IEEE Transactions on Computers*, vol. 41, no. 8, pp. 1026–1032, 1992.

[238] D. Gamberger, "New approach to integer division in residue number systems," in *Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on*, IEEE, 1991, pp. 84–91.

[239] M. A. Hitz and E. Kaltofen, "Integer division in residue number systems," *IEEE transactions on computers*, vol. 44, no. 8, pp. 983–989, 1995.

[240] Y. Shimazaki, R. Zlatanovici, and B. Nikolic, "A shared-well dual-supply-voltage 64-bit alu," *Journal of Solid-State Circuits*, vol. 39, no. 3, pp. 494–500, 2004.

[241] S. Rusu, "Multi-domain processors design overview," 2010.

[242] A. Preethy and D Radhakrishnan, "A 36-bit balanced moduli mac architecture," in *Circuits and Systems, 1999. 42nd Midwest Symposium on*, IEEE, vol. 1, 1999, pp. 380–383.

[243] G Norman, *Einspruch, vlsi handbook*, 1985.

[244] J. Tan and O. Rosen, *Process for determining competing cause event probability and/or system availability during the simultaneous occurrence of multiple events*, US Patent App. 10/272,156, 2004.

[245] K. C. Gower, B. Hazelzet, M. W. Kellogg, and D. J. Perlman, *High reliability memory module with a fault tolerant address and command bus*, US Patent 7,234,099, 2007.

[246] M. Sachdev, *Fault-tolerant memory address decoder*, US Patent 5,831,986, 1998.

[247] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 89–104, 2002.

[248] M. Talluri and M. D. Hill, *Surpassing the TLB performance of superpages with less operating system support*, 11. ACM, 1994, vol. 29.

[249] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 449–464, 2008.

[250] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A dram page-mode scheduling policy for the many-core era," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2011, pp. 24–35.

[251] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, ACM, vol. 40, 2005, pp. 190–200.

[252] S. CPU2006, *Standard performance evaluation corporation*, 2006.

[253] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[254] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, IEEE, 2009, pp. 469–480.

[255] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, IEEE, 2011, pp. 694–701.

[256] K. Bains, J. Halbert, C. Mozak, T. Schoenborn, and Z. Greenfield, *Row hammer refresh command*, US Patent 9,117,544, 2015.

[257] B. T. Bennett and V. J. Kruskal, “Lru stack processing,” *IBM Journal of Research and Development*, vol. 19, no. 4, pp. 353–357, 1975.

[258] C. CaBcaval and D. A. Padua, “Estimating cache misses and locality using stack distances,” in *Proceedings of the 17th annual international conference on Supercomputing*, ACM, 2003, pp. 150–159.

[259] J. H. Patel and L. Y. Fung, “Concurrent error detection in alu’s by recomputing with shifted operands,” *IEEE Trans. Computers*, vol. 31, no. 7, pp. 589–595, 1982.

[260] D. P. Vasudevan, P. K. Lala, and J. P. Parkerson, “Self-checking carry-select adder design based on two-rail encoding,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 12, pp. 2696–2705, 2007.

[261] E. Mizan, T. Amimeur, and M. F. Jacome, “Self-imposed temporal redundancy: An efficient technique to enhance the reliability of pipelined functional units,” in *Computer Architecture and High Performance Computing, SBAC-PAD. 19th International Symposium on*, IEEE, 2007, pp. 45–53.

VITA

Sriseshan Srikanth is a PhD candidate in the School of Computer Science at Georgia Tech, advised by Prof. Thomas M. Conte and his primary research field is computer architecture. His research interests include on-chip and off-chip acceleration of sparse data applications, and computationally-resilient, energy-efficient processing of dense data applications. His other research interests have included memory controller scheduling policies to lower DRAM latency (Intel labs), compiler assisted reconfiguration of shared multi-processors via function unit power gating (Georgia Tech) and hardware/software support for thread level speculation (IIT Madras and University of Alberta). Prior to joining Georgia Tech, he received his dual degree in Electrical Engineering from IIT Madras.