CACHE-AWARE IMPLEMENTATION OF COMPRESSED SENSING RECONSTRUCTION USING WALSH-HADAMARD TRANSFORM ON MULTICORE ARCHITECTURE

by

Aneesh Ramgopal

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2014

Approved by:

_____
Dr. Bharat Joshi

_____
Dr. Michael Fiddy

_____
Dr. Arindam Mukherjee

ABSTRACT

ANEESH RAMGOPAL. An efficient multicore implementation of compressed sensing reconstruction using Walsh-Hadamard transform. (Under the direction of Dr. BHARAT JOSHI)

The recent development of the compressed sensing (CS) theory has given rise to several algorithms being proposed for signal acquisition and reconstruction. Implementing these algorithms are computationally demanding and pose several challenges for effective shared resource utilization. We describe a native, parallelized realization of the compressed sensing problem on a commercially available multicore architecture. A quick and efficient reconstruction algorithm, Smoothed L0 (SL0), is parallelized and adapted to benefit from the multicore implementation of the sampling basis, the Walsh-Hadamard transform (WHT).

Valuable insights on data cache locality are presented from the characterization of miss rate patterns using a cache profiler. We develop performance models for the algorithms using regression, correlating response time with application parameters, memory utilization and other overheads. The matrix generation algorithm shows a high degree of parallelizability with speedup up to 5.6 on an 8-core Intel Xeon, while the recovery algorithm shows a speedup up to 4.5. Our models also demonstrate the synchronization bottlenecks and cache limitations of such threaded multicore implementations.

ACKNOWLEDGMENTS

I would like to take this opportunity to thank several people who helped, contributed and enabled me to successfully complete this research. Firstly, I would like to thank my advisor Dr. Bharat Joshi and Dr. Michael Fiddy for their invaluable support to assist and direct the research. I would also like to thank Dr. Arindam Mukherjee for his helpful advice while serving on the committee. Secondly, I would like to thank my research colleagues Ashish Panday and Akhil Arunkumar for their interesting ideas and productive conversations. Lastly, I would like to thank my parents for always being there with me, to help accomplish my goals.

DEDICATION

This research is humbly dedicated to Swami.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CS	Compressed sensing or compressive sampling
WHT	Walsh-Hadamard transform
DFT	Discrete Fourier transform
FFT	Fast Fourier transform
SL0	Smoothed L0

CHAPTER 1. INTRODUCTION

Compressed sensing has led to a paradigm shift in the field of signal acquisition and reconstruction due to its significant implications in diverse applications. The central idea behind compressed sensing is to sample inherently ‘sparse’ signals below the Nyquist rate and still adequately reconstruct the signal. Capturing signals or images that are compressively sensed has necessitated utilization of suitable patterns in the acquisition stage. The choice of such basis patterns largely determine the ability to extract necessary features and the resultant image quality. In imaging systems, such as magnetic resonance imaging, images are encoded with sinusoids, essentially amounting to collecting Fourier coefficients. Generating these Fourier transform matrix values are time-consuming and, therefore, one of the primary objectives of this thesis is to look at an efficient algorithm to generate these large matrices on a symmetrical multiprocessing (SMP) system. The optimizations that enable the proposed algorithm to exploit the characteristics of a multilevel hierarchical cache memory are also explored.

Arriving at sparse solutions from an underdetermined system of equations has led to development of suitable approaches, requiring a computationally involved process called reconstruction. There has been considerable interest towards improving algorithmic performance for reconstruction on modern multicore systems which is another objective of this research.

Chapter 2 introduces some of the basic concepts of compressed sensing, an overview of the coding functions and the specific advantages of the Fourier transform variant employed. Furthermore, the chapter delves into the process of CS reconstruction and the relevance of the particular technique used in this thesis. Chapter 3 describes the algorithms used for matrix generation and reconstruction as well as a brief overview of the Intel architecture, which is used to implement and analyze these algorithms. Chapter 4 reports on the efficiency of the algorithm's implementation on the hardware, described by profiling for cache performance and through regression modeling of the relevant parameters.

CHAPTER 2. COMPRESSED SENSING

Nearly every conventional form of signal acquisition protocol used in modern-day communications, audio/video electronics, medical imaging etc. is dictated by the Shannon-Nyquist theorem – that is the sampling rate must be at least twice the maximum signal frequency or the Nyquist rate. Compressed sensing (interchangeably referred to as compressive sampling and abbreviated as CS) is a novel sensing idea that goes against the grain by challenging the traditional sampling methodologies.

2.1. Overview

CS theory asserts that it is possible to recover signals from far lesser samples or measurements in comparison to the Nyquist criterion. Essential to this assertion are the dual principles upon which lay the foundations for CS – sparsity and incoherence.

- *Sparsity* of an image represents the notion that an image is inherently sparse or that the essential information contained in an image is generally concentrated in fewer concise data points. This means that a natural image is compressible when it has a sparse representation using an appropriate basis.
- *Incoherence* represents the notion that the sparse representation of the signal in a certain basis implies that this sampling basis is incoherent with the signal of interest. This also expresses the duality between the time and frequency domain signal representations, for instance, the time-domain Dirac delta spreads out in the

frequency domain. In other words, the sampling basis should be dense in order to sufficiently correlate with the sparse representative basis.

Essentially, compressed sensing relies on the inherent sparsity of signals and the incoherence between the representation basis and the sensing basis. Mathematically speaking, a signal represented by $x \in \mathbb{R}^n$, which is an $N \times 1$ vector, a sampling basis A, which is an $m \times N$ matrix appropriately chosen from the sensing basis Φ, and the sampled signal $y \in \mathbb{R}^n$, which is an $m \times 1$ vector, can be depicted as:

$$y = Ax$$

Here $m << N$, where N is the total number of signal points in the source signal and m is the number of measurements. The representative basis Ψ induces sparsity in the signal and this continuous-time signal is represented in a discretized space as the S-sparse vector x, that is it consists of S non-zero entities and $S << N$. The sensing matrix A consists of the row vectors $\varphi_1 \ldots \varphi_m$. Since there are m measurements of y from the $N \times 1$ vector x, the acquisition of the signal accomplishes sampling and compression in a single step. The coherence between the representation basis Ψ and the sensing basis Φ, both of which are considered to be orthonormal bases in $\mathbb{R}^n$, can be defined as:

$$\mu(\Phi, \Psi) = \sqrt{n} \cdot \max_{1 \le k, j \le n} \left| \langle \phi_k, \psi_j \rangle \right|$$

This calculates the maximum coherence between any two elements of the bases Φ and Ψ. The possible values of this coherence pair lies in the range $[1, \sqrt{n}]$. If $\mu(\Phi, \Psi) = 1$, the bases are maximally incoherent. This means that for a low coherence value if the signal is sparse in Ψ, it would have a dense representation in Φ. This necessitates identification of suitable incoherent pairs of bases for the purposes of compressed sensing. Some examples of incoherent Φ-Ψ pairs are a spike-Fourier basis, wavelet-noiselet basis and a

random IID matrix paired with a fixed basis. Here, IID refers to an independent and identically distributed matrix. The advantage to using a random matrix that is, for example, Gaussian or binary ±1, is that it is quite incoherent with any fixed representative basis Ψ [1]. In addition to this, a ±1 matrix can be stored in a compact data structure and manipulated efficiently using appropriate code transformations. Hence, this work explores generating the ±1 sampling matrix from a Walsh-Hadamard Transformation matrix.

2.2. Walsh-Hadamard Transform

The Walsh-Hadamard Transform (also known as the Hadamard Transform and abbreviated herewith as WHT) is a digital signal processing (DSP) transform with applications in video compression, data encryption and quantum computing. It performs symmetric, orthogonal, linear operations on a set of real numbers that are in powers of two. The WHT can be constructed using several size-2 discrete Fourier transforms (DFT). The WHT of size N, represented by WHT_N, is an $N \times N$ matrix and N takes powers of two values i.e. $N = 2, 4, 8 \ldots$ The WHT matrix is composed of only ±1 values and the basic 2 × 2 WHT construct is depicted as:

$$\mathrm{WHT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The generalized form for WHT can be written using the Kronecker product, denoted by the symbol $\otimes$, as follows:

$$\mathrm{WHT}_N = \bigotimes_{i=1}^{n} \mathrm{WHT}_2 = \mathrm{WHT}_2 \otimes \ldots \otimes \mathrm{WHT}_2$$

Here, $N = 2^n$. The Kronecker product, which is a direct matrix product, of two matrices A and B is defined as:

$$A \otimes B = [a_{ij}B], \quad \text{where} \quad A = [a_{ij}]$$

For instance,

$$\mathrm{WHT}_4 = \mathrm{WHT}_2 \otimes \mathrm{WHT}_2$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

And,

$$\mathrm{WHT}_8 = \mathrm{WHT}_2 \otimes \mathrm{WHT}_2 \otimes \mathrm{WHT}_2 = \mathrm{WHT}_4 \otimes \mathrm{WHT}_2$$

Thus, the higher indices can be expressed as a Kronecker product of the lower indices recursively generating the result. This recursive process, known as the Cooley Tukey Algorithm, can be optimized to work with modern memory systems and will be detailed in the algorithm section 3.1. It can also be noticed that each row of the WHT matrix are orthonormal and they have uniform distribution, befitting the criteria to be a CS matrix that can show incoherence with any fixed basis. Hence, the chosen sampling matrix consists of *m* rows of the WHT, where *m* is much smaller than *N*. The requirement for the number of measurements, *m*, is decided beforehand and can be delivered to the acquisition system after parallel generation of the necessary pattern. This matrix pattern is optically represented using opaque/black (0) and transparent/white (1), and the setup, as depicted in FIGURE 1, is used in single pixel cameras for compressively sampling images [2]. It uses bi-convex lenses to focus the incident light from the image onto a digital micro-mirror device (DMD) and again onto a single photodiode. The DMD's tiny mirrors are programmed to black or white with the rows of the sampling matrix. Another aperture assembly alternative to DMD is a lens less device that uses a transparent monochromatic LCD.

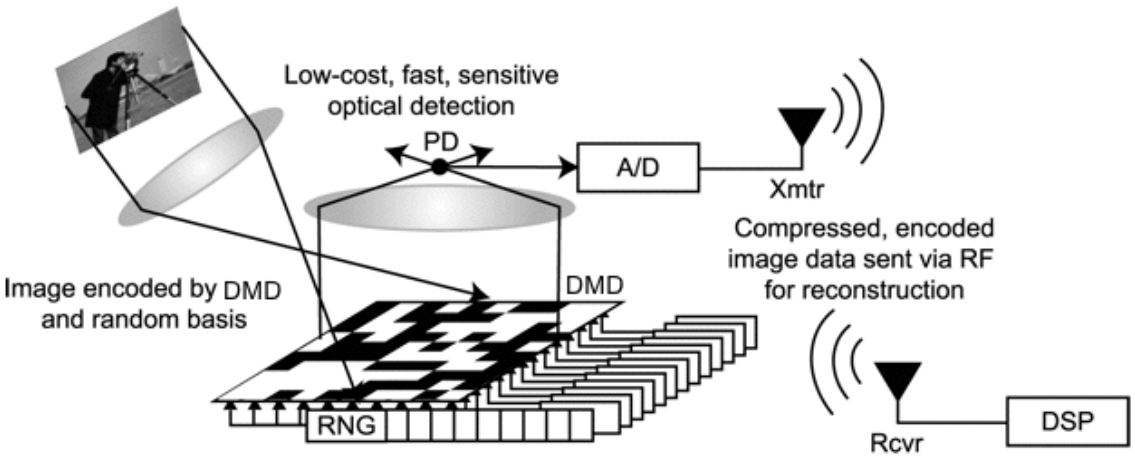

FIGURE 1: Single-pixel camera setup [2]

2.3. Reconstruction Techniques

While the acquisition stage is a linear process, the CS recovery procedure is decidedly non-linear, as it shall be seen here. The process of reconstructing a compressively sampled signal involves using the sampled vector y with m compressive measurements to recover x^* which has N signal points, i.e. solving for x^* in the equation, $Ax^* = y$, where A is the sampling matrix. Since, the number of measurements taken are far lesser than number of data points, the equation above is underdetermined and there exists infinitely many solutions for x^*. So, reconstruction requires solving an optimization or a minimization problem within a feasible region.

Before discussing the available minimization problems, a quick overview of the vector norms is in order. A norm assigns a positive magnitude on each vector in vector space. Starting with the well-known Euclidean norm defined over the n-dimensional Euclidean $\mathbb{R}^n$ space is:

$$\|x\|_2 = \sqrt{x_1^2 + \cdots + x_n^2}$$

This is the Euclidean distance of the vector *x* from the origin, also known as the ℓ_2-distance or the ℓ_2-norm. This can be extended to an ℓ_p-norm defined as:

$$\|x\|_p = \sqrt[p]{|x_1|^p + \cdots + |x_n|^p}$$

It can further be seen that for $p = 1$, the ℓ_1-norm defined for *x* is a summation of the absolute values. This is also known as the Manhattan distance. For $p = 0$, the so-called ℓ_0-norm is not homogeneous and is defined as follows:

$$\min \|x\| \rightarrow \|x\|_0 = \#\,(i\ |x_i^* \neq 0)$$

This indicates that the ℓ_0-norm finds the number of non-zero entries in the vector *x*. This non-zero counting norm can find sparse solutions useful for CS recovery.

Given a known $m \times 1$ vector *y* and a matrix *A* with dimensions $m \times N$ where $m < N$, an optimal solution is needed for x^*, an $N \times 1$ signal being recovered from an *S*-sparse signal *x*, by solving the constraint minimization problem:

$$\min\|x^*\|_0 \quad \text{subject to} \quad A_{m\times N}x^*_{N\times 1} = y_{m\times 1}$$

This is the ℓ_0-norm minimization, which indicates the number of non-zero components in the vector. Hence, finding a solution for x^* requires locating the *S* non-zero entities of the original vector *x*. There are two relevant criteria for reconstructing from an IID matrix (a Walsh-Hadamard matrix in this work):

- The sensing matrix *A* can guarantee discriminating *S*-sparse signals from *m* compressive measurements provided that all subsets of *S*-columns taken from *A* are nearly orthogonal. This condition is known as the Restricted Isometry Property (RIP).

- These matrices can demonstrably reconstruct *S*-sparse vectors from m samples provided that:

$$m \geq C.S \log \frac{N}{S}, \qquad \text{where } C \text{ is a } +\text{ve constant}$$

One method of achieving reconstruction using minimization, is by minimizing the ℓ_0-norm of the vector x^* as shown above. Another common technique is the ℓ_1-norm minimization that approximates the ℓ_0-minimizer's solution. The ℓ_1-minimization is a convex optimization problem with a feasible set that is convex as well. Whereas, the ℓ_0-minimization is non-convex, that is, it has numerous local minima and maxima. Finding the solution to a global optimization, such as the ℓ_0-minimization, is numerically challenging and is NP-hard. There are many deterministic as well as heuristic methods to numerically approximate the solution to this problem. Graduated optimization is one such heuristic strategy that attempts to arrive at the global optimum for the hard problem by starting at a conveniently easy or convex problem and gradually moving towards the harder problem by passing on the solution at each step to use as a starting point for the next step. This is also called a Graduated Non-Convexity procedure and forms the basis of the algorithm used for CS reconstruction in this research.

2.4. Related Work

Compressed sensing has been receiving a lot of attention from researchers ever since 2006 when Emmanuel Candès et al. proposed the theory in [3, 4] for recovering sparse signals from fewer samples than stipulated by the Shannon-Nyquist sampling theorem. This concept has found applications in several fields like information theory [5], machine learning [6], image processing [2, 7] among others. This research deals specifically with multicore CS processing and enables efficient computational realization

of CS imaging systems, such as single-pixel compressive sensing [8]. This work contributes the ability to service a large sampling pattern on-demand to a compressively sensed acquisition system using modern parallel computing devices.

Identifying matrices that are feasible for CS reconstruction are described in [3, 9, 10]. The suitability of using the Walsh-Hadamard matrix in this work is justified by [1], which describes WHT fitting the feasibility criteria of having nearly orthogonal rows and being largely incoherent with any fixed representative basis. Unlike other transforms that require double-precision floats, the binary ±1 valued WHT stores into integer or short integer variables, contributing to superior memory management and a performance boost due to integer operations. While the famous FFTW package realizes multicore DFT implementations on varying architectures [11, 12], specialized parallel packages of the WHT have been developed by the SPIRAL project [13]. While these packages are useful to perform DSP transforms, they are not optimized for forward and inverse transforms as a single, integrated program. This research focuses on an optimum parallel WHT implementation that uses minimal storage and is effectively manipulated for sparse recovery by exploiting the matrix's unique properties.

The sampling matrices chosen can be effective at CS recovery provided they can satisfy the Restricted Isometry Property (RIP) [14], which has formed the basis for many compressed sensing reconstruction algorithms like the Basis Pursuit (BP) [15] and Matching Pursuit (MP) [16] greedy algorithms, ℓ_1-norm based iterative thresholding algorithms [17] and others. Beyond the RIP criteria and the Donoho-Tanner phase transitions [18], other solvers have been devised using different heuristics that achieve better phase transitions. One such recent solver, which is used as the basis for the parallel

algorithm in this thesis, is the global optimization based Smoothed L0 (SL0) algorithm [19, 20], which has been shown to have improved performance with adaptive parameter selection in [21]. A random IID matrix, such as WHT, shows excellent convergence for the SL0 recovery and also reduces the number of matrix-vector multiplications, making it a suitable choice for this research. While CS reconstruction using ℓ_1-minimization techniques have been implemented on shared-memory multicore and vectorized GPU processors [22, 23], this research implements parallel SL0, a fast ℓ_0-minimization, with appropriate cache and algorithmic optimizations to envision real-time recovery of sparse images. Such realizations of compressed sensing favor uses in surveillance low-light imaging and feature extraction processing like template matching or motion detection.

CHAPTER 3. ALGORITHM STRATEGY AND DESIGN ANALYSIS

The late 1980's marked the arrival of the first shared memory architectures, spawning research leading up to development of parallelizing compilers today. These compilers, while providing good performance scaling for straightforward simple programs, are unable to achieve parallel speedup for more complicated programs like DFT or convex optimization. Moreover, image processing researchers use standard software packages, such as MATLAB, that do not support thread-core bindings or identify program-specific parallelization capabilities. It is for this reason that this research presents explicit methods to write fast parallel programs, which are invariably harder than their sequential counterparts. Generally, when writing parallel programs, the programmer needs to address:

- Load Balancing/Task Granularity – Equal sharing of the workload between processors. Sequential execution should be minimal as speedup is affected as per Amdahl's Law.
- Avoiding False-Sharing – Ensuring that processors do not share private data in the same cache line as this could severely impact performance due to cache thrashing.
- Synchronization Overhead – Ensuring that threads do not unduly wait at synchronization points by minimizing cache bandwidth usage and reducing simultaneous shared cache line accesses.
- Memory Hierarchy – Writing code that effectively utilizes temporal and spatial locality on shared and private caches.

3.1. Multicore Cooley-Tukey Algorithm

The Cooley-Tukey algorithm, in its recursive depth-first form, is a divide-and-conquer algorithm most commonly used to compute fast Fourier transform and its variants. It recursively re-expresses the discrete Fourier transform (DFT) as a product composed of two smaller DFTs, which are in turn computed recursively from two smaller DFTs, as well as the application of the twiddle factors. This algorithm can be applied to the Walsh-Hadamard transform by dividing into smaller sizes and computing upwards. It can be illustrated with the following equations using matrix formalism, manipulated to efficiently map on to multicore architectures.

$$y = \mathrm{WHT}_N x$$

$$\mathrm{WHT}_N = \mathrm{WHT}_R \otimes \mathrm{WHT}_S ,$$

where vector size is N and $N = R \times S$.

$$\mathrm{WHT}_N = (\mathrm{WHT}_R \otimes \mathrm{I}_S)(\mathrm{I}_R \otimes \mathrm{WHT}_S)$$

$$\mathrm{WHT}_N = \left(\left(\mathrm{WHT}_R \otimes \mathrm{I}_{\frac{S}{p}}\right) \otimes \mathrm{I}_p\right)\left(\mathrm{I}_p \otimes \left(\mathrm{I}_{\frac{R}{p}} \otimes \mathrm{WHT}_S\right)\right),$$

where p indicates the number of threads (= number of cores for 1 thread/core) and the Identity Matrix (I_x) can be represented for $x = 2$ as: $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. The above expression can be further rearranged using a stride permutation matrix, L^n_m, which is a matrix with m-stride accesses in an n-by-n matrix. For example,

$$\mathrm{L}^8_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Stride permutations can be used to dynamically permute data. For example, for an $n \times n$ matrix A,

$$A \otimes \mathrm{I}_m = \mathrm{L}_n^{mn} (\mathrm{I}_m \otimes A) \mathrm{L}_m^{mn}$$

The parallel expression for WHT can be further rearranged using a) parallel constructs as depicted in FIGURE 2; and b) a stride permutation matrix, L_m^n, which is a matrix with m-stride accesses in an n-by-n matrix. The first parallel construct is $(\mathrm{I}_p \otimes \mathrm{WHT}_\mu)$, which expresses block diagonal matrices with 'p' blocks i.e. could be mapped on to 'p' cores using embarrassingly parallel computations with μ WHT elements. Here, moving μ elements of WHT at a time, where μ is the number of elements in a cache line, ensures false sharing free usage patterns. For example, for an integer data type element, which is 4-bytes in size, μ = 16 for a 64-byte cache block size. This would mean that for $N \geq 64$, each processor in a quad-core system ($p = 4$) would control 16 or more WHT elements among the 64 elements in a row of the matrix. This implies the construct is communication free and avoids sharing cache lines between cores. The second is a vector construct of the form $(\mathrm{WHT}_\mu \otimes \mathrm{I}_p)$, which reorders blocks of μ consecutive elements, thus moving elements in the size of the cache line keeping communication overheads to a minimum. The stride permutation construct L_n^{mn} restructures the vector form to the parallel form as indicated in the equation above.

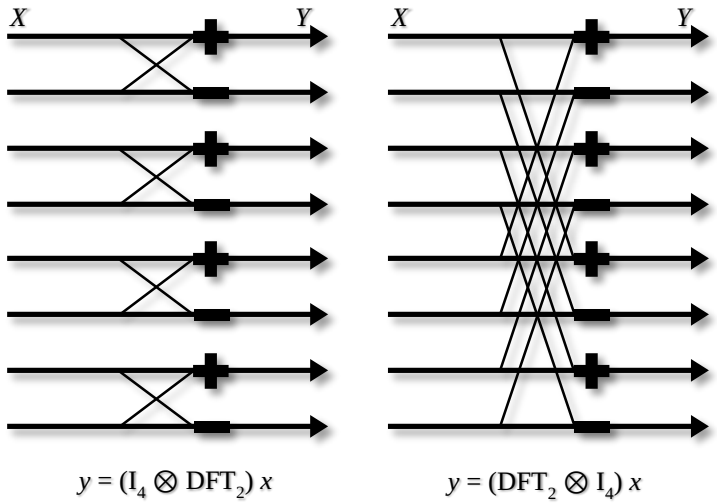

FIGURE 2: Parallel constructs

Using the parallel constructs and the stride construct, the recursive expression can be further expanded as:

$$\mathrm{WHT}_N = \mathrm{L}^{RS}_{\frac{RS}{p}}\left(\mathrm{I}_p \otimes \left(\mathrm{WHT}_R \otimes \mathrm{I}_{\frac{S}{p}}\right)\right)\mathrm{L}^{RS}_{p}\left(\mathrm{I}_p \otimes \left(\mathrm{I}_{\frac{R}{p}} \otimes \mathrm{WHT}_S\right)\right)$$

Finally, the above expression is modified to make it cache-aware, resulting in the expression for the Multicore Cooley-Tukey Algorithm:

$$\mathrm{WHT}_N = \left(\left(\mathrm{L}^{Rp}_{R} \otimes \mathrm{I}_{\frac{S}{p\mu}}\right) \otimes \mathrm{I}_{\mu}\right)\left(\mathrm{I}_p \otimes \left(\mathrm{WHT}_R \otimes \mathrm{I}_{\frac{S}{p}}\right)\right)\left(\left(\mathrm{L}^{Rp}_{p} \otimes \mathrm{I}_{\frac{S}{p\mu}}\right) \otimes \mathrm{I}_{\mu}\right)\left(\mathrm{I}_p \otimes \left(\mathrm{I}_{\frac{R}{p}} \otimes \mathrm{WHT}_S\right)\right)$$

The algorithmic implementation scheme of this equation is described as follows. A WHT of size *N* splits into WHT of sizes *R* and *S*, where *R* and *S* are powers-of-two factors of size *N* and $R \geq S$. As shown in FIGURE 3, this splitting process repeats itself in a depth-

first recursion until size-2 WHT is reached. As the nodes in bold letters indicate, it is only needed to compute the leftmost (or rightmost) nodes of the tree to reach all the way up to the root, since nodes at any particular height are exactly the same, hence requiring no additional storage or computational time. There is one variation, though, for odd power-of-two values of N; WHT_S is a sub matrix of WHT_R, in which case, the program control follows the sub-tree under the parent node of size R.

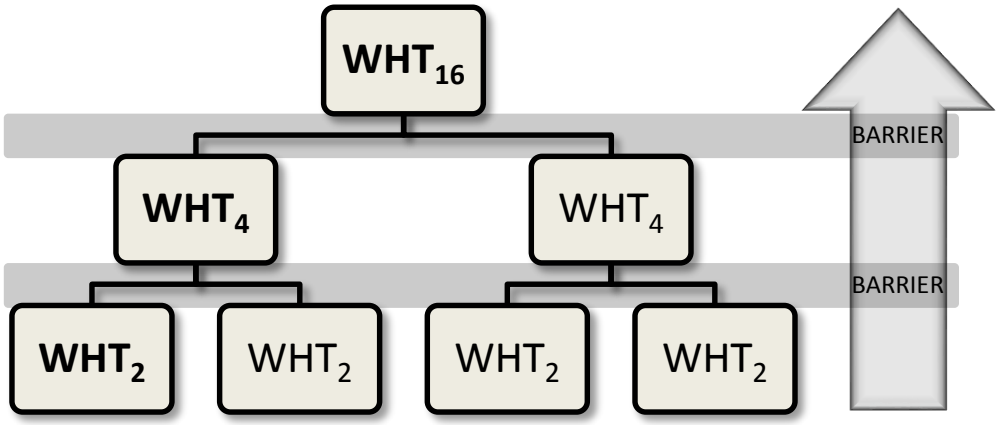

FIGURE 3: Depth-first recursion tree for $N = 16$

There are synchronization points at the end of each recursion step to ensure that all the cores are ready with the necessary data to move on to the next higher computational level. To understand the core-wise access workload distribution during computation, FIGURE 4 breaks down the inner workings of the algorithm for $N = R \times S$, i.e., $16 = 4 \times 4$ running on two threads. The progression on the right side (for $S = 4$) shows how each core obtains complete access to the WHT_4 matrix with the shaded letters in the last step indicating the work done by each core. The progression on the left side (for $R = 4$) is quite

different from the right side as it can be noticed that the accesses per core in the final step are at stride-4 column-wise and stride-2 row-wise. To rectify the strided accesses, the stride permutation L is used to convert the vector form into the parallel construct – $I_2 \otimes (WHT_4 \otimes I_2)$, with each core bearing the workload of the term within parentheses. It should be stressed that WHT_R and WHT_S are just used to refer to the same intermediate matrix and the resultant matrix, WHT_N, is stored in-place, with the operations taking place during computation described by the right and left figurative depictions.

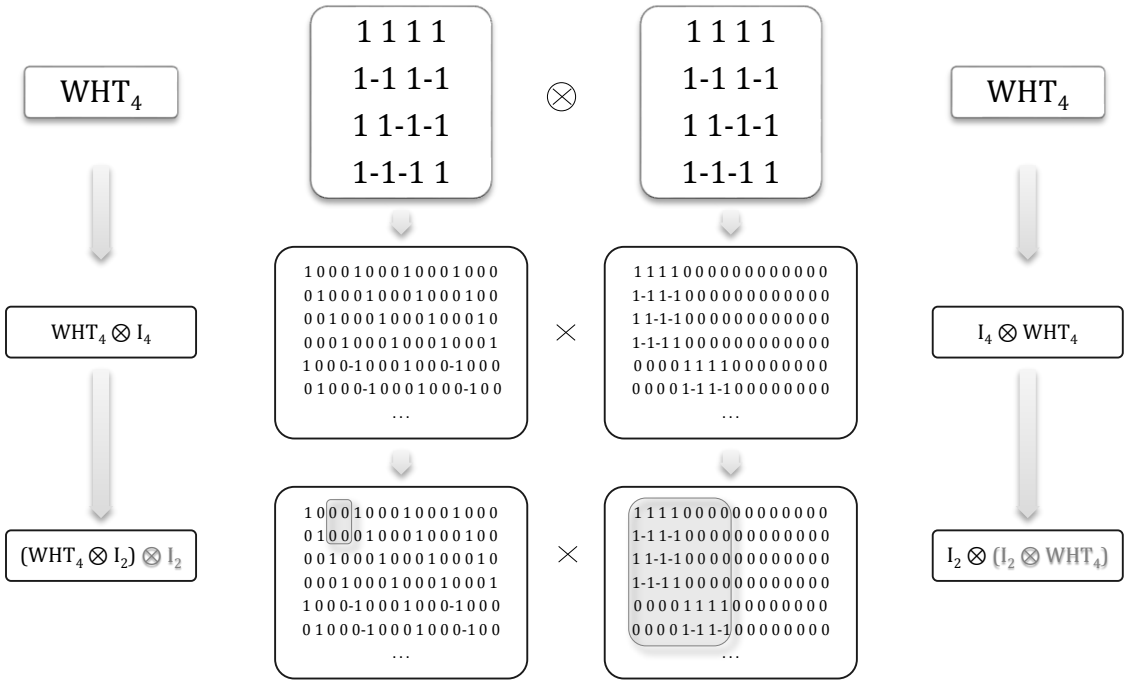

FIGURE 4: Illustration of workload distribution for $p = 2$, $R = 4$ and $S = 4$

Critical to the algorithm's performance on each of the cores is the cache access patterns when executing their individual workloads. While the WHT_S component is equally distributed to all the cores, each core accesses R/p columns of WHT_R, where 'p' is the

number of cores, assuming a single thread-core affinity. Simultaneously, the output matrix WHT_N is split vertically into $N \times N/p$ sub matrices. For each of the N^2/p iterations per core of the algorithm, there are at most three cache lines drawn. Essentially, every μ^{th} iteration needs two cache lines (i.e. a line each for WHT_S and WHT_N) with an extra cache line required after μ^2 iterations (i.e. a line for WHT_R). This works out to an average cache line requirement of $(2\mu+1)/\mu^2$ per iteration. Using FIGURE 4 as an example and assuming $\mu = 16$, it can be seen that a cache line is needed by each core about every 16^{th} iteration from a total of 128 iterations.

The average cache line requirement per iteration doesn't specify if the lines drawn were hits or misses, which would require knowledge of the underlying cache architecture. For a α-way set associative cache of size C, there are $\sigma = C / (\alpha\mu \cdot sizeofint)$ sets, where *sizeofint* is 4 bytes for a C++ *int* data type and 2 bytes for *short-int* on an Intel machine. As described above, wherein drawing utmost three (aligned) cache lines at a time, it would be advisable to use at least a 4-way or higher associativity. So, assuming $\alpha > 2$ and not accounting for compulsory misses, a good measure of keeping the miss rate down is through high associativity, i.e. $\mu^2 \leq \sigma\alpha$. So, it must be understood that in picking μ, a value too low reduces spatial locality while a large value increases conflict misses. For completely in-cache operations of the N^2/p elements per core, with $(\sigma\alpha)$ cache lines:

$$\sigma\alpha\mu < \frac{N^2}{p}$$

For a quad-core where each have a private L1 cache with two cores sharing an L2 cache, this would mean that sizes of $N > 2^7$ would be expected to miss a 32KB L1 private data cache and sizes of $N > 2^9$ would fill up a shared L2 cache of size 4 MB, assuming 50% of the program are data operations. To handle larger sizes more efficiently, this WHT

implementation also uses optimizations to localize cache accesses by exploiting the properties of the transform. Loop folding, or folding the WHT_S vertically reduces accesses by half. Loop interchange presents row-major matrix multiplication. Loop tiling allows utilizing the lines present in cache already by partitioning the iterations into chunks or blocks of 16 elements (assuming 64-byte cache lines with 4-byte elements).

This algorithm is quite similar in implementation to the fast Fourier transform, but it computes the Fourier basis rather than the transform. The depth-first recursion yields $\lceil \log \log N + 1 \rceil$ steps with the number of computations in each recursion step from the root to the leaf nodes following the progression $(N^2, \sqrt[2]{N}^2, \sqrt[4]{N}^2 \ldots)$. In effect, this gives a computational complexity of $O(N^2)$, with the rest of the terms being quite insignificant. If there are p cores, then the computations reduce to $O(N^2/p)$ per core. Hence, the speedup of the efficient parallel implementation over its sequential ($p = 1$) counterpart is ideally p.

In practice, there are significant inter processor communication overheads involved and the specific cache memory hierarchy that determine the actual speedup achieved. Task granularity and load balancing are decisive factors in understanding the parallel overheads incurred by the algorithm. The algorithm balances the workload evenly across all cores, assigning larger tasks to each core with increasing input size. This implies that increasing input size for a fixed number of threads leads to coarser granularity and decreasing corresponding communication overhead. For example, when computing the WHT for 4 cores: When $N = 16$, the first level recursion distributes a 4×2 input matrix to each core while each stores into a 16×4 output matrix. When $N = 256$, each of the four cores works on a 16×8 matrix to produce its individual output that fits into a 256×64 output matrix. Therefore, the inter-process cache line requirement between recursion levels can be

generalized as ($N^2/p\mu$), where μ = number of elements in a cache line and $N \geq \mu$. An inclusive shared L2 cache of size 4MB can service these communication requests until $N \leq 2^9$. Coarse-grained parallelism for increasing input sizes as described here requires larger computation steps followed by relatively smaller communication and recursion control overheads. To ensure better cache reuse, a blocking factor is introduced to utilize blocks of array elements before moving on to retrieve the next cache line. In general, blocking factor is the number of elements in a cache line (μ) = the cache line size ÷ the size of an array element.

3.2. Parallel Smoothed L0 (SL0) Reconstruction Algorithm

The SL0 algorithm attempts to reconstruct a compressively sampled image by approximating the ℓ_0-norm with a continuous function. A continuous function with respect to x, $f_\sigma(x)$, can be described as follows:

$$f_\sigma(x) = e^{-\frac{x^2}{2\sigma^2}}, \qquad x \in \mathbb{R}, \sigma \in \mathbb{R}_+$$

This Gaussian function based on the parameter σ gives the number of zero elements in the vector x since:

$$\lim_{\sigma \to 0} f_\sigma(x) = \begin{cases} 1, & |x| \ll \sigma \\ 0, & |x| \gg \sigma \end{cases}$$

Taking the summation of all the cost functions for each vector element yields the number of zero elements in vector x, $f'(x)$. Hence, the problem in effect is:

$$\text{minimize } \|x\|_0 : N - f'(x) \quad \text{subject to } y = Ax,$$

where A is an $m \times N$ sampling WHT matrix. A decreasing sequence of carefully selected σ are needed to compute $\|x\|_0$. This process is called a Graduated Non-Convexity procedure, wherein the σ_{max} is gradually reduced to zero while the maxima of f_σ is then

used to locate the maximum f_σ for the next smaller σ. This is essentially a steepest descent approach. This process is shown in the algorithm below.

1. $\sigma_{factor} = 0.7$, $\sigma_{min} = 0.01$, $L_{factor} = 1$, $L = 3$, $k = 1$
2. $x \leftarrow A^{+}y$
3. $\mu \leftarrow [0.001, 0.001, 0.001, 0.05, 0.06, 1.4, \ldots, 1.4]$
4. Search σ_{max} in the vector x
5. $\sigma \leftarrow \max |x| / (2.75 \times \delta)$
6. while $\sigma > \sigma_{min}$ do
    a. $i \rightarrow 0$ to L
        i. $z \leftarrow x \,.\, \exp(-(x^2/2\sigma^2))$
        ii. $x \leftarrow x - \mu_k \,.\, z$
        iii. $x \leftarrow x - A^{+}(Ax - y)$
    b. $\sigma \leftarrow \sigma \,.\, \sigma_{factor}$; $L \leftarrow L \,.\, L_{factor}$; $k \leftarrow k + 1$

The parameters – iteration number (L), step-size (μ) as well as the starting point for σ are chosen from empirical analysis to result in optimal phase transition for varying values of indeterminacy ($\delta = m/N$). In the algorithm above, A^{+} is an $N \times m$ Moore-Penrose pseudo-inverse matrix, which is used to compute the inverse for non-square matrices. As it can be observed right away, the compute-intensive steps of this algorithm are the inverse computation and the matrix-vector products in the gradient descent step. The Moore-Penrose pseudo-inverse of A, denoted as A^{+}, is an extension of the inverse matrix. In this case, since A is an $m \times N$ matrix where $m < N$, the equation for the right inverse of A is:

$$A^{+} = A^{*}(AA^{*})^{-1} \quad \text{where } AA^{+} = \mathrm{I}_m$$

The Walsh-Hadamard transform case, which has linearly independent, orthonormal rows with all elements in the real domain, when applied to the equation above results in:

$$A^{+} = \frac{A^{\mathrm{T}}}{N}$$

Hence, the WHT pseudo-inverse is simply, the transpose of the sampling matrix *A*. This substitution when plugged back into the algorithm hugely simplifies the recurring gradient descent step. Keeping in mind these observations and modifications to the SL0 algorithm, a suitable scheme is devised to parallelize computationally demanding parts of the algorithm.

Step 2 of the algorithm merges the parallel computation of the pseudo-inverse with the initial vector calculation of *x*, as shown below in FIGURE 5.

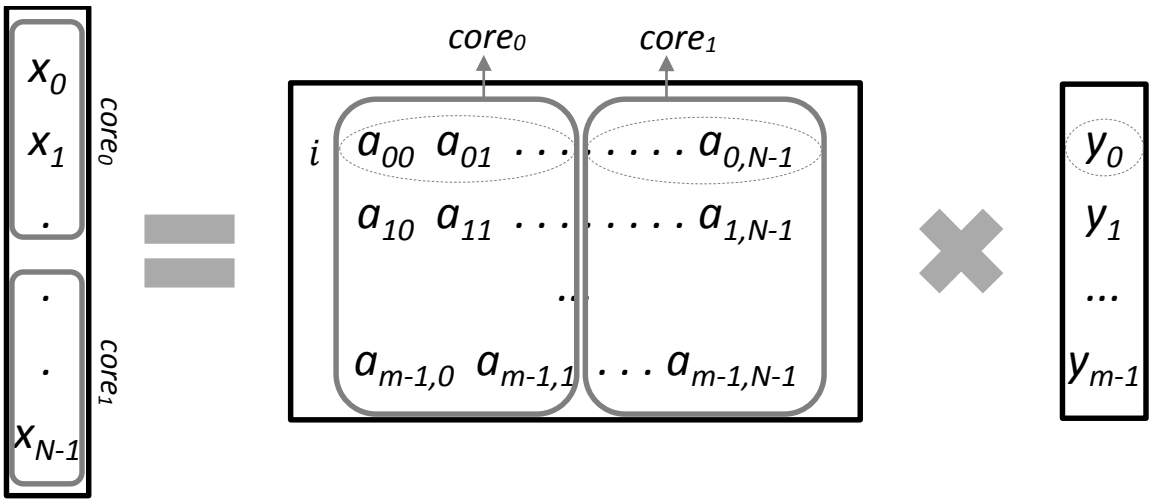

FIGURE 5: Illustration of $x = A^{+}y$ for two threads

The pseudo-inverse doesn't need to be directly computed and is implemented implicitly by accessing matrix *A* appropriately. The pseudo-inverse matrix A^{+} is needed to

be thread-wise partitioned horizontally, with each core accessing N/p rows; meaning that the sampling matrix A is simultaneously partitioned vertically into N/p segments and accessed row-major wise as seen in FIGURE 5. This can be further optimized by accessing each of the partitioned matrices block-wise in the size of the number of elements in a cache line. The search algorithm equally divides the vector x to each of the cores and each runs a linear search on its piece of the vector; a maximum is identified from the search results of each core. An involved search like mergesort may accrue unnecessary control overheads – although an image-based heuristic search could provide significant improvement in performance of the search algorithm.

In the gradient descent iteration, optimizations are used to reduce matrix-vector multiplications by equally partitioning the intermediate as well as the reconstructed vector x into N/p elements and the matrix A vertically into N/p segments, before supplying the sampled vector y to proceed with multiplying this temporary result with the pseudo-inverse. This entire process is depicted in FIGURE 6 for a dual core, two thread program. Barrier synchronization is required in the algorithm's iterations after step a.ii, during the $(Ax - y)$ computation and after step a.iii.

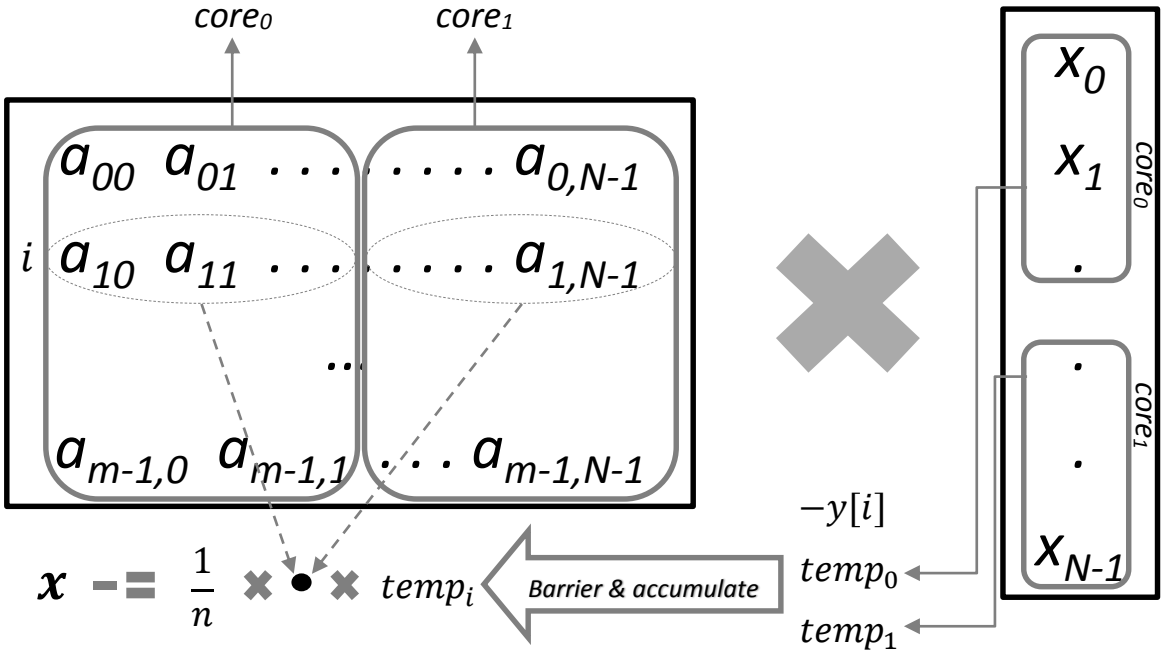

FIGURE 6: Illustration of $x = x - A^{+}(Ax - y)$ for two threads

Indeterminacy ($\delta = m/N$) of the CS image decides the speed as well as the accuracy of the reconstruction. As indicated above, each core receives the workload of N/p elements of the vector x and the computation of a single vector element of x requires N row elements of matrix A and m row elements of A^{+} (or m column elements of A). In other words, each core performs $2N/p$ multiplications, iterating m times to compute x. Hence, the cache line requirement per iteration is $N/p\mu + 2N/p\mu = 3N/p\mu$ before and after the barrier each, since μ halves for the single-precision float x. Apart from the $2N/p$ operations per core, there are synchronization overheads incurred to update the intermediate variable $temp_i$, which is expected to increase with the number of threads as $O(p)$. In contrast, an alternative implementation is to generate $temp_i$ on each core individually, requiring $(N + N/p)$ multiplications without any barrier stage. Hence, making the assumption that each

operation executes in a single cycle and uniform miss rate patterns for either implementation method, the benefit to dividing the ($Ax - y$) workload against performing it on each core can be expressed as:

$$N > \frac{N}{\mathrm{p}} + \mathrm{O}(p)$$

Or,

$$N > \frac{p}{p-1}\,\mathrm{O}(p)$$

This means that given the number of threads p, workload distribution of ($Ax - y$) gains for sufficiently larger N. Conversely, for large p, O(p) is large too, making workload distribution unsuitable for small sizes of N.

For a quad-core cache system, given the data cache requests needed per iteration, $N > 2^{12}$ will exceed a 32KB L1 private caches of each core and $N > 2^{17}$ would necessitate out-of-cache requests in a shared 4MB L2 cache size, assuming half the requests to this unified cache are data load/store operations. For acceptable reconstruction at $m = 0.4N$, updating the vector x requires a total of $5N^2/2p\mu$ cache lines in m fully in-cache iterations.

The convergence analysis of the SL0 algorithm is a bit involved and can be referred to in [20]. As explained in that paper, the SL0 algorithm presents an asymptotic computational complexity in the order of O(N^2). This implementation of the SL0 reduces it further. As seen above, the modified algorithm performs implicit transpositions and merges computation steps. Hence, the computational complexity for this algorithm is O(mN). This indicates a speedup of at least N/m over the generalized SL0 implementation. The parallel speedup over its single-threaded counterpart could be affected by the barrier synchronization overheads.

3.3. Implementation on the Intel Xeon

The platform used for the implementation of this compressed sensing algorithm is an Intel Xeon machine. This uses the dual LGA 771 socket supporting two Intel Xeon X5365 processors, codenamed Clovertown, each of which is a 65 nm Intel Core™-based quad-core 64-bit processor with a maximum CPU clock rate of 3 GHz. The FSB is quad-pumped with a 64-bit bus between the two L2 caches running at a frequency of 333 MHz, thus delivering a throughput of 10.66 GB/s. The Intel 5000 P (Blackford) chipset's memory controller hub (MCH) interfaces to four FB-DIMM 667 MHz DRAM channels that are capable of a read memory bandwidth of 21.66 GB/s.

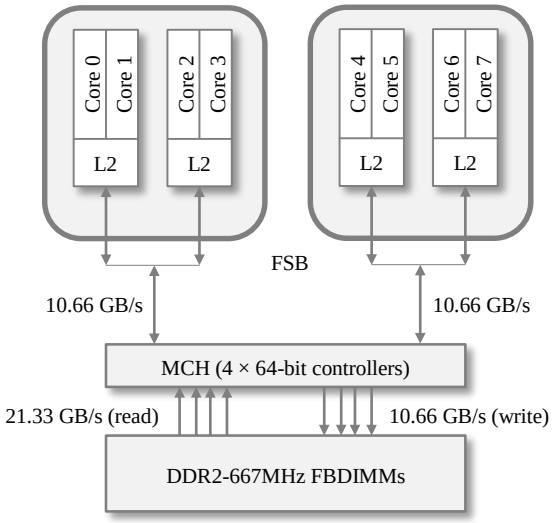

FIGURE 7: Architectural layout of dual-socket quad-core Xeon

Each of the four cores have private, independent data and instruction L1 caches. Each L1 cache is a 32KB cache with 8-way associativity. There are two, inclusive, unified,

16-way, 4MB L2 caches, each of which are shared between two cores. All caches have 64-byte cache lines.

TABLE 1: Xeon X5365 processor specifications

	Intel Core
L1 data cache	32 KB
L1 instructions cache	32 KB
L1 latency	3 clock cycles
L1 associativity	8-way
L1 TLB size	Instructions: 128 entries Data: 256 entries
Max. L2 cache	4 MB for two cores
L2 latency	14 clock cycles
L2 associativity	16-way
L2 cache bus width	256 bit
L2 TLB size	?
Pipeline	14 stages
x86 decoders	1 complex and 3 simple
Integer execution units	3 ALU + 2 AGU
Load/Store units	2 (1 Load + 1 Store)
FP execution units	FADD + FMUL + FLOAD + FSTORE
SSE execution units	3 (128-bit)

Apart from the processor features listed above, the Intel Core™ microarchitecture includes several enhancements, which are critical to optimizing and fine-tuning the multithreaded performance of the program:

- Each core has a peak six µops per cycle issue rate, four x86 decoders decoding up to five instructions/cycle and a retirement bandwidth up to four µops/cycle in a 14 stage pipeline.
- Superscalar out-of-order speculative execution engine that uses memory disambiguation to reduce cache-miss exposure
- Hardware prefetchers that prefetch data in response to data access patterns to reduce effective cache-miss latency. The hardware data prefetcher consists of a data cache unit (DCU) prefetcher for the L1 data cache and data prefetch logic (DPL) for the L2 cache.
- Advanced branch prediction and stack pointer tracker for efficient procedure entry/exit.

To effectively take advantage of the Core microarchitecture, it is necessary to evaluate the code for performance gains measured in terms of the wall-clock time and/or the CPU time. Identifying the processes that are CPU-bound or memory-bound are important to recognize the bottlenecks and optimize the code to improve performance. Apart from elapsed time, compilers also optimize performance of the target hardware and have common features to fine-tune code like profile-guided optimization (PGO), multithreaded support, cache-management features and so on. These, along with code-specific optimizations used in the program that target and harness the capabilities of the architecture are:

- Eliminating branches through code rearrangement and loop unrolling.
- Reorganize data by blocking, loop interchange and loop tiling to reduce cache misses through temporal and spatial locality.

- Aligned and consistent data access patterns that are in sequential cache lines as well as strided locations which can ensure sustained µop throughput and can expect reduced latency from employing hardware and software (PREFETCH) prefetching mechanisms.
- Avoiding store forwarding stalls by proper alignment and padding. Using write combining buffers to allow threads on multiple cores to write into a single cache line.
- Barrier synchronization (fence) using the PAUSE instruction to indicate a spin-wait loop that improves performance by significantly reducing the chance of a memory order violation.
- Utilization of the Loop Stream Detector (LSD) to optimize small loops containing branches by replaying instructions from instruction queue back into the decoder.
- Minimizing bus latency by segmenting code into read phases and write phases of bus transactions.

The Xeon system has 16 GB DDR2 RAM with 120 GB HDD space. It uses an installation of the Red Hat Enterprise Linux distribution, which runs a lightweight kernel (version 2.6.18) along with the Intel VTune™ Performance Analyzer suite. The Eclipse Kepler Software Development Kit (SDK) for Linux is used to write the stand-alone program in C++. The program is compiled using the GCC C++ Compiler with ANSI C compatibility. The program uses the portable POSIX thread (pthread) library to implement threads and synchronization. The real-time extensions library is needed to enable features like memory mapped files and high-res timers. Vectors or dynamically allocated arrays are used to handle the input images and transform matrices.

The basic structure of the program is listed here along with its salient functions.

1. Main
    - Interpret command line parameters - input image, sampling size, no. of measurements, cores and threads.
    - Go to thread initialization and execution
    - Reconstructed signal is encoded as a PNG image
2. Initialize Threads
    - Setup barrier synchronization functions
    - Create threads and execute respective functions
3. Thread Function
    - Set each thread's CPU affinity.
    - Decode input Image from PNG
        - Resize arrays based on image size
        - Reinterpret image (or discretization of image)
    - Generate Walsh Hadamard Sampling Matrix
    - Use above matrix and input image to generate the compressively sampled signal, which is to be reconstructed in the next step.
    - Smoothed L0 Reconstruction Algorithm
    - Mean Squared Error between input and reconstructed images.

CHAPTER 4. RESULTS AND DISCUSSION

Performance of a workload is often measured using the total execution time represented in terms of instructions per cycle (IPC) or clocks per instruction (CPI) as:

$$t = n \times \text{CPI} \times 1/f$$

Here, *n* is the number of instructions and *f* is the processor's clock frequency. While CPI is adequate to describe the performance of a single-threaded workload, it is an insufficient metric for examining multi-threaded workloads. The instruction throughput varies across runs since thread execution paths and thread interleaving are not always the same. Performance evaluation on a parallel architecture can be interpreted from the actual runtime of user code and the shared resource utilization. The tools used to measure the program's performance in this research are:

- Cachegrind Profiling Analysis
- VTune Event Ratios
- Response Time Regression Model

4.1. Cachegrind Profiling Analysis

The data access patterns of multi-threaded programs determine effective usage of the CPU cache hierarchy. A useful measure of interactions between the program code, the private L1, shared L2 cache levels and the main memory is miss rates. The cache miss rates can be broken down into requests missed for loads and stores as well as data and instruction misses for the independent L1 D-cache and I-cache units. Cache parameters (cache size,

associativity, write policy) and problem size dictate the miss rates of the algorithm. This work presents two methods to evaluate miss rates; program simulation for varying cache parameters on a cache profiler, Cachegrind, and, hardware event based-sampling of the program using Intel VTune Performance Analyzer.

Cachegrind [24] is part of a larger framework called Valgrind, a tool suite that uses dynamic binary instrumentation for debugging and profiling programs. Cachegrind simulates a machine with independent first-level instruction and data caches backed by a unified second-level cache, matching the architectural requirements of the analytical model of this work. Moreover, it implements write-allocation but ignores the write policy, and since the Intel Core platform uses write-back consistently at all levels, there is no requirement to account for any extra write requests. Also, it auto-configures the unified 4MB L2 cache of the Xeon platform. This makes it viable to use Cachegrind in order to model a cache hierarchy that gathers instruction and data cache reads and writes for a single-threaded execution of the workload.

Cachegrind is configured to collect the profiling information for representative problem sizes and L1 data cache parameters. L1 miss rates are shown FIGURE 8 (a) – (f) on the next page for the matrix generation and the reconstruction algorithm. The miss rate variation is depicted for problem sizes N = 1024, 4096, 16384; L1 cache sizes (C) from 8KB to 64KB and associativity (α) from direct to fully associative. The cache line size for this simulation is 64-bytes, consistent with the underlying Intel Xeon.

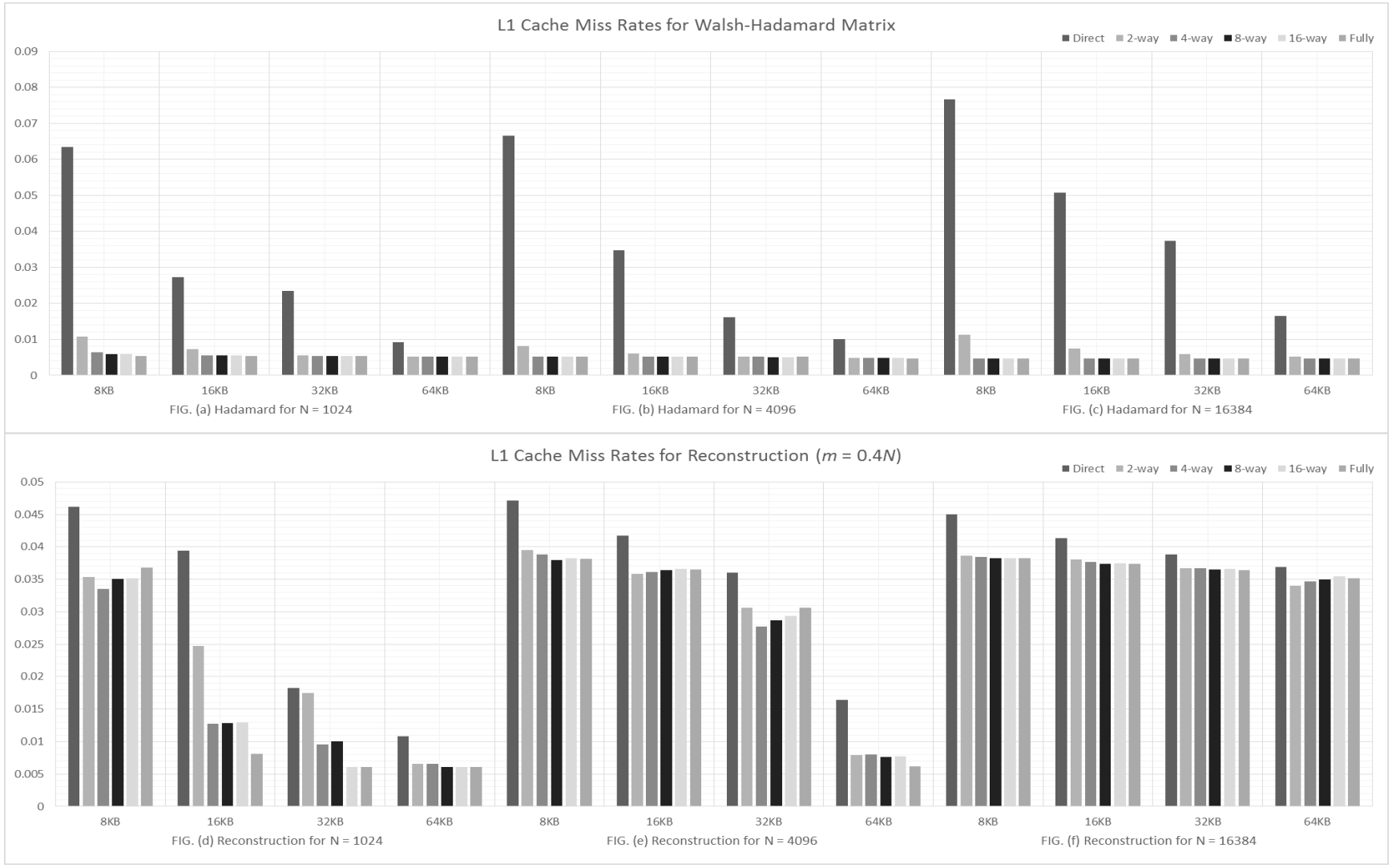

FIGURE 8: Data cache miss rate for different cache parameters and problem sizes

Low associativity and small cache sizes for L1 data caches tend to significantly impact performance of code and this can be directly inferred from the higher miss rates for direct-mapped and two-way associative caches in FIGURE 8. The miss rates are interpreted by observing the contributions of the individual load and store requests missed. Here, the total miss rate is the ratio of total misses to total accesses while the read miss rate would be the ratio of the read misses to the total read requests to that cache level.

TABLE 2: Total accesses and last-level misses

Size (*N*)	Instruction Count	Data Reads	Data Writes	L2 Read Misses	L2 Write Misses
WHT Algorithm					
1024	21,143,031	6,522,529	1,089,106	0	30,778
4096	331,661,039	101,897,889	17,064,146	840	541,771
16384	5,266,613,215	1,618,147,521	270,623,746	10,781	8,480,543
Reconstruction Algorithm					
1024	422,175,603	70,464,085	17,488,718	5	0
4096	6,672,236,955	1,093,083,644	271,345,259	8,419,824	32
16384	106,405,567,784	17,347,995,349	4,306,738,435	134,360,760	692

In the WHT program, as indicated in TABLE 2 above, the L1 data reads account for about 85% of total accesses on average, which translates to 15 write requests for every 100 accesses. Meanwhile, the read misses account for a larger share (> 50%) of the total misses for $\alpha \leq 2$ as seen in FIGURE 8 (a) – (c), which is because each iteration has three aligned accesses, resulting in large conflict misses. It follows that for $\alpha > 2$, the L1 read miss rate reduces dramatically (~ 70%), which is less than a single read miss for 1000 read requests. Simultaneously, for any set associative cache, 1 write miss occurs for about 30 write requests, which also miss in the L2 cache, indicating a compulsory miss occurs for every 32 elements accessed (32 two byte *short-int* elements occupy a single 64-byte cache

line). This works out to a total L1 miss rate of less than 0.005 for $\alpha > 2$, which is about a miss (load/store) for 200 accesses. The latency to service these misses can be hidden by non-blocking caches, out-of-order processors and write buffers – all of which are present in the Xeon platform. In effect, the execution of this program is close to being processor bound since larger problem sizes show reducing miss rates.

In contrast, the recovery algorithm exhibits quite different miss rate patterns as seen in FIGURE 8 (d) – (f). As indicated in TABLE 2, the L1 data reads account for 80% of total accesses and the read misses constitute 98% of all misses. This implies that the L1 write miss rate is negligible with most of the accesses (of the output data) occurring in-cache. In FIGURE 8 (d) and (e), the miss rates can be seen to drop suddenly for $N = 1024$ when $C \geq 16$KB and again for $N = 4096$, when $C \geq 64$KB. This behavior is explained by the L1 read miss rate dropping by as much as 80% (from about 5 read misses in 100 read requests to about 1 in 100) when an iteration i fits in the cache of size C. For instance, when $N = 1024$, a single iteration requires 160 (= 5×1024/32) cache lines and when $N = 4096$, 640 cache lines, which fit within the L1 data cache of size $C = 16$KB and $C = 64$KB respectively. When i fits within C, every L1 read miss almost results in a L2 miss, which indicates that these misses are compulsory. When i doesn't fit within C, about every 5^{th} L1 miss results in a L2 (capacity/conflict) miss and a main memory access.

It can be inferred from TABLE 2 that, while the L2 misses for the WHT algorithm are compulsory write misses growing in the size N^2/μ, the L2 misses for the recovery algorithm are read misses growing as the term $\sigma N^2/\mu$, where $\sigma = 16$ for the Lena image used in this program.

4.2. VTune Event Ratios

The Intel's VTune package is used to execute the multithreaded WHT implementation on the Intel Xeon platform and sample relevant hardware events to analyze its performance. The L1 and L2 miss rates with respect to the matrix sizes generated are depicted in FIGURE 9 and are measured using the following events from VTune:

$$\text{L1 Data Miss Rate} = \frac{\text{L1D\_REPL}}{\text{INST\_RETIRED.ANY}}$$

$$\text{L2 Miss Rate} = \frac{\text{L2\_LINES\_IN.DEMAND.ANY}}{\text{INST\_RETIRED.ANY}}$$

The event L1D_REPL counts the number of lines brought into L1 data cache, L2_LINES_IN.DEMAND.ANY counts cache lines allocated in L2 due to requests by the same L1 data and instruction cache.

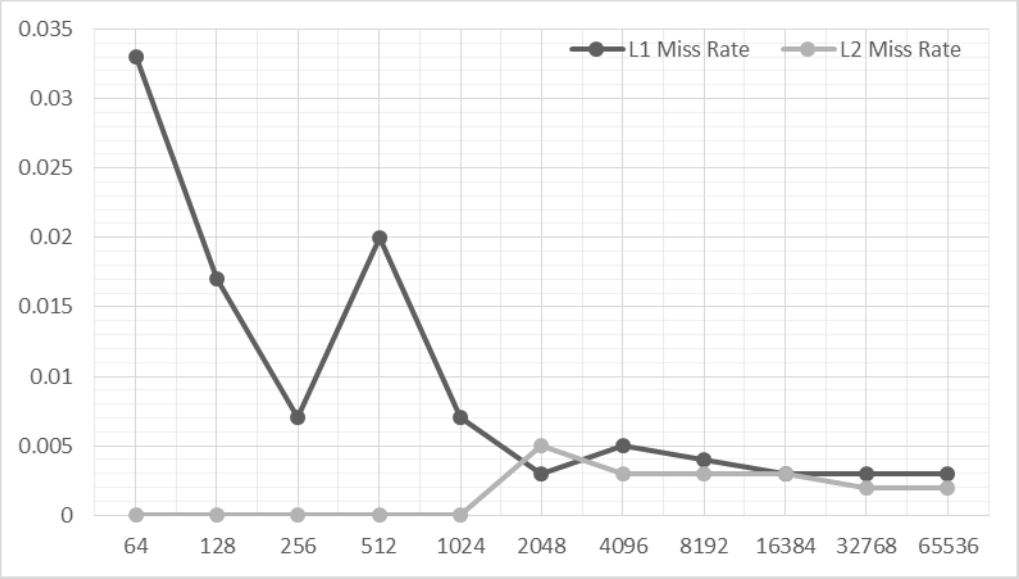

FIGURE 9: Miss rates for generating WHT on 8-core Xeon

Since L1 instruction cache and L2 unified cache are supplanted by prefetchers, the instruction miss rates are negligible. As discussed before in section 4.1, increasing problem size leads to lower L1 miss rate explained due to reducing write miss rates compensating for control overheads. So, as indicated in the FIGURE 9 the L2 miss rate is almost zero for sizes of $N \leq 1024$ fitting wholly in the 4MB cache. For $N > 1024$, the number of L2 misses serviced grows with the size of problem but the total number of instructions retired brings the L2 miss ratio lower. There are some contradictions, though, arising due to the false sharing of data. This can be interpreted using the data sharing ratio shown in and measured by:

$$\text{Modified Data Sharing Ratio} = \frac{\text{EXT\_SNOOP. ALL\_AGENTS. HITM}}{\text{INST\_RETIRED. ANY}}$$

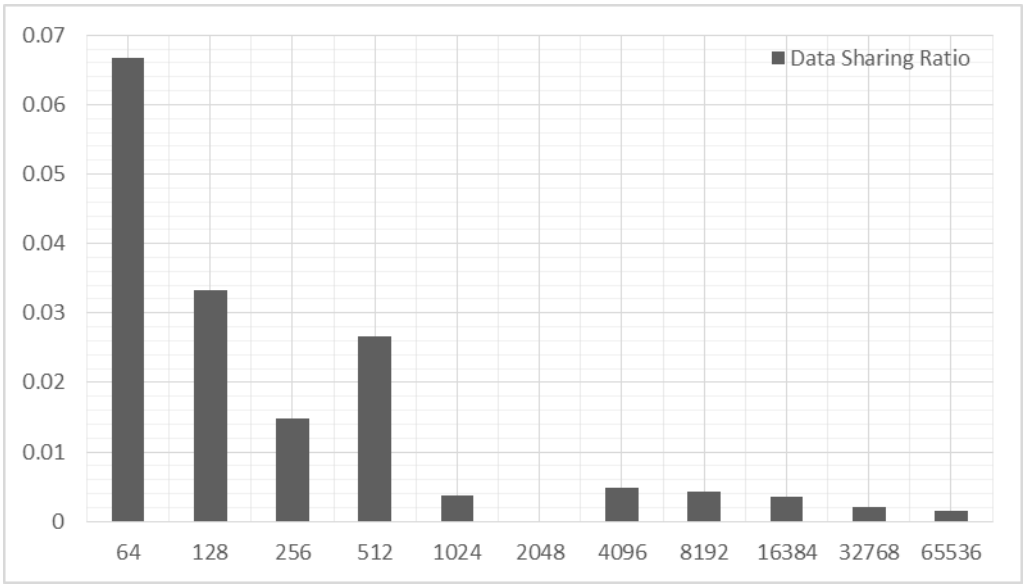

FIGURE 10: Data sharing ratio between threads

The EXT_SNOOP.ALL_AGENTS.HITM counts snooped responses for addresses in a modified state. The sharing ratio should ideally be zero for good multithreaded performance.

False sharing happens when one thread writes to a cache line and another thread attempts to read from or write to the same cache line also known as cache ping-pong. Smaller data sizes have false sharing due to threads reading and writing into a small number of cache lines; e.g. $N = 64$ has 2 cache lines per row shared between 8 threads. Meanwhile, $N = 512$ and 4096 shows higher than expected data sharing due to traversing more recursion levels and picking sub-trees that have false sharing at each step of the way.

The CPI ratio can be used to calculate the execution time using the processor frequency (3GHz) and the total instructions retired for each matrix generated of size N as shown in TABLE 3. The CPI takes a hit when VTune samples a large number of events, so execution time needs to be measured with minimal overheads.

TABLE 3: Execution times calculated using CPI

Size	65536	32768	16384	8192	4096	2048	1024	512	256	128	64
CPI	1.162	1.172	1.273	1.296	1.325	1.345	1.556	2	2.25	2.5	3
Time(s)	31.927	8.100	2.224	0.574	0.155	0.039	0.014	0.010	0.020	0.005	0.006

4.3. Response Time Regression Model

The elapsed time for the algorithms can be measured using a wall clock timer like the Linux real-time clock. The tests are run for the WHT algorithm as well as the barrier and barrier-less implementations of the SL0 reconstruction algorithm. The representative problem sizes of N range from 64 (2^6) to 65536 (2^{16}) for indeterminacy $\delta = 0.1$ to 0.8. The

threaded implementation of the workload is tested for 1, 4, 8 and 16 threads. The threads are core bound and threads sharing data are assigned to adjacent cores. When p = 16, two threads are bound to the same core. The speedup is calculated for a particular thread configuration using the ratio of its elapsed time with respect to the corresponding elapsed time for the single-threaded workload. The speedups are shown in FIGURE 11 on the following page.

A regression model is constructed from the response time for each of the algorithms on single or multiple cores using a hybrid mechanistic-empirical approach, by leveraging the understanding of the interaction between the algorithm and the underlying multicore system.

For the compute-bound WHT algorithm, the response time T_p for p threads is influenced by the input size N and can be described by the following non-linear model:

$$T_p = \mathrm{O}(N^2) + \mathrm{Overhead} \; = \; a_p N^2 + b_p$$

Here, a_p and b_p are constants for a given thread configuration where a_p denotes the instruction cycles spent to perform useful work, i.e., generate the WHT matrix and b_p denotes the thread setup and control overheads. Hence, expanding these terms deliver the overall regression model for the WHT algorithm as presented below:

$$T_p = N^2 \left(\frac{t_{parallel}}{p} + t_{seq} \right) + \beta \cdot p$$

where, $\quad t_{parallel} = 3.628 \times 10^{-9}$

$$t_{seq} = 2.504 \times 10^{-10}$$

$$\beta = 0.00001$$

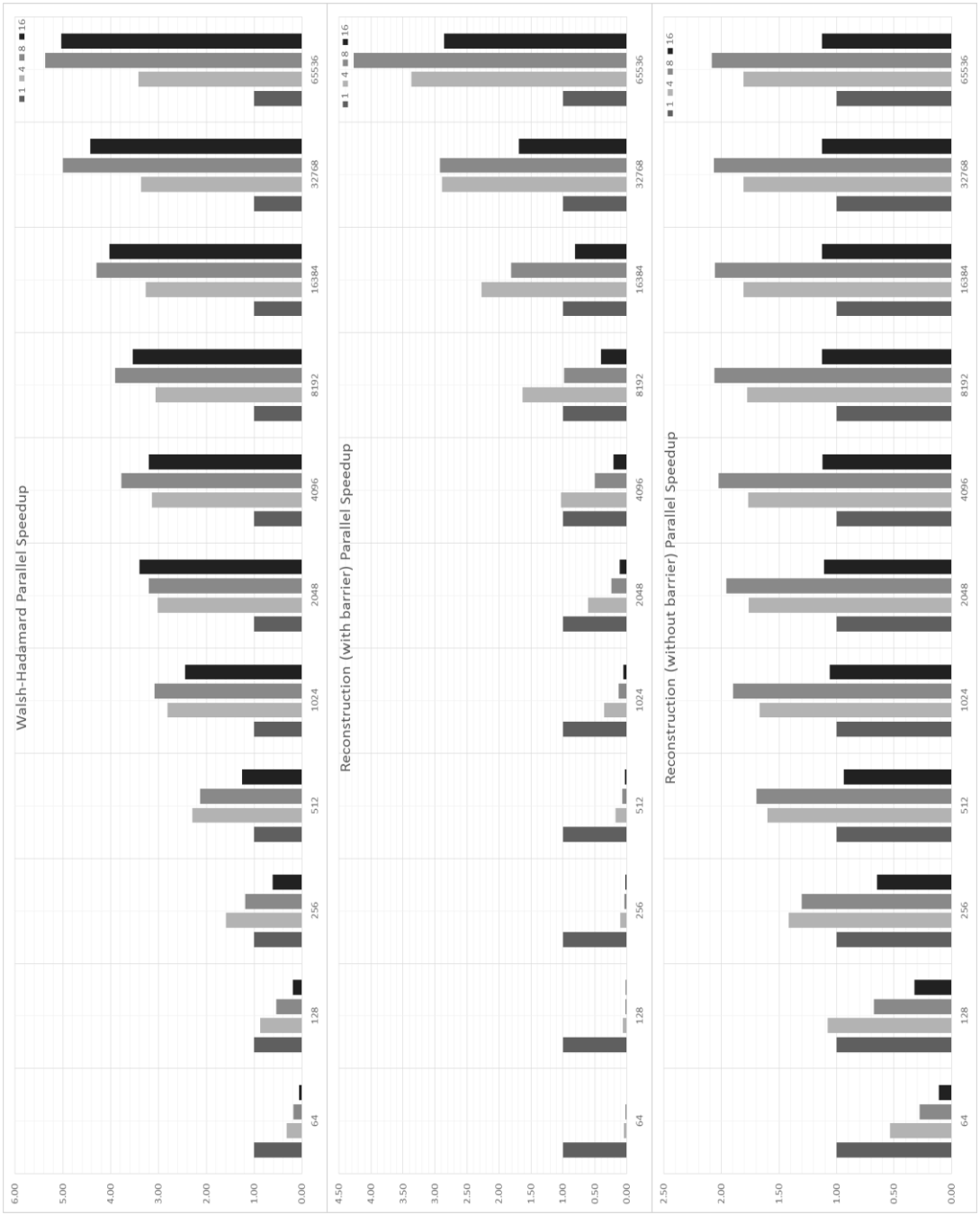

FIGURE 11: Speedup of the multithreaded workload for varying problem size N

This regression model exhibits good fit to the data with a coefficient of determination, R^2 = 99.8%. Here, $t_{parallel}$ and t_{seq} are the respective parallel and sequential code runtime factors within the algorithm, which indicate the benefit in increasing threads for a given working set of the problem. 93.5% of the application code is parallelizable, as shown by the equation $t_{parallel} * 100/(t_{parallel} + t_{seq})$. For N < 256, while the t_{seq} component is relatively small, β makes up a significant portion of the overheads that threaded workloads have to deal with. Hence, the four thread configuration gains for $N \geq$ 128 and the eight thread configuration gains for $N \geq 256$ over the single threaded counterpart. For p = 16, resource allocation stalls double the t_{seq} component and context switching delays increase the overhead β, ensuring that this sees improvement only at $N \geq$ 512. The model also shows the threaded workload reaching its performance limit at 3.4 for p = 4 and at 5.5 for p = 8. It should be noted here that due to optimized memory access patterns, the memory stalls that could occur from compulsory L2 write misses are completely hidden by the pipeline and write buffers.

In the case of the recovery algorithm, apart from the measurement vector size m and image size N, the average memory access time (t_{mem}) plays a significant role in determining the response time. As explained before, L1 misses are serviced completely within the L2 for N < 1024, whereas $N \geq 1024$ requires drawing data from the main memory. From the analytical model:

$$T_p = \mathrm{O}\left(\frac{mN}{p}\right) + t_{mem} + \mathrm{Overhead}$$

While the L1 misses are in the order O($N/p\mu$), the L2 misses serviced are in O($N^2/p\mu$). In the equation above, overhead accounts for coherency latencies, thread

initialization delays and other measurement errors. Hence, the recovery algorithm can be depicted in general as follows:

$$T_p = \left(N\left(\frac{t_{parallel}}{p} + t_{seq}\right) + t_{barrier} \cdot p\right)m + t_{mem} + \beta \cdot p$$

$$where, \qquad t_{mem} = \begin{cases} \dfrac{\gamma N}{p\mu}, & N < 1024 \\ \dfrac{\gamma N^2}{p\mu}, & N \geq 1024 \end{cases}$$

Here, γ is the miss penalty factor and μ is the number of elements contained in a cache line. While both the barrier and barrier-less implementations of the algorithm are modeled by the equation above, there are some significant variations that distinguish their performance for different problem sizes and threads. The values for the input variables obtained from the regression analysis are depicted in TABLE 4.

TABLE 4: Regression values for SL0 algorithms

Factors	**Barrier**		**Barrier-less**	
$t_{parallel}$	2.151×10^{-7}		1.245×10^{-7}	
t_{seq}	≈ 0		8.498×10^{-8}	
$t_{barrier}$	2×10^{-4}		≈ 0	
t_{mem}	$N < 1024$	$N \geq 1024$	$N < 1024$	$N \geq 1024$
γ	1.55×10^{-4}	8.921×10^{-8}	1.494×10^{-4}	1.899×10^{-7}
β	0.0005	0.025	0.00025	0.017
$\mathbf{R^2}$	96.032%		96.711%	

The barrier implementation sets $t_{seq} \approx 0$ indicating a high degree of parallelizability (above 99%) but the barrier wait overhead denoted by $t_{barrier}$ reduces this benefit, with the term '$t_{barrier} \cdot p \cdot m$' significantly impacting performance for small *N*. On the other hand, the barrier-less implementation sets $t_{barrier} \approx 0$ as there are low CPU wait stalls, but performs

more computations per core to avoid communication overheads. These computations are represented by the variable, t_{seq}, with only 59.43% of the code being parallelizable, resulting in a speedup bottleneck of 1.8 for 4 threads and 2.0 for 8 threads at $N \geq 2048$.

For $N < 1024$, the miss patterns given by the value of γ in TABLE 4 are quite similar for a single or multi-threaded workload using either implementation. This changes significantly for $N \geq 1024$, as the miss penalty factor for the barrier-less implementation is about two times that of the barrier implementation. This makes the barrier implementation more efficient for $N \geq 16384$ for $p = 4$, compensating for the wait states with lower miss penalty.

CHAPTER 5. CONCLUSIONS AND FUTURE WORK

The emerging field of compressed sensing has a lot of scope to benefit from efficient real-time realizations on modern computing systems and this research looks at implementations that exploit shared memory designs. Generation of the Walsh-Hadamard matrix using a high performance parallel algorithm shows excellent speedup of up to 5.6 over using single-core processors, while the CS recovery algorithm implemented for multicore systems achieves a speedup boost of up to 4.5. This research also develops performance models for the respective algorithms, correlating computational complexity and miss rate patterns with the response time over single and multiple threads.

The insights presented in this work are valuable to drive multicore-aware compressed sensing research forward. One such immediate goal would be to develop power-optimized parallel algorithms for low power embedded devices. Analytical modeling can be extended to include micro-architectural characteristics tailored for CS imaging. Apart from WHT, other transforms like Haar wavelets can benefit from multicore implementations too. While this work efficiently parallelizes a fast recovery algorithm, it does not evaluate the quality of the reproduced image. Picking a suitable algorithm for an application by weighing in the accuracy requirements and the computation complexity present a broader direction for future research.

REFERENCES

[1] M. Rudelson and R. Vershynin, "On sparse reconstruction from Fourier and Gaussian measurements," *Communications on Pure and Applied Mathematics,* vol. 61, no. 8, pp. 1025-1045, 2008.

[2] M. F. Duarte, M. A. Davenport, D. Takhar, J. N. Laska, T. Sun, K. F. Kelly and R. G. Baraniuk, "Single-Pixel Imaging via Compressive Sampling," *IEEE Signal Processing Magazine,* vol. 25, no. 2, 2008.

[3] E. Candès, T. Tao and J. Romberg, "Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information," *IEEE Transactions on Information Theory,* vol. 52, no. 2, pp. 489-509, 2006.

[4] D. Donoho, "Compressed Sensing," *IEEE Transactions on Information Theory,* vol. 52, no. 4, pp. 1289-1306, 2006.

[5] R. Chartrand, "Nonconvex compressed sensing and error correction," *IEEE International Conference on Acoustics, Speech, and Signal Processing,* vol. 3, pp. III-889, 2007.

[6] B. J. Borgstrom and A. Alwan, "Utilizing Compressibility in Reconstructing Spectrographic Data, with Applications to Noise Robust ASR," *IEEE Signal Processing Letters,* vol. 16, no. 5, pp. 398-401, 2009.

[7] G. Huang, H. Jiang, K. Matthews and P. Wilford, "Lensless Imaging by Compressive Sensing," *IEEE International Conference on Image Processing,* pp. 2101 - 2105, 2013.

[8] D. Takhar, J. N. Laska, M. B. Wakin, M. F. Duarte, D. Baron, S. Sarvotham, K. F. Kelly and R. G. Baraniuk, "A New Compressive Imaging Camera Architecture using Optical-Domain Compression," *Proceedings of Computation Imaging IV at SPIE Electronic Imaging,* pp. 606509-606509, 2006.

[9] E. Candès and T. Tao, "Near optimal signal recovery from random projections: Universal encoding strategies?," *IEEE Transactions on Information Theory,* vol. 52, no. 12, pp. 5406-5425, 2006.

[10] P. Wojtaszczyk, "Stability and Instance Optimality for Gaussian Measurements in Compressed Sensing," *Foundations of Computational Mathematics,* vol. 10, no. 1, pp. 1-13, 2010.

[11] S. G. Johnson and M. Frigo, "The Design and Implementation of FFTW3," *Proceedings of the IEEE,* vol. 93, no. 2, pp. 216-231, 2005.

[12] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa and J. M. F. Moura, "Discrete Fourier Transform on Multicore," *IEEE Signal Processing Magazine,* vol. 26, no. 6, pp. 90-102, 2009.

[13] K. Chen and J. Johnson, "A self-adapting distributed memory package for fast signal transforms," *IEEE Proceedings of 18th International Parallel and Distributed Processing Symposium,* p. 44, 2004.

[14] E. Candès, "The restricted isometry property and its implications for compressed sensing," *Comptes Rendus Mathematique,* vol. 346, no. 9, pp. 589-592, 2008.

[15] F. Malgouyres and T. Zeng, "A predual proximal point algorithm solving a non negative basis pursuit denoising model," *International Journal of Computer Vision,* vol. 83, no. 3, pp. 294-311, 2009.

[16] J. A. Tropp and A. C. Gilbert, "Signal Recovery From Random Measurements Via Orthogonal Matching Pursuit," *IEEE Transactions on Information Theory,* vol. 53, no. 12, pp. 4655-4666, 2007.

[17] J. Bioucas-Dias and M. Figueiredo, "A new TwIST: two-step iterative shrinkage/thresholding algorithms for image restoration," *IEEE Transactions on Image Processing,* vol. 16, no. 12, pp. 2992-3004, 2007.

[18] D. Donoho and J. Tanner, "Observed universality of phase transitions in high-dimensional geometry, with implications for modern data analysis and signal processing," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences ,* vol. 367, no. 1906, pp. 4273-4293, 2009.

[19] H. Mohimani, M. Babaie-Zadeh, I. Gorodnitsky and C. Jutten, "Sparse Recovery using Smoothed ℓ0 (SL0): Convergence Analysis," *arXiv preprint arXiv:1001.5073,* 2010.

[20] H. Mohimani, M. Babaie-Zadeh and C. Jutten, "A fast approach for overcomplete sparse decomposition based on smoothed norm," *IEEE Transactions on Signal Processing,* vol. 57, no. 1, pp. 289-301, 2010.

[21] C. S. Oxvig, P. S. Pedersen, T. Arildsen and T. Larsen, "Improving Smoothed ℓ0 Norm in Compressive Sensing Using Adaptive Parameter Selection," *arXiv preprint arXiv:1210.4277,* 2012.

[22] A. Borghi, J. Darbon, S. Peyronnet, T. F. Chan and S. Osher, "A Simple Compressive Sensing Algorithm for Parallel Many-Core Architectures," *Journal of Signal Processing Systems,* vol. 71, no. 1, pp. 1-20, 2012.

[23] S. Lee and S. J. Wright, "Implementing Algorithms for Signal and Image Reconstruction on Graphical Processing Units," *Optimization Online,* vol. 10, 2008.

[24] N. Nethercote, R. Walsh and J. Fitzhardinge, "Building workload characterization tools with Valgrind," *IEEE International Symposium on Workload Characterization,* pp. 2-2, 2006.