SAISort: An Energy Efficient Sorting Algorithm for Many-Core Systems

By

LUCAS STILLMAKER
B.S. (California State University, Fresno) May, 2008

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Chair, Dr. Bevan M. Baas

Member, Dr. Soheil Ghiasi

Member, Dr. Venkatesh Akella

Committee in charge
2011

© Copyright by Lucas Stillmaker 2011
All Rights Reserved

Abstract

Energy efficiency is an important aspect of nearly all computing systems for applications ranging from large data centers to mobile devices. Many applications, especially those used in data centers, make heavy use of sorting algorithms. This thesis proposes the implementation of a novel sorting algorithm on a many-core chip to be used as a co-processor in conjunction with a general purpose processor. This algorithm takes advantage of the parallelism offered by a many core system to sort lists for database applications. The algorithm is implemented on the Asynchronous Array of Simple Processors second version (AsAP2) as a proof of concept.

The results show that large gains in energy efficiency can be obtained for the first phase of a database sort. The implementation on the AsAP2 chip is able to sort a 10 GB list of entries into 783-entry lists using 23% of the energy a mobile i7 processor consumes during the execution of a quicksort while leaving the keys and payload attached. When sorting the keys and payloads separately, the AsAP2 implementation uses 21% of the energy compared to the i7 implementation while sorting into lists of 3,911 entries. Lowering the voltage of the AsAP2 processor allows the processor to function even more efficiently, with the AsAP2 implementation using 8% and 7% of the energy compared with their i7 counterparts for payload and key attached and separated cases respectively.

Acknowledgments

I want to thank all of those who made this project possible. I would like to thank my advisor Professor Bevan Baas as without his guidance and patience through my years at UC Davis, this would not have come together. I would like to thank Professor Ghiasi for starting me down the path of thinking that lead to this project. I would also like to thank Professor Akella for his valuable support in reviewing my thesis.

I would like to thank my brother and lab-mate Aaron Stillmaker for all of his help. If it weren't for the countless hours bouncing ideas off of Aaron, this project never would have gotten off the ground. I would also like to thank my friend Ameen Akel from the Non-Volatile Systems Laboratory at UC San Diego for offering ideas to get me unstuck when I was stumped while writing C++ code.

I am very appreciative of all of the work that was done that made this project possible. Thank you everyone at the VLSI Computation Lab at UC Davis who helped create the amazing AsAP2 processor, and everyone there who gave me their support during my time at Davis.

A big thank you to my my parents who provided so much support, both financial and otherwise through the years. Thank you also to the rest of my friends and family who have helped me get to this point.

Contents

Abstract **ii**

Acknowledgments **iii**

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**

1.1 Enterprise Database Sorting 1
1.2 Energy Efficient Database Sorting 2
1.3 Project Contributions 3
1.4 Organization 3

2 Overview of Enterprise Database Sorting **5**

2.1 Internal Sort 5
2.1.1 Background 5
2.1.2 Commonly Used Internal Sorts 6
2.2 External Sorts 8
2.2.1 Background 8
2.2.2 Commonly Used External Sorts 9
2.3 Related Work 11
2.3.1 Parallel Database Sorting 11
2.4 Proposed Enterprise Database Sort 12

3 Attempted Implementations **14**

3.1 Phase One Sorts 14
3.1.1 Sequential Internal Sorts 14
3.1.2 Merge Sorts 15
3.2 Phase Two Buffer Sort 19
3.2.1 Implementation at Processor Level 20
3.2.2 Chip Mapping 22
3.2.3 Benefits / Limitations of AsAP2 Implementation 25

4 Proposed Sort **26**

4.1 Phases 26
4.1.1 Phase One / SAISort 27

4.1.2 Phase Two / Buffer Sort . . . 32
4.1.3 Phase Two / Co-Processor Binary Merge . . . 32
4.2 AsAP2 Chip Implementation . . . 33
4.2.1 AsAP2 Background . . . 33
4.2.2 SAISort Implementation on AsAP2 . . . 35
4.3 C++ Implementation . . . 38
4.3.1 Phase One / Quicksort . . . 38
4.3.2 Phase Two / Binary Merge . . . 41

5 Results and Analysis 43
5.1 Calculations of Energy Consumption . . . 44
5.1.1 Calculation of Power Consumption . . . 44
5.2 C++ Performance Comparison to SAISort . . . 47
5.3 AsAP2 Implementation Performance Comparisons . . . 50
5.4 Hardware Variations . . . 52
5.4.1 Varying the Number of Processors . . . 53
5.4.2 Varying the Quantity of On-Chip Memory . . . 54
5.5 Feasibility of Implementation . . . 55

6 Future Work and Conclusion 56
6.1 Future Work . . . 56
6.1.1 Searches . . . 56
6.1.2 Data Calculations . . . 57
6.1.3 Physical Implementation . . . 57
6.1.4 Efficient Second Phase Sort . . . 58
6.2 Conclusion . . . 58

Bibliography 59

List of Figures

2.1 The breakdown of the entries sorted 10
2.2 The two phases of an enterprise database sort 11

3.1 Processor mapping of phase one merge 17
3.2 Chip mapping of SAISort with "fast lane" 18
3.3 Simulation results of records sorted per joule for 1 GB and different buffer sizes 23
3.4 Processor mapping for buffer sort 24

4.1 The layout of the AsAP2 chip 34
4.2 The flow of the payload seperated sort 36
4.3 Processor mapping without using on-chip memories 39
4.4 Processor mapping using on-chip memories 40

5.1 Heatmap of active percentage of each processor in the key separated sort . 46
5.2 Graph of records sorted per joule with varying processors 53
5.3 Graph of records sorted per joule with varying quantities of on-chip memory 54

List of Tables

2.1 A bitonic sequence being split into two bitonic sequences 6
2.2 The steps to make a sorted list from two bitonic sequences 7
2.3 The steps to turn a randomly ordered list into a bitonic sequence 7

5.1 Time and energy to sort several quantities of unsorted records keeping the payload and key together 48
5.2 Time and energy to sort two different quantities of unsorted records keeping the payload and key separated for the initial sort 49
5.3 Number of passes for the second phase, time, and energy to sort varying quantities of unsorted records 51

Chapter 1

Introduction

Energy efficiency is becoming more and more important in today's computer market with the growing number of mobile applications and the growing sizes of data centers. As storage space, and physical size of the modern large scale data center grows, so does the cost to power it. Aside from simply the cost to power the systems themselves, these large database centers produce a large amount of heat, and cooling down the system has become an issue [1]. This cost is taken in high consideration as new data centers are being created, which generates a demand for ways to run database centers efficiently.

1.1 Enterprise Database Sorting

Sorting is the ordering of data entries that were previously out of order. The most common example of sorting is ordering a list of entries so that their keys go from the lowest value to the highest or visa versa. Structured Query Language, or SQL is a programing language for data management that utilizes sorting. It was created by Donald D. Chamberlin and Raymond Boyce in 1974 [2]. In SQL, sorts can be executed using the Order By instruction [3]. Finding duplicates in a list is also vital, and can be achieved through the Distinct function on SQL [3]. This function also begins with a sort, putting all the lists in order. The function would then go through the list and delete entries that shared a key with their neighbor. Sort functions are so important in fact that they are one of the most used functions in database systems [4].

Sorting has been the subject of a significant amount of research. The two major types of sorting are, internal and external sorting. The differences between these two types of sorts is explored in further detail in Chapter 2, but in general terms: internal sorting takes place in main memory, and external sorting requires external memory. Internal sorting has received a considerable amount of attention in the Computer Science community, both parallel and sequential implementations [5]. External sorting though, is the more useful subject when looking at creating an energy efficient database sort for large datasets. External sorting for uniprocessor systems has been deeply explored, but there is still room for advancement in the subject of external many-core sorting algorithms that this thesis focuses on.

1.2 Energy Efficient Database Sorting

Data centers in the United States spent a combined $4.5 billion in 2006 to cover power consumption, accounting for 61 billion kilowatt-hours and 1.5% of total United States electricity consumption [6]. Power consumption of database systems can be so large in fact, that after a few years of running a common database system, money spent to power the system can easily be more than the money spent to purchase the hardware in the first place [7]. Since data centers spend a large amount of their processing time sorting large databases [4], an energy efficient sorting algorithm for a database system could go a long way in reducing the overall power consumption and operating throughput of a data center.

Even on a per company basis, energy consumption can add up quickly. Power consumption is such a large factor in database center creation that weather condition, and power costs are often taken into account in the creation of new database centers. Within the last two years for example, both Google and Hewlett Packard have made plans to create database centers in Finland, taking advantage of the cheap power in the country, along with the cold weather to help with cooling costs [8][9]. Looking at the high power cost of running these centers, increased energy efficiency in database systems could potentially save millions of dollars for database centers in the United States.

The processors used in database centers are most often built with speed and

throughput valued more than energy efficiency. Speed is often times preferred to power efficiency for every day tasks. This thesis does not propose the the general purpose CPU be wholly replaced, simply aided. If a co-processor were used that could do tasks that the CPU completes often with much higher energy efficiency, then the user could enjoy speed for their normal operations and greatly increased energy efficiency. With database systems, there is an obvious task that takes up great deal of the processor's time; sorting.

1.3 Project Contributions

This thesis proposes a novel approach to first phase external sorting algorithms specifically to utilize a many-core chip as a co-processor for a general purpose CPU. The goal of the algorithm is energy efficiency. Since this thesis explores the first phase of an external sort, the goal is to set up the second phase to sort as power efficiently as possible. As the second phase of the sort is considered the merge phase of the sort where the algorithm merges all of the sorted lists into one entry; the goal of the first phase of the sort is to make the initial sorted lists as large as possible. For completeness, this thesis explores the possibility of using the many-core system for the second phase of the sort as well.

As this thesis focuses on the fairly specific implementation of a many-core system, it is imperative that parallelism is exploited as much as possible on the chip. Taniar et al. began the exploration of external sorting on many-core systems [5]. Taniar et al. pointed out that one of the more common external sorts are a version of a merge; either a binary merge or a multiple list merge [5]. Merge algorithms can be very effective for uni and multi core systems, but the merge algorithm has a few downfalls that are explored in Section 2.2.2 on page 9.

1.4 Organization

The remainder of this thesis is organized as follows. Chapter 2 gives an overview of enterprise database sorting. Chapter 3 discusses the implementations attempted. Chapter 4 is a description of how the algorithm would be created and implemented. Chapter 5 looks

at the results of the implementation, along with analysis of it. Chapter 6 discusses future work for the project, and finishes up with a conclusion.

Chapter 2

Overview of Enterprise Database Sorting

2.1 Internal Sort

2.1.1 Background

Volatile memory, or main memory such as DRAM loses its stored data without power but operates at drastically faster speeds than non-volatile secondary memory. Internal sorting algorithms take place completely in this main memory. As most secondary memory sources such as hard drives and tape drives require comparatively large access times of approximately 100,000 times longer than the average RAM [10], internal sorts often run much quicker than their external sorting counterparts. The downside of internal sorts, is that RAM memory is comparatively expensive, consumes a large amount of power, and is volatile memory. It is therefore impractical to make large databases out of RAM, and internal sorts are not a practical option for large databases.

Internal sorts have been the focus of a considerable amount of research. Both parallel and sequential algorithms have been given a great deal of consideration [11]. Most external sorts have an initial phase that is effectively an internal sort. Because of this, the efficiency of the internal sort can affect the overall completion of an external sort. As database systems usually deal with large amounts of data, external sorts are focused on

in this thesis. Internal sorts are looked at for their ability to increase the efficiency of an external sort.

2.1.2 Commonly Used Internal Sorts

During the course of the research for this thesis, many internal sorts were explored as possible options for the first phase sort presented here. As these sorts are discussed later in this thesis, a brief introduction for the sorts explored is given here.

Bitonic Merge Sort

Bitonic sorting was introduced by K.E. Batcher in 1964 [12]. To understand a bitonic sort, it is first important to know what a bitonic sequence and bitonic split are. A bitonic sequence is any list of numbers where the entries either increase to a maximum then decrease, or decrease to a minimum then increase. A bitonic split starts with the first half of a bitonic sequence being compared in order with the second half. Every entry that is lower in the second half (assuming that the bitonic sequence increases then decreases) is exchanged. If the bitonic sequence decreases to a minimum then increases, then the opposite switch function is performed to achieve two bitonic sequences. An example of a bitonic split is shown in Table 2.1. As can be observed in Table 2.1, the two resulting sequences after the split are also bitonic.

Bitonic Sequence	3	7	16	10	9	5	2	1
Two Bitonic Sequences After Split	3	5	2	1	10	9	7	16
	Sequence 1				Sequence 2			

Table 2.1: A bitonic sequence being split into two bitonic sequences. Here the 3 is compared to the 9 in the second half of the sequence. Since the 3 is lower, the two numbers remain where they are. The 7 and 5 are then compared. As 5 is lower, it is switched with 7. The 16 and 2 as well as the 10 and 1 both switch places as well.

Taking this same group of numbers a few steps further, we can get the list sorted by completing more bitonic splits, ending with each neighboring pair switched in the same way the previous splits were. As can be seen in Table 2.2, after 2 bitonic splits, and a pairwise exchange, this list is now sorted from lowest to highest.

Two Bitonic Sequences After First Split	3	5	2	1	10	9	7	16
Bitonic Splits in Quarters	2	1	3	5	7	9	10	16
Sorted List After Pairwise Exchange	1	2	3	5	7	9	10	16

Table 2.2: The steps to make a sorted list from two bitonic sequences

It has now been shown that it is possible to take a bitonic sequence and turn it into a sorted list, so the only remaining task would be to get a list of random numbers into a bitonic sequence to begin with. This is accomplished by performing a pairwise exchange in alternating directions. Then splits take place, starting with a four entry split, increasing in size until each half of the list is a bitonic list, with the beginning list ascending and the ending list descending. One more pairwise exchange completes the transformation, this time putting lower entries to the left on the first half, and to the right in the second half. It can be seen in Table 2.3 that this can go from an unsorted list, to a bitonic sequence following these steps. This bitonic sequence could then go through the process described above to come to a sorted list.

Randomly Ordered Entries	5	3	2	10	1	9	16	7
Pairwise Exchange	3	5	10	2	1	9	16	7
Bitonic Splits	3	2	10	5	16	9	1	7
Bitonic Sequence After Pairwise Exchange	2	3	5	10	16	9	7	1

Table 2.3: The steps to turn a randomly ordered list into a bitonic sequence

Quicksort

Quicksort is a sorting algorithm first proposed by C.A.R. Hoare [13]. It is one of the faster sorts in use, sorting by moving entries in a subsection of the unsorted list in relation to a pivot point. The sort starts off by looking at the entire list and choosing a pivot point. The pivot point alone can change the speed of a quicksort; the closer to the middle element in a sub section the more effective the sort is. After the pivot point is chosen, every item in the list is moved to either before or after the pivot point in numerical order. The function then recursively calls the quicksort function for both the above and below subsections. This continues on until the called quicksort function orders the last two

entries for all of the subsections, at which point the list is sorted.

Insertion Sort

Insertion sorts are simple to implement. The idea behind them is that each item is inserted into a sorted list being formed. This sort starts off with one entry, then the next entry is compared with the first and order them accordingly. The third entry would be compared until its position was found, and so on. The sort does not require very complex code to implement, so it is perfect for the type of system being looked at here. The sort is also reasonably efficient for sorting small quantities of lists, and in the proposed architecture only a maximum of 10 entries could fit in each processor.

2.2 External Sorts

2.2.1 Background

External sorting algorithms assume that the unsorted data, cannot reside completely in the main memory of the machine. This means that non-volatile memory such as hard drives or tape drives must be utilized to store the dataset. This type of sort poses quite a different problem than the internal sort as memory I/O speeds are one of the largest bottlenecks for the system. In other words, it is not the computation that takes the most time in most external sorts that decide their efficiency, but rather the number of times that times that data must be read and written to the secondary memory.

Most large database systems employ either a single, or multi-core setup, with a large amount of secondary memory on many disks to reduce the I/O time of the system. As this has been the trend, the majority of research into the subject of external sorting for database systems has been for sequential sorting where only a few cores are used [14]. Taniar et al. was one of the first to look into the idea of utilizing a many-core systems to create parallel external database sorts [5]; their work is explained in chapter three of this thesis. The subject of many-core external database sorting is still a largely unexplored field, and it is the author's opinion that many-core database systems can achieve considerable energy efficiency, as is presented in this thesis.

2.2.2 Commonly Used External Sorts

One of the most commonly used external sort methods for uniprocessor environments is some version of a sort-merge [14]. In the sort-merge external sort, an internal sort is first executed as the first phase of the sort. This first phase sorts lists as large as the main memory allows. After this first phase is complete the algorithm will then perform a merge, either binary or otherwise, until all of the lists have been combined into one. This method does not translate very well to many-core systems however; a many-core system could have each core working on a separate binary merge, but as the number of lists to be sorted gets smaller than double the number of cores, the cores will be underutilized.

A possible solution to utilizing a many-core system to sort a list would be to break the sorted list up into ranges for each core to process. This solution actually does work fairly effectively if the range of possible key values being sorted is not very much larger than the number of keys being sorted. If this is the case, than each core could be assigned a range that would be able to ensure that the work is fairly evenly distributed across the cores, and is explored in detail by Taniar et al. [14]. As discussed later in this thesis however, this method does not prove to be an effective solution when the range of possible key values is considerably larger than the number of keys sorted, as an efficient workload distribution would be near impossible with randomly numbered keys in in a large range.

Commonly Used External Sort Benchmarks

As external sorting for database systems is very important to the database industry, there are a number of external sort benchmarks that have been proposed over the years. The most commonly used sort benchmarks are the Gray, Penny, Minute, and Joule sorts. The Gray sort is a measurement of how many TB of entries can be sorted in one minute. The Penny sort is the amount of data that can be sorted for a penny's worth of system time. Minute sort is the amount of data that can be sorted in 60 seconds or less. The JouleSort is one of the newer benchmarks that measuring the number of records that can be sorted per joule. The JouleSort is described in more detail in the next section.

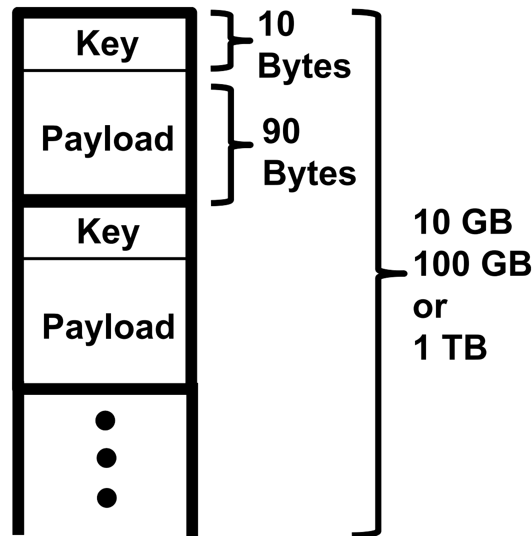

Figure 2.1: The breakdown of the entries sorted

JouleSort

The JouleSort is a database benchmark that was proposed by a group from Stanford [15]. They saw a need in the database benchmark field for something that would measure the power efficiency of a sort. They pointed out that energy consumption, both for cooling and power the database systems accounts for millions of dollars a year that companies spend for operating a data center. They therefore thought that energy efficiency should be taken into account when trying to measure database sorts against each other.

They proposed that the benchmark use 100 byte entries, the same as the majority of popular external sorting benchmarks. The JouleSort also requires that the entire list to be sorted must begin and finish on completely non-volatile memory to assure that the sort is a true external sort. The benchmark has 3 different size classifications: 10 GB, 100 GB, and 1 TB. The entire system's power consumption is measured, including the cooling costs for the system while it is sorting the alloted entries. The number of entries sorted is then divided by the amount of joules the system required to sort to find the number of entries per joule the system can sort. The sort proposed here follows the requirements of the JouleSort benchmark, focusing on the 10 GB sort.

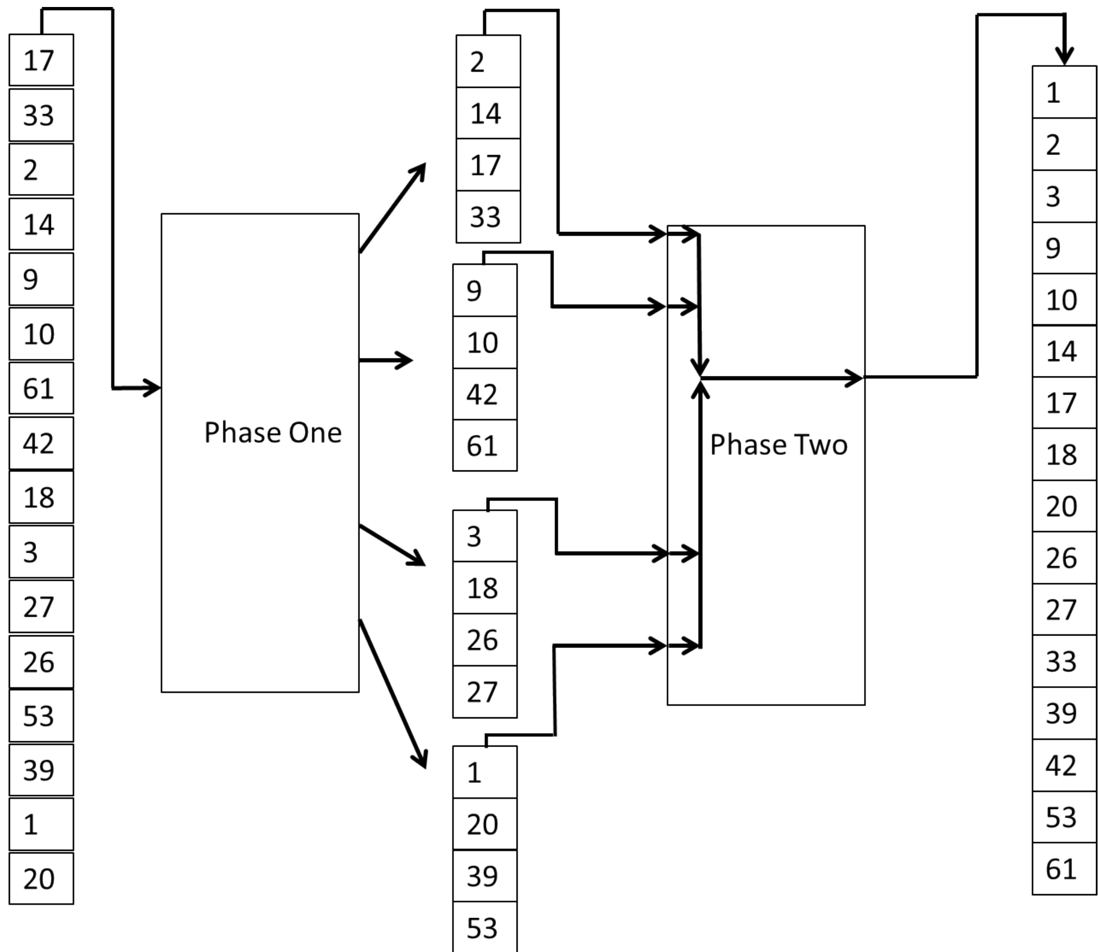

Figure 2.2: The two phases of an enterprise database sort

2.3 Related Work

2.3.1 Parallel Database Sorting

Taniar et al. [14] proposed an approach to database sorting that would take advantage of parallelism for many-core systems. They also proposed three specific types of sorts that would lend themselves to parallelism. The three new sorts that Taniar et al. proposed used redistribution and repartitioning of data to increase the speed of the sorts [14].

The paper from Taniar et al. [14] looked into why sorts are efficient. In the paper, Taniar et al. pointed out that external database sorts can be effectively described in two distinct phases: a sort phase and a merge phase. The sort phase, or first phase, is where the data goes from one unsorted list, to multiple sorted lists as can be observed on the left

side of Figure 2.2. The merge phase, or second phase, is where all of those sorted lists are merged until there is only one large sorted list as shown on Figure 2.2. It is pointed out in the paper that the first phase of the sort has a direct and substantial effect on the sort time. Because of this, it was decided to focus on the first phase of the sort.

The paper focuses on the second phase of the sort, the final merge. The paper also proposes the novel idea of redistribution and repartitioning of data to increase the speed. The redistribution proposed takes place after the first phase. There would be a range of entries assigned every available processor, splitting the range evenly. After the first phase, each entry is sent to the processor for the range in which it belongs. Repartitioning is very similar, but the phases are merged; the sort starts out by partitioning the entries to the processor that correlates with the range the entry belongs in. Each processor then does its own local sort, which would result in one sorted list.

Taniar et al. show that their new methods create a considerable of parallelism [14]. There is one major drawback though: both methods rely on the sorted keys to not have a large range of available positions. The authors mention that this can be an issue, and attempt load balancing by creating more ranges than processors, so that ranges with a small number of entries can just get multiple ranges. This solution would help if the total range is not extremely large however. This thesis, focuses on a sort of 10 GB of entries, with a range of entries over 100 times the size. In a random environment it would be near impossible to effectively load balance each processor in this situation.

2.4 Proposed Enterprise Database Sort

This thesis proposes a novel approach to the first phase of an external database sort, utilizing a low power many-core system. The algorithm proposed takes advantage of the large amount of parallelism that is possible with a many-core system. As the sort is created with the focus of setting up the second phase of an external database sort, the sort creates sorted lists that are as large as possible.

The algorithm is proposed to operate on a low powered many-core system, as a co-processor to a general purpose CPU. With this setup, the main processor is still be able

to utilize the system resources as the co-processor is sorting the first phase of entries. This thesis also explores the most energy efficient setup for the proposed algorithm. Varying sizes of memories, of processors on-chip, and clock speed are all explored.

Chapter 3

Attempted Implementations

3.1 Phase One Sorts

The first phase of an external database sort is extremely crucial. This is because the size of the sorted lists created dictates the number of passes through the data that the second phase requires. Because the first phase is so important, it is the focus of this research. Many avenues were explored in an attempt to create the most efficient sort possible. In the end, the SAISort was decided as the most effective option, but the attempted sorts helped lead to the end decision, and is described here.

3.1.1 Sequential Internal Sorts

Popular Uniprocessor Sorts

One of the first implementations attempted was modifying a popular and efficient uniprocessor internal sort to work on a many-core system such as Quick Sort. Because of the limited memory that each processor has on the many-core framework assumed, it was quickly decided that using each of the processors to sort individual lists as if each were a uni-core processor would not be efficient. The internal memory on each of the chips is assumed to only hold 10 entries at most, so without further functions the phase one lists would only be 10 entries long. The many-core chip general characteristics used also dictate that all of the processors cannot be loaded and unloaded in parallel, so time would be

wasted simply bringing the entries to and from the processors.

Many uniprocessor sorts were also ruled out because of their complexity. The instruction memory of the setup was a limiting factor. It was assumed that each processor could store only 128 lines of code. With only 128 lines of assembly language code, it turned out to be very difficult to program any algorithm that was very complex.

Altering a uniprocessor sort so that it would work across the entire chip was also explored. In this algorithm, the list would be loaded whose entries spanned all of the processors. The quicksort was attempted in such a fashion. With this implementation, a lot of processor to processor communication would be required for the comparisons though, which would create a bottleneck. Each of the subroutines called could possibly be ran on different processors, but this parallelism only becomes available after a number of subroutines have been called. For example, on the 164 core chip being assumed here, 163 subroutines must be called before the chip is completely utilized.

In the end, the SAISort that was used and described here was a hybrid of the two previously mentioned attempts. The sort is explained in more detail in subsequent sections, but the general idea of the sort is that each processor sorts as many entries as it can hold, then outputs the lowest entry on to the next processor in a chain. If only the number of entries that can be held by the processors is input, then the output of the last processor would be a sorted list.

3.1.2 Merge Sorts

Bitonic Merge

A Bitonic Merge sort was explored extensively. The sort implements very efficiently on a many-core system, as it allows for a considerable amount of parallelism to be taken advantage of. The many core system being designed for also lends itself well to the implementation of the system. The network of a Bitonic Merge Sort moves towards a sorted list in a fairly linear fashion, with an entry point of an unsorted list, and an exit point of a sorted list which fits with the assumed chip architecture. The network does require that nodes can send information long distance to contact processors that are more than

neighboring, but again this fits with the setup as long distance communication is available.

In the end, it was simply the assembly language code size that did not allow the implementation of this sort on the chip. Creating a Bitonic Merge algorithm using the 128 lines of assembly language code available was attempted, but the algorithm was just too large. If it were possible to shrink the algorithms size down to fit in the alloted instruction memory space, or if the instruction memory were increased, a Bitonic Merge Sort should be explored.

Merge Tree

To take advantage of the parallelism that the multi-core processor offers, it was decided to investigate using a merge tree style sort. To accomplish this, the entries would be loaded into the chip, then merged along a path through the processor until the result was a sorted list. The first processor of each line would be loaded with the maximum number of entries. The first processor would sort these entries. The sorted lists would then be merged on the chip into one continuous sorted list. To reduce bottlenecks, it would also be beneficial to use processors as buffers with enough processors to buffer every entry so that the preceding merge processors are not required to wait for the next merge processor.

As can be observed in Figure 3.1 however, a merge tree does not fully utilize all of the available processors. In Figure 3.1, only 22 lines can be input, creating sorted lists of only 220 entries at the output assuming each processor were holding the maximum number of entries. This solution would require less processing time than other solutions, as other than the very beginning, all of the comparisons are simply to merge two sorted lists. The number of lists that would be sorted are enough of a limiting factor to make this option not worth it though. When sorting 220 entries, the second phase of the 10 GB sort being considered here would require 19 passes through the data to come to one list. If all of the processors can be completely filled, it is possible to start off with sorted lists of 1641 entries which would only require 17 passes through the data. As is discussed later, the C++ program used for the second phase sort takes 7.5 seconds and 371 joules for every pass, so this sort would use over 1,000 joules more than a sort that sorted 1641 entries.

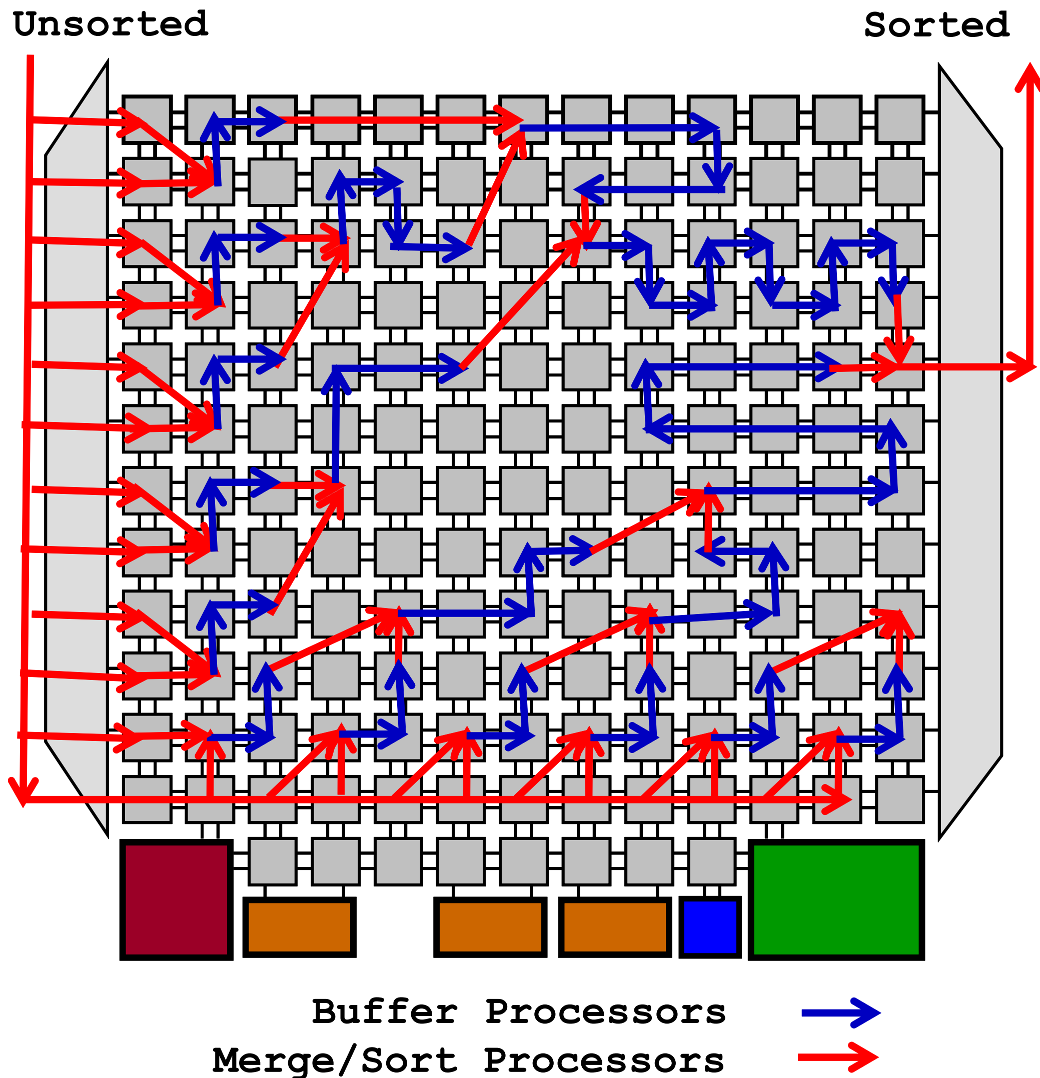

Figure 3.1: Processor mapping of phase one merge

SAISort With Fast Lane

The sort that was decided upon was the SAISort which is described in Section 4.1 on page 26. It is effectively a single processor sort on all available processors, with each processor being linked sequentially. Because all of the processors are linked sequentially, the bottleneck of the sort is the fact that every entry must move through every processor in the chip. In an attempt to fix this bottleneck, the idea of a fast lane was explored.

The general idea of this fast lane would be that if there were an entry that could be assumed to be lower than a certain subsection of the entries it would be forwarded. As can

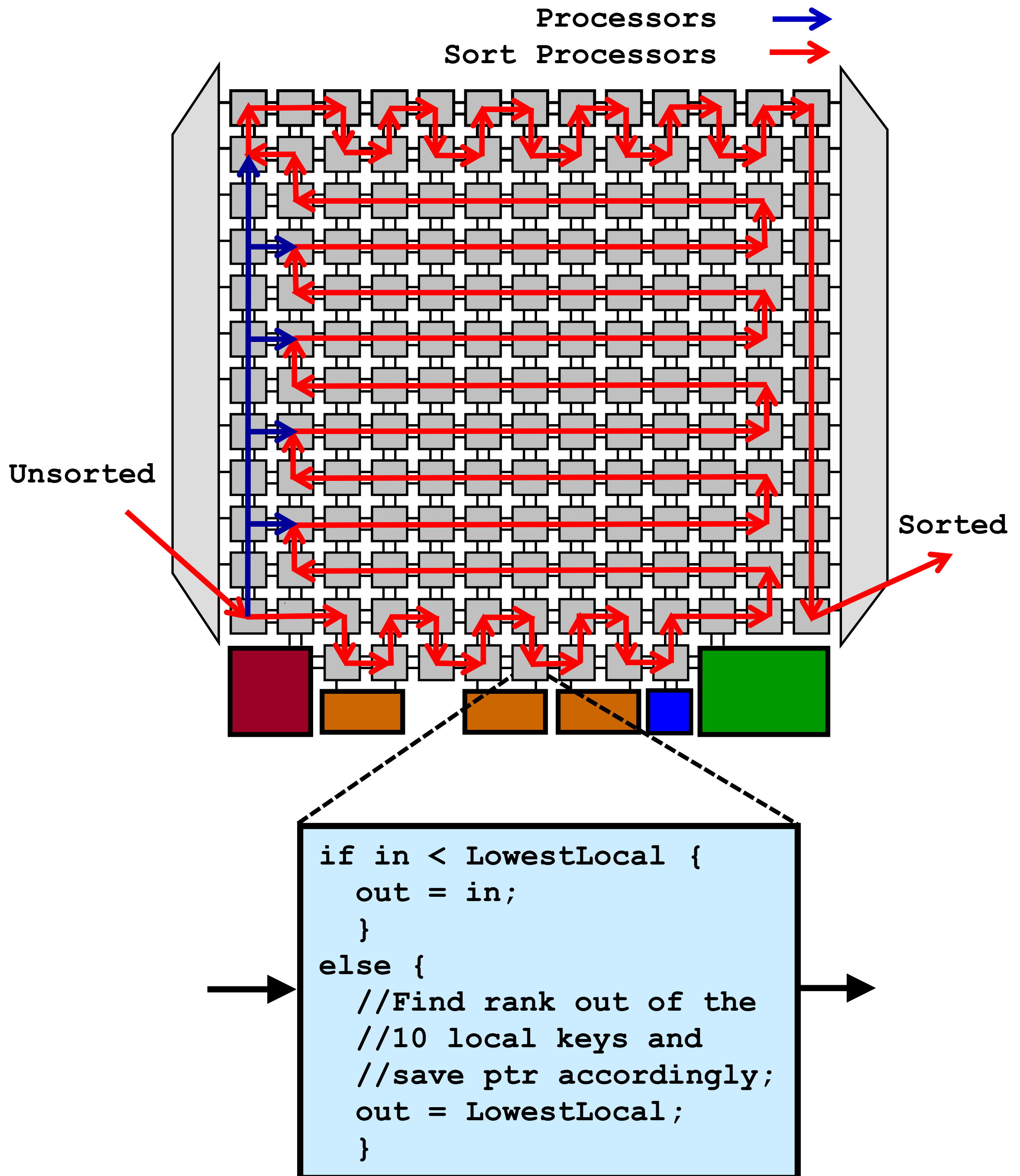

Figure 3.2: Chip mapping of SAISort with ”fast lane”

be seen from Figure 3.2, the entry would be forwarded along the side of the chip as far as it could be moved without looping back and forth sequentially through all of the processors. This would save a considerable amount of processing time. This method is actually fairly close to the idea presented by Taniar et. al [5]. In this implementation each of the different rows would be the "buckets" described in their paper.

The difficulty of this method, which in the end was the reason that it wasn't used is that ranges of numbers would be required that would not overflow a row. This limitation was also described Section 2.3.1 on page 12 when discussing Taniar et. al [5]. As mentioned earlier, there are 100 times the amount of possible key values for the keys than there are entries to be sorted. This means it would be near impossible to create 6 ranges that would equally distribute the entries across the processors with randomly generated keys.

3.2 Phase Two Buffer Sort

A phase two sort was created to run on the many-core processor described in this thesis. To create an optimal sorting algorithm the number of memory accesses required should be minimized. If the SAISort described later was used for merging multiple sorted lists together then a wait for the output would be required to see what list the entry came from before another input could be sent. One of the larger time consumers would be the memory accesses, so a simulation was created, described in Section 3.2.3 on page 25 that showed using the SAISort would not be an effective use of energy for merging a large amount of presorted lists because of the amount of time that each entry would spend moving through the chip. Implementing a string of merges while buffering entries in processors, removed the requirement of waiting for each new entry individually.

In designing this Buffer Sort it was found that there is a trade off between merging more lists at one time, and fewer lists at one time. If a large number of lists are merged at one time, then a small number of passes through the data are required before one list is reached, meaning there are fewer total reads and writes of the entire database stored on the external memory. If a smaller number of lists are merged more phases would be required, but there could be larger buffers, meaning the chance of stalling the chip would be reduced

because the buffers could hold many entries.

Because the largest downfall of the previous method was the amount of time each entry would take to move its way through 164 processors, each entry had to go through a smaller number of processors before it exited the chip. It was decided that it would be more efficient to make a system that would use physical buffers to store extra entries from a list. It was also known that the larger the number of lists combined at once, the less amount of combines would be required before the lists were all merged into one. When looking at how to plan out the processor, it was found that if more than 24 entries were merged at one time a considerable amount of processors would be required for use as transmission lines, to the point that the chip would stall its output as entries worked their way around the processor, which is what should be avoided. Because of this, it was decided to use a setup that would allow the combining of 24 lists at a time. After deciding on how to execute the sort, it was implemented onto the AsAP2 processor.

3.2.1 Implementation at Processor Level

There are three distinct programs running on the processors in the Buffer Sort: transmission, buffer and merge programs. Because of the nature of the AsAP2 processor, it was decided that the most power efficient way to send the entries to the buffers would be to use processors that would function solely as transmission processors. These processors would be in charge of making sure the data ended up where it needed to be. The buffer program functions purely as a buffer to the merge lines so that they don't stall during a memory access. The merge processors are where the real computation happens; each processor merges two sorted lists into one. Setting the merge processors up in a row allows them to merge multiple sorted lists together.

Transmission Processors

The transmission program should send off the entries to the correct buffers as soon as possible. Because the transmission processors do not store any entries on their internal memory, they simply decide where the entry belongs, and move it along. The processors in the AsAP2 chip can simultaneously output to both of their output FIFOs.

This simultaneous output was taken advantage of by having each transmission processor broadcast the incoming entry to two different transmission processors. When an entry comes into the processor, it loads the tag to check if the entry should be forwarded. If the entry should be forwarded, the key is output simultaneously to both its output FIFOs. If the entry does not belong, the entry in the FIFO is simply deleted, and the program waits for the next input.

Buffer Processors

The buffer processors job is to hold enough entries that the merge processor is never without entries to merge. It also should use as few instructions as possible to keep the speed of the program up. The processor accepts inputs from the previous processor to fill up all of the processor's internal memory. The next entry that comes after the processor is full causes the processor to output the lowest stored entry before the input entry is stored. When a reset command is input to the processor, it outputs all of the entries that are stored to the next processor in the line. It then outputs the reset command to let the neighboring processor know that that was the end of the list. After the reset command is sent, the processor resets itself and begins accepting entries from the next sorted list.

Merge Processors

The merge processors merge multiple lists into one. Because it is assumed that the processors can communicate with only nearest neighbors, the processors merge two lists at a time. The algorithm itself is fairly simple, the lowest entry from the two lists are simply compared, with the lower entry being output to the next processor. The next processor then compares that entry with the lowest entry from another list. This continues on until all of the input lists are merged.

At the processor level, the program completes the merge with as few operations as possible, as the merge is the bottleneck. The program only loads the first byte of incoming data before it checks for a reset code. If there is no reset command, it downloads the rest of the key, leaving the payload in whatever FIFO the processor utilizes. The processor then loads a byte from its other entry point to check it for a reset code, then the key of that

entry. The program then compares the two keys, outputs the lower key and outputs the payload from the input FIFO to the output FIFO for the lower entry. The next input is then compared with the stored key and this continues until a reset code is found. The reset codes are a bit of data sent at the end of a list of sorted entries.

When a reset command is found from one of the input FIFOs, the chip stops receiving data from that FIFO. The chip then begins to output directly from the other FIFO to the output of the chip. This situation occurs when one list has finished before the other has, meaning that all of the remaining entries from the remaining list have larger valued keys than all of the entries on the finished list. Because of this, the output list is still ordered correctly by simply outputting the remaining list to completion. When the second list reaches completion and the second reset command is input, the program then sends out a reset command to the next processor letting it know that the input lists are completed. The processor then resets itself and begins accepting inputs from both attached FIFOs to begin merging the next two lists input.

3.2.2 Chip Mapping

The buffer sort should be mapped out in a way that as many lists as possible are merged at once, while buffering enough entries that memory accesses cause as few stalls as possible. To accomplish this, there must be a decent buffer for the merge processors. The merge processors should also be next to each other. The first processor in the merge line can take two lists through buffers, and the rest of the merge processors in a row take a buffer and the neighboring merge processor as its entries. In this way the chip can merge many sorted lists, while each processor performs the much easier task of comparing two keys together.

After much deliberation and simulation it was decided that it would be most efficient to merge 24 lists at the same time. Different configurations for the processors inside the chip to optimize the buffer space were worked out, and it was found that up to 24 buffers could fit in a way that would have effectively no stalls (if all of the entries come from one of the two lists furthest from the output, there will be a few stalls, but this is very unlikely and even if it does occur will only slightly slow down the chip). When adding

a 25th buffer to the chip, another transmission line and merge line would be required on top of the two already there, which takes away a large amount of processors available for buffers. This would make stalls fairly prevalent, slowing down the chip considerably. After it was known that there could only be up to 24 merged lists at a time, a simulation was created to model one gigabyte of data being sorted using a varying number of merged lists from 2 to 24, see Figure 3.3. The simulation showed that 24 lists merged would give the quickest speed, and because the main power costs are the fixed costs of the memory and the development board, 24 is the most power efficient solution as well.

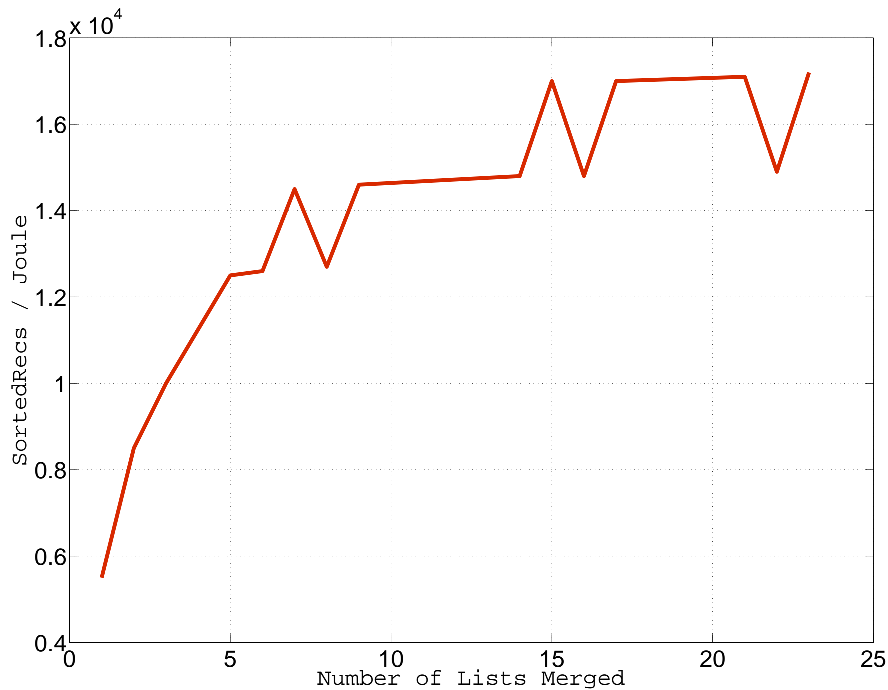

Figure 3.3: Simulation results of records sorted per joule for 1 GB and different buffer sizes

There are two rows in the middle of the chip that merge the incoming list with the running list that is going down the chip, see Figure 3.4. At the far right, the two merged lists are combined before they are output from the chip. The perimeter processors are used to transmit the data to fill the lists. The bottom row can be used for administrative processors, and the rest of the processors are used as buffer space, where their data memory and FIFOs are used to store three entries per processor if they key and payload are kept together, and 15 if they are separated. As the entries go into the processor they are tagged

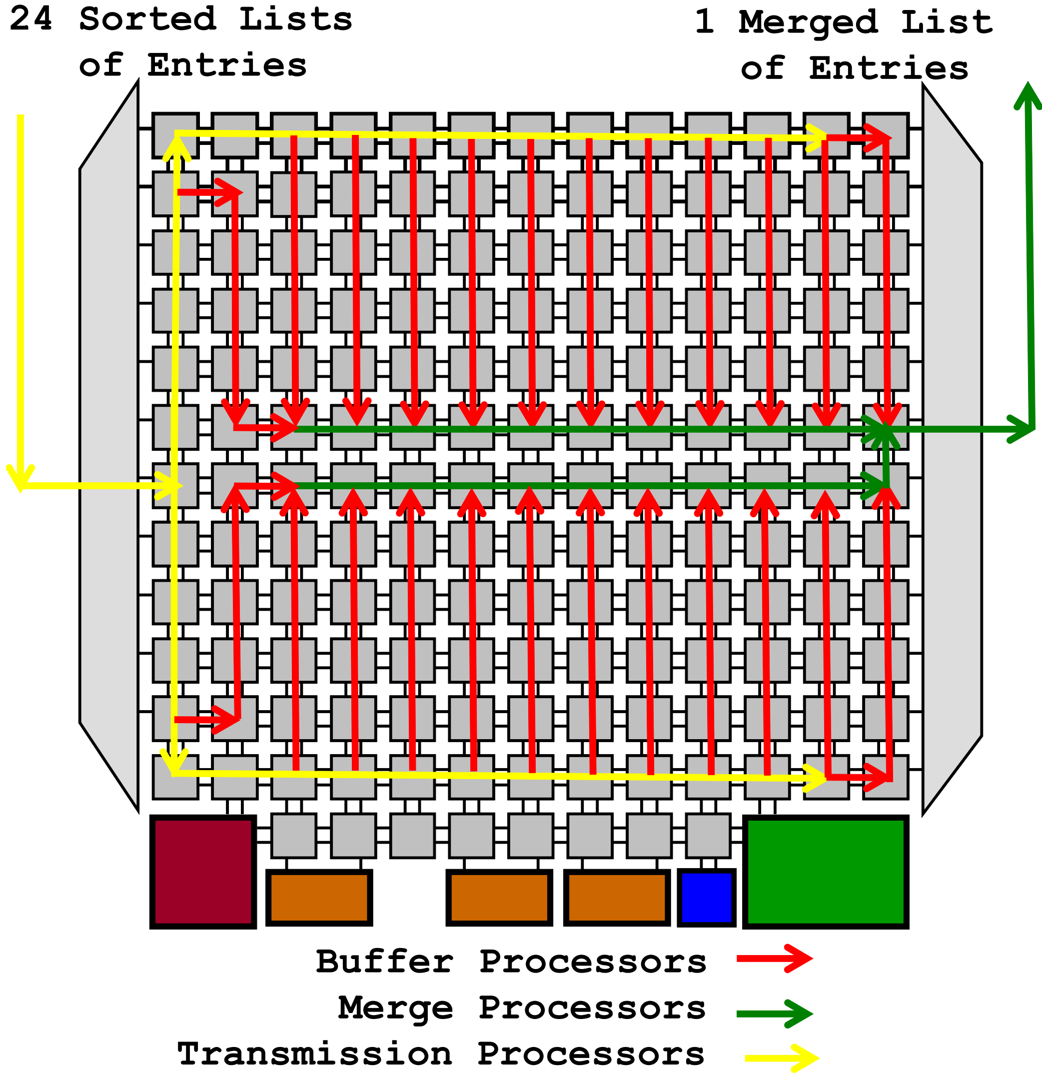

Figure 3.4: Processor mapping for buffer sort

by an administrative processor, which specifies which list the entry came from.

3.2.3 Benefits / Limitations of AsAP2 Implementation

The Buffer Sort described here has limitations that outweigh the benefits of such an implementation. The benefit would be that the entire external sort could take place on the many-core processor, which would leave the general purpose processor available. The limitation however is the speed of the AsAP2 chip, running at 1 GHz, combined with only being able to merge 24 lists at a time makes the chip simply not fast enough to accomplish this fairly sequential task of merging. If more parallelism could be taken advantage of, the AsAP2 might still be a viable option, but right now it is not.

Because the physical AsAP2 chip does not currently have a viable method to output information, simulations were relied upon to test the feasibility of the algorithm created. The described program was run on the AsAP2 chip to find how quickly the algorithm merges input lists. This information was used to create a MatLab script, to accurately simulate the amount of time it would take the previously described merge to merge lists of 329 (the total sorted list size of the SAISort with the payload and key attached without using the on-chip memories) into one 1 GB list.

The simulation showed that this sort is not nearly as efficient at the phase two database sort when compared to a general purpose CPU implementation. The simulation showed that it would take 39.799 sec to sort each individual 1 GB of information. This equation was used: 50 ns (RAM access time) + 34.375 ns (transfer time for 110 B at 3.2 BG/s) + 156000 ns (time to fill up all of the processors being used) + 100000000 ns (number of entries) * 1000 ns (time to output one entry). It would take 100.157 sec to merge the 10-1 Gb lists into one final list. The total time for the merge would then be 10X39.799 s + 100.157 s = 498.147 seconds. This is in comparison to the 142.5 seconds for the phase two C++ implementation described later. The Buffer Sort described here takes 3.5 times the time to merge the same lists, and the power consumption is only approximately half. Because of this, for the remainder of this thesis it is assumed that the C++ implementation would execute the second phase of the proposed sort.

Chapter 4

Proposed Sort

4.1 Phases

To implement an efficient database sort, an algorithm that minimizes the number of memory accesses is desirable. After looking at past sorts, one of the more important factors for efficiency is the beginning of the sort, where there are no sorted lists. To look at this problem, the total sort was split into two phases as described in Section 2.3.1 on page 11. In the first phase the sort goes from a completely unsorted list to many small sorted lists. The second phase is a merge to combine the sorted lists into one sorted list.

The number of entries per list that the first phase can sort directly and drastically affects how long it takes for the merge to output one sorted list. As the first phase is so critical, the first phase is focused on. Because many-core systems will very likely play a pivotal role in the future, it was decided to create an algorithm that would try and take advantage of the parallelism that a many-core system makes possible.

A few assumptions about the many-core system the sort would be running on were used during the creation of the algorithm that were touched on earlier in this thesis, but are described in detail here. It was assumed that there was some sort of communication between the cores on the chip, so that the entries could be passed around the processor. It was also assumed that each core would have some sort of dedicated internal memory, though the created algorithm could easily be changed for a shared memory setup.

4.1.1 Phase One / SAISort

To keep things efficient, the algorithm should output a sorted list that is as close to the size of the entire internal memory of the chip as possible for the first phase. To accomplish this, the algorithm should have many cores working in parallel that stay coordinated to create one large list. It was also important to keep the complexity of the sort on each core simple so that it could easily be used on different systems.

It was decided that the way to accomplish this would be an insertion sort on the processor level, with each processor being connected in a line. Because the algorithm uses an insertion sort in a serial array, it is called the Serial Array of Insertion Sorts or SAISort. The general idea is that each processor keeps as many entries as it can on its internal memory. As each new entry comes to the core, it inserts the entry in the stored list where it belongs, and outputs the lowest entry to the next processor. If only the number of entries that can fit in the chip's internal memory is input to the chip, by the time the list leaves the chip, it is sorted.

After all of the processors are full, a way to quickly output all of the entries stored in the chip, so a separate output mode was created. After all of the entries are input, a code is input to put the first processor into output mode. The first thing the processor does is send the output code to the next processor, so that all of the used processors go into the mode as soon as possible. When the processors are in their output mode, they simply output all of their saved entries, then begin moving entries straight from the input to output of the chip to reduce the time it takes to move the entries into the internal memory, then back out.

Implementation at Processor Level

The processor implementation is straight forward. When the first entry is input, it is stored as the lowest entry in the memory. The next entry is then compared with the first, creating a sorted list of two entries. This continues until the internal memory is full. The entries that come after the memory is filled are compared first with the lowest entry. If the new entry's key is lower than the lowest in the memory, the new entry is immediately

output to the next processor. If the entry's key is higher, then the lowest entry is output to the next processor. The new entry's key would then be compared up the stored list until its correct position were found. The incoming entry would then be loaded into the correct position.

Each processor also must be able to output all of its stored entries after the total number of entries has been input. To accomplish this, there is an output mode described previously. To engage the output mode, a command byte is put on every entry that holds the output and reset command. The program checks all entries to see if the output command byte is detected. If the output command is detected, the processor sends off the output command to the next processor then outputs all of the saved entries in its main memory. When all the saved entries are output, the algorithm simply move entries from the input straight to the output. The processors then checks the entries that are moved from input to output for the reset command byte. If the reset code is found, it is sent on to the next processor. As soon as the reset code has been sent, then resets itself to be ready to receive the first entry of the next list to be sorted.

Parallelism

Even though every processor operates on its own list independently, there is a large amount of parallelism with the SAISort. When the program first starts up, all of the processor's memories are empty, so the first entry only has one processor working. The first entry after the first processor's memory is filled up starts the second processor working as well, so by the end of the first list, all the processors in the many-core system are working at the same time.

After this initial startup, all of the processors continue to work in parallel because the next unsorted list is input as soon as the first processor sends out its reset command. In this way the chip can fill up with new entries even as it is outputting a sorted list. When the final list is being output, there is a period of time where all of the processors are not working, though the total time that all the processors aren't working is close to negligible as there are hundreds of thousands of sorted lists being output for any such sort.

The workload is not exactly evenly distributed across all of the processors. As

the chain of processors gets nearer to the end, the entries are closer to being sorted. The entries that go into the first processor are spread out in a random fashion, so there is a high chance that the processor only compares keys with the first or second entry before the correct position is found. Because of the nature of the algorithm, the entries that make their way to the processors near the point they are output, are lower than the entries saved in the previous processors. This means that there is a higher likelihood that the algorithm is required to check with many of the saved entries before the correct position is found. Likewise, the closer to the end, the sooner the processor goes into the output mode. For example the last processor would only be required to sort the number of entires that can fit in its main memory before it would go into the output mode which does not require a lot of computation time. Because of these two extremes, it is actually the center processors that end up doing the most computations. This is explored in further detail in chapter 5.

Chip Mapping

When mapping this algorithm, it was necessary to focus on a setup that would use the least amount of power. To accomplish this, it is necessary to use every processor available for the algorithm. It was also important to use the most energy efficient mode of data transfer available to the chip. Nearest neighbor transmission is assumed to be the most energy efficient data transfer, so while mapping the processor data path, every processor on the path should communicate to its neighbor. Care must also be taken that no processor is cut from the chain while mapping so that every available processor is utilized.

SAISort With Payload Attached

The simplest way to execute this sort would be to leave the payload attached to the key as it goes through the multi-core system. This means that there would be no necessity to separate the two, and no pointer system so that the entries could be re-united. The large downfall of this method is, that the internal memory of the multi-core chip would not be able to save nearly as many entries into its internal memory. Even though it would be necessary to add an additional 10 bytes to the entries to save an address to the payload, the size of each entry with payload attached would be five times the size of its separated

counterpart. This also means that in the end, the initial sorted list that the SAISort creates would be one fifth the size of a list that could be created when the key and payload are separated.

The payload attached sort would be implemented fairly simply. The algorithm would also not have as much processing time as the payload separated sort because there would be less entries to compare. In this sort, each processor would save entries until its internal memory was full, then compare down the list for the correct position for the incoming entry. Here, the entire entry would be switched in the internal memory since there would be such a small number of saved entries. For the implementation of this a shared cache on the processor could be used, or just the local memory for energy savings.

The simplest implementation of the algorithm would be to not use any shared cache on the chip. Not using any memory other than what local to each processor would mean that the many-core chip would have the same program loaded to every single processor. No processor would required to transmit data or administrate the memory. The detriment would be that not as many entries would be sorted per run.

For the implementation, a path should be mapped out that maximizes the amount of processors used. The sorting program would then be loaded onto all of these processors. As soon as each list gets through all of the utilized processors, the list is output from the chip as a new list is loaded. This version of the sort could operate slightly faster than its shared cache counterpart as final merge is not required as the entries leave the processor.

Using a shared cache on a chip can increase the number of sorted lists, but can also increase the power consumption of the chip. To make their use effective, the increased number of sorted lists should reduce the second phase of the sort enough that the increased power was worth it. Whether or not a shared cache would be used would also come down to the chip itself, if the cache cannot be turned to a state where its power consumption is low, then there would be less detriment to using it.

To implement the algorithm using the shared cache, different programs would be implemented for some of the processors on the chip. The algorithm would be implemented with sorting processors as described earlier, but processors would also be required to execute a merge. Depending on the way the chip is used, processors would also be used for

transmission and for communicating with the shared cache.

The mapping of the system would be similar to the simpler implementation without shared cache. After the entries had gone through as many-cores as possible, the sorted list would be sent to the shared cache. While the entries were being saved into the shared cache a second set of entries would be input. Sets of entries would continue to be sorted and saved into the shared cache until the cache was full. One more set of entries would be sorted through the cores, and as those entries arrived at the last processor, they would be compared with the lowest entry of the saved lists, with the lowest entry being output from the processor. This merge would continue until all of the entries were output from the chip.

SAISort With Payload Separated

The second way this sort can be implemented is by separating the payload and key. Separating the 90 byte payload reduces the size of the entries considerably. Since the SAISort algorithm utilizes all the available memory on the chip, the size of the output sorted list is increased with smaller entries. To implement this, another key that shows where the payload is stored would be required. Even though the sort being explored here is only 10 GB, the key used should have enough bits of address to implement the same algorithm with a 1 TB sort. Because of this a 10 B key is used to address the payloads.

The payload separated sort utilizes a slightly more complicated algorithm. Because every core can store 5 times the number of entries compared to the payload attached algorithm, moving the actual entries around with every inserted number would take a decent amount of computation time. To remedy this, pointers are set to each of the entries, and when the entries are re-ordered, only the pointers are moved around. In this way, only the pointers are moved instead of all the entry. The rest of the algorithm is implemented the same as when the key and payload are attached.

The SAISort with the payload separated uses the shared cache the same as it does with the payloads still attached. The algorithm saves entries in groups to the shared cache, and does a final merge as all of the entries leave. The algorithm also operates the same without the cache, utilizing the number of processors that maximizes the efficiency of the sort.

SAISort With Payload Saved in Shared Memory

The last category explored was separating the key and payload in the multi-core processor, saving the payload in the shared memory. This sort would operate much as the key separated sort; the chip would save the payload to the shared memory, and attach a pointer to the key that shows where the payload is stored. After the keys were sorted, a processor as the entries were output would combine the key back with its payload.

4.1.2 Phase Two / Buffer Sort

After the initial sort, the sorted lists are merged together into one sorted list. The output of the first phase is thousands of sorted lists. The second phase merges as many of these lists as possible at a time. Each pass through the data reduces the number of lists and increase their size. The algorithm makes passes through the data until all of the lists are merged into one. This algorithm is fairly implementation specific, depending on the size of the processor's internal memory, communication abilities and shared cache implementation. The goal of the program is simply to merge as many lists as possible at a time. An implementation of this Buffer Sort was described earlier in Section 3.2 on page 19 where it was shown to not be the most energy efficient option at least for the AsAP2 chip.

4.1.3 Phase Two / Co-Processor Binary Merge

Another option for the second phase of the sort would be using a general purpose processor. Depending on the many-core processor used, this might be the more efficient method. The first phase SAISort is where the efficiency is gained over a general purpose implementation. When running a merge for the second phase though, a general purpose processor performed slightly better. This is the reason that this thesis proposes the SAISort to operate on a many-core chip as a co-processor to a general purpose CPU. The general purpose processor would run the database system, and use the co-processor to quickly and efficiently sort the first phase of database sorts.

This second phase implementation would be executed by a merge performed on the general purpose CPU, as this is commonly used in database sorting [14]. Having the

general purpose CPU would also allow the CPU to take care of separating the key from the payload for the previously mentioned executions that require it. It would then input a group of unsorted entries into the many-core processor. The output lists would be merged together by the general purpose processor until there was one sorted list.

4.2 AsAP2 Chip Implementation

The previously described SAISort algorithm was implemented and ran on an actual many-core processor. The selected platform was the 167 processor chip created by Truong et al. of the VLSI Computation Lab at the University of California, Davis [16]. A background of the chip is given here, then the details of the SAISort implementation on the chip is described.

4.2.1 AsAP2 Background

The first version of the AsAP processor was created by Baas et al. [17] and Yu et al. [18] on 180 nm technology and was utilized for DSP applications [19] [20]. The chip has 36 programmable processors and can operate at frequencies over 610 MHz at 2.0 V. Each of the 36 processors has a local oscillator on it, so that every processor operates asynchronously with respect to neighboring processors on the chip [21].

The described algorithm was implemented on the second version of the chip: AsAP2 created by Truong et al. [16]. The chip was fabricated using 65 nm technology and has 167-processors. 164 of those processors are programmable, along with three algorithm specific processors. The algorithm specific processors are: Viterbi, FFT, and video motion estimation processors [16], though none of them were used to implement the SAISort.

Each programmable processor uses RISC instructions, along with a few added instructions such as a min/max instruction that were used for the algorithm. The programmable processors were created with die size in mind and have a few limitations, though they are still within the assumptions made earlier in this thesis. Each programmable processor has an instruction memory of 128 lines of code. The on-chip memory comes in at 256 bytes (arranged as 128 16-bit words). Each processor also has two 128 byte dual clock

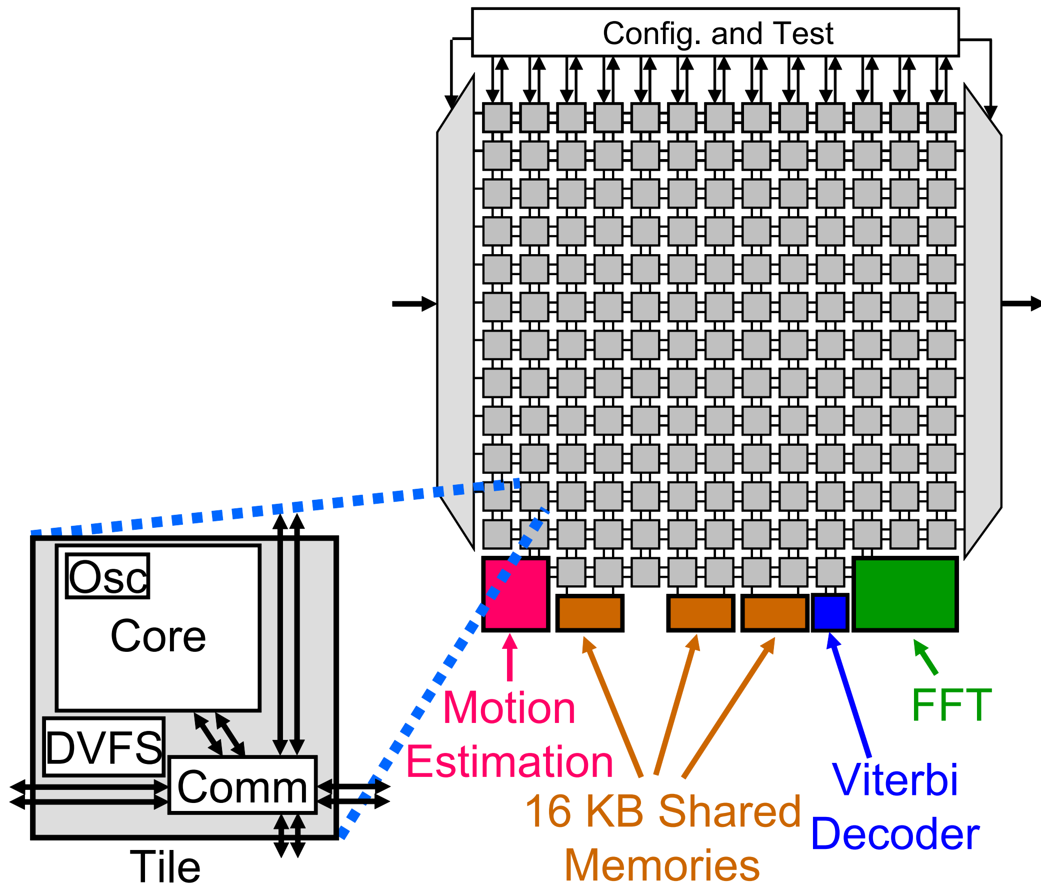

Figure 4.1: The layout of the AsAP2 chip

FIFO memories (arranged as 64 16-bit words) [22].

The AsAP2 chip's processors can each individually and dynamically change their supply voltages and clock frequencies [23]. This allows processors to alter their voltage for varying workloads across the processors. The processors also use a negligible amount of energy when they are stalling while waiting for an input, which was taken advantage of in this implementation as the workload is not distributed evenly.

There are three 16 KB on-chip shared memories on the AsAP2 chip [24] [25]. As can be seen on Figure 4.1, the three memories are at the bottom of the chip, and two processors each can communicate directly with the memories (though long-distance communication can be used from any processor to the processors that communicate with the memories). These three on-chip shared memories are be used as described in Section 4.1.1 on page 30, and their implementation and the layout to utilize them is described in Section 4.2.2

on page 37.

4.2.2 SAISort Implementation on AsAP2

Implementation at Processor Level

The algorithm mentioned earlier was created with a minimal amount of instructions to keep the process time low, and also to allow the algorithm to be implemented on systems that do not have a large amount of instruction memory. Even with the simple algorithm, address generators on the chip were taken advantage of to make sure that the entire program could fit in the instruction memory of 128 instructions. The address generator allows the processors to handle the 51 word entries (50 2-byte words with a 2-byte tag) for the key and payload attached mode by creating pointers for the key and payload separated implementation as described earlier. The programs have an initializing configuration routine that initializes all of the data for the algorithm including masks to find certain codes like the reset code.

The key and payload stored together SAISort implementation first searches to see if the key has the output command. If it does, it sends the output command to the next processor, flushes the entries currently saved in the processor, then outputs all incoming entries in order until the reset command is found in the key. When the reset command is found, the processor outputs the reset command to the next processor, then prepares itself to receive the next entry. If there is no output code, the program checks to see how many entries are already in the processor. If the processor is empty it loads the incoming entry to the higher slot. If there is only one entry already saved it calls the sorting routine to determine where the two entries should be stored in processor. If there are already two entries it moves to the main part of the sorting algorithm.

In the main part of the algorithm the keys of the incoming entry, and lowest saved entry are compared one word (2 bytes) at a time. If it is found that the incoming key is lower than the lowest stored value, it outputs the incoming entry directly to the output. If it finds that the incoming entry is higher than the lowest in the processor it outputs the lowest stored entry, then calls the sorting routine to determine which position the incoming

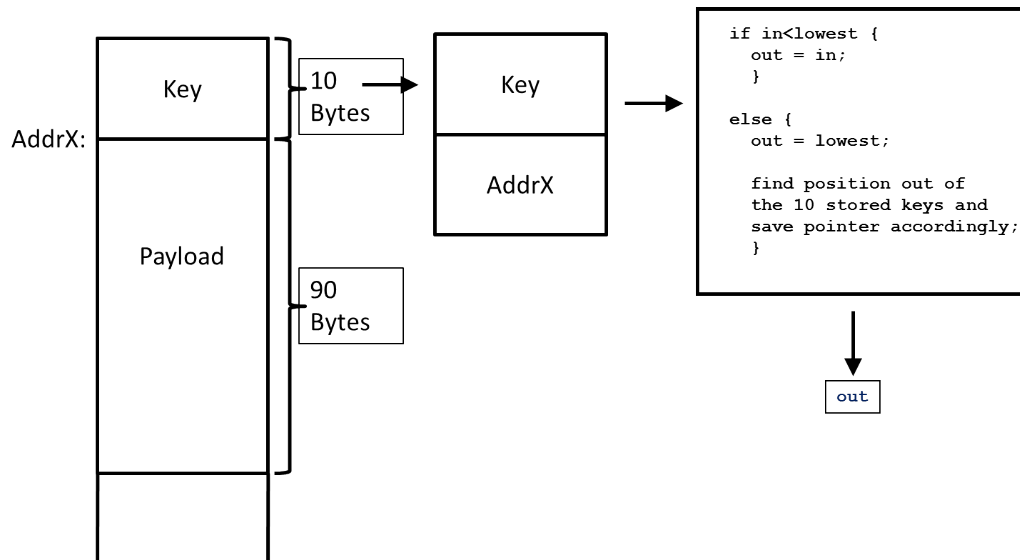

Figure 4.2: The flow of the payload seperated sort: the payload is replaced with A 10 byte address, then the key and address are sorted

and higher stored entry should be saved in. The sorting routine compares the keys of the two entries, one word at a time, storing the lower entry in the location reserved for it, and store the higher entry in its respective slot.

The largest difference for the algorithm created for sorting the keys that have been separated from their payload is that 10 entries can be stored per processor, instead of 2. With this sort, the payload of each entry is the 10 bytes that addresses the entry to its corresponding payload. The process that the entries go through is shown in Figure 4.2. Pointers are also be utilized in this implementation to reduce the number of times that the entries have to be shuffled while sorting. The program uses an output code and reset code in the same way previously described for the payload attached sort. If the output key is not found, the program checks to see how many entries are stored. If there are less than 10, the entry is saved to the nearest free spot in memory. The key is then compared to the stored entries one by one. Once the correct spot is found, the pointers that point to the saved entries are updated.

Once the processor is completely full, the next entry input only has its key saved. The key is still be checked for the output command, then the key is compared to the lowest saved entry in the processor. If the new entry is lower than the lowest saved, its key is

output, then the 10 byte payload is sent from the input to output FIFO. If the lowest saved entry is the lower, then that entry is output. The new entry's key is then saved to the newly vacated position in memory, and the 10 byte payload from the input FIFO is saved in the memory spot as well. The program then compares down the list of saved entries to find where the new entry belongs. Once its position is found, the pointers are updated.

The pointers utilized are necessary for the effective implementation of the sort. The pointers were implemented by using the AsAP2's address generators. The address generators are initialized by saving a word to a specific spot in the processor's memory that defines the starting and ending memory address for the generator. To take advantage of this, 10 words in the memory were used as the pointers. Each of these 10 memory locations had the starting and ending addresses of 10 different spots in the memory stored in them. When comparing, an address generator is used to cycle between the 10 pointers, and another address generator is used to compare the data stored where the pointers point at. In this way, the 20 bytes of each entry are kept in the same memory location, the pointers are simply changed when the order is changed.

The sort program that stores the payload in the shared memory would sort the same way the payload and key separated sort would. The only difference would be administrative processors at the input and output of the chip that would assign the pointer address to the key and save the payload to the shared memory on the AsAP2 chip, then combine the two back again after the sort was completed.

Processor Mapping

When deciding the mapping for the programs, a maximum amount of processors should be utilized, while keeping a continuous string of utilized processors from the input of the chip to the output of the chip. When utilizing the on-chip shared memory, allowances would also be required for processors to transfer the entries to the memories and a processor to merge the lists as they were output. To make the shared memory programs efficient, the number of processors that are not sorting should be kept to a minimum.

The mapping for not using the shared memory is fairly straightforward, all the processors should be utilized if possible, and the processors should be kept in a continuous

string to allow for nearest neighbor communication. After some trial and error, the mapping path shown in Figure 4.3 was decided upon. This path allows for a continuous path from beginning to end for the power efficient nearest neighbor communication while using all of the 164 programmable processors. This means that the SAISort outputs 329 entries if the payload is kept with the key, and 1641 entries if the key and payload are separated.

Mapping for utilizing the shared memory required that some of the processors be used simply for interfacing with the shared memories, and that some of the processors be used for transferring the entries to those processors. This mapping would be used for both the sort with key and payload attached and the sort where the key and payload are separated inside the chip. After working through numerous mappings, the mapping showing in Figure 4.4 was found to minimize the number of processors that would not be used for sorting. In this way, there are still 151 processors used to sort lists.

4.3 C++ Implementation

To provide a very clear comparison to the implementation on the AsAP2 chip, a C++ program was created that mimicked the phases of the of the AsAP2 chip as closely as possible. Because the second phase of the sort depends so heavily on the size of lists created in the first phase, a sort was desired that would allow the size of the sorted lists to be defined by the user. It would then be possible to make comparisons for each of the algorithms created. To make good comparisons, programs were created that kept the key and payload separate, or together. Quicksort was used for its ease of implementation and efficiency for the first phase. The second phase is a binary merge. The system that the code was run on, is a laptop with an Intel Core i7 720 processor. It also has 4 gigabytes of DDR3 RAM and a 320GB 7200RPM SATA Hard Drive.

4.3.1 Phase One / Quicksort

To create the C++ quicksort, a simple implementation was desired. An algorithm from www.24bytes.com was used as a template for the code [26]. The quicksort algorithm did indeed turn out to be fairly simple. The program loads all of the entries, then creates

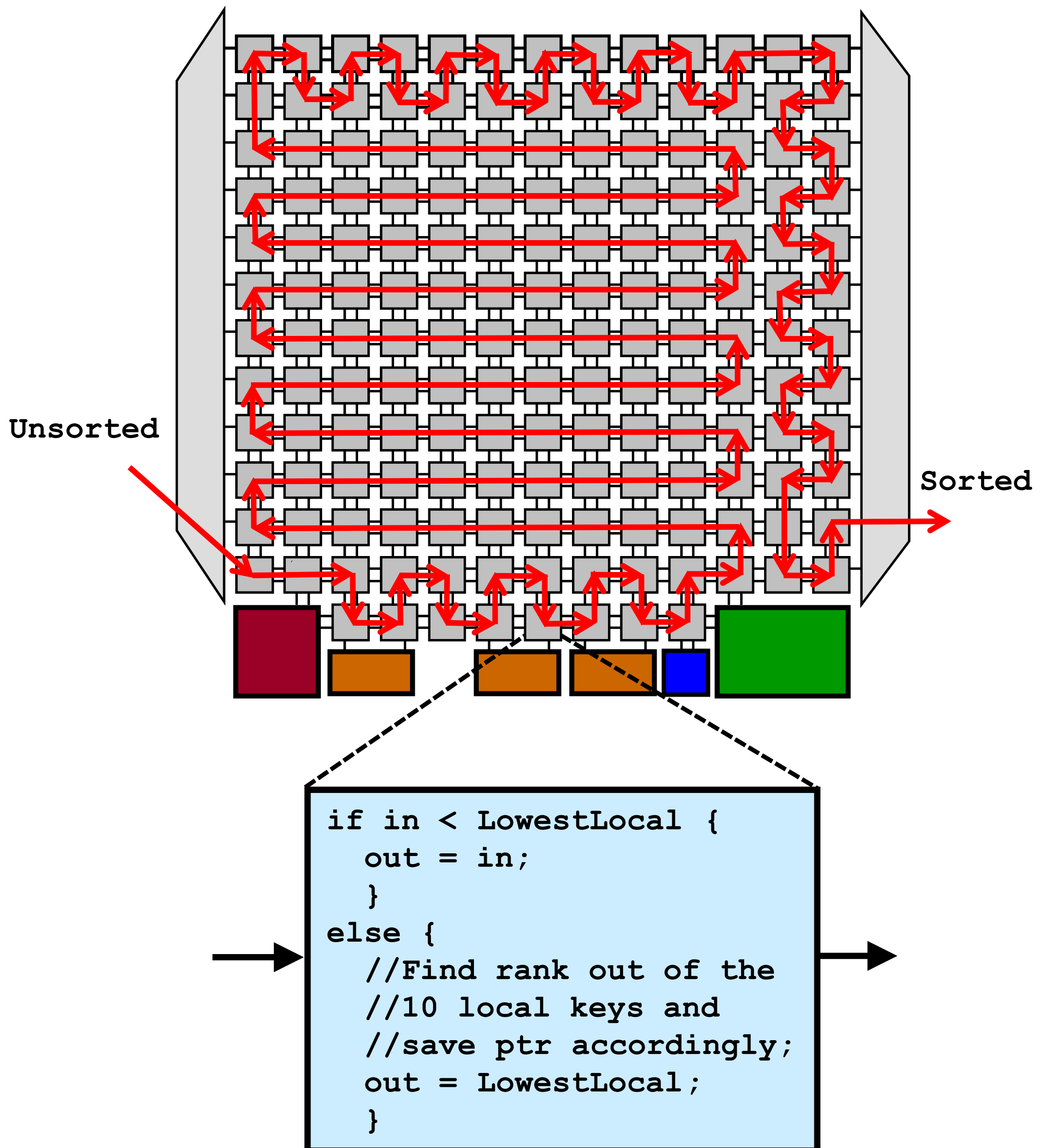

Figure 4.3: Processor mapping without using on-chip memories

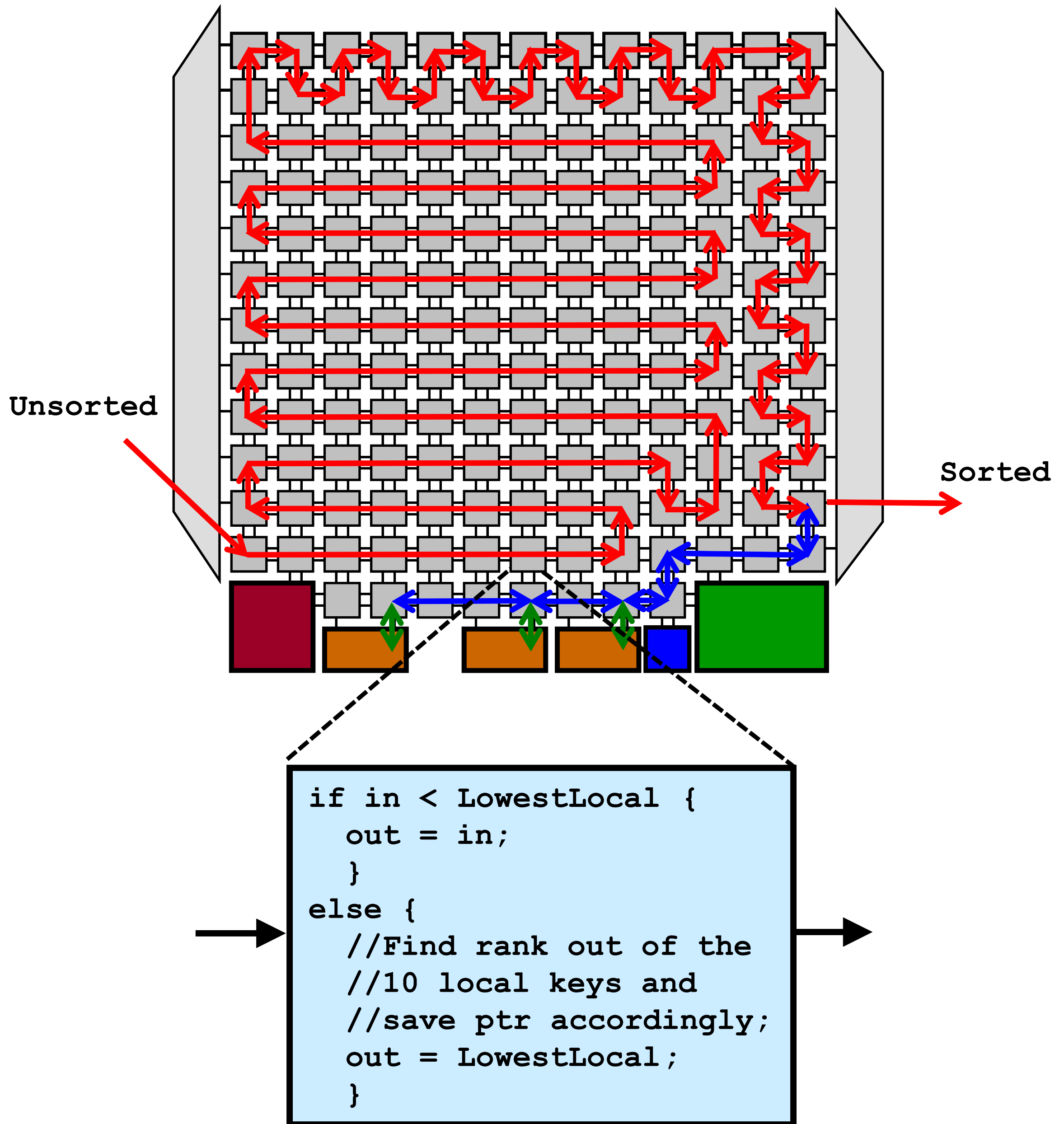

Figure 4.4: Processor mapping using on-chip memories

pointers for each entry. The program then choses the final entry of a list as its midpoint, swaps entries until there are two lists, one list above the midpoint and one below. The program then calls the quicksort routine for the two lists recursively until each quicksort routine is simply choosing the higher and lower of two entries. After a sorted list is created, the program then uses the pointers to save the entire list with payload and key into an array stored in the RAM of the computer. The user can define the size of the unsorted list, along with the size of the lists that are output.

To create a realistic benchmark for the SAISort, an efficient C++ program was required. The first version of the program swapped the entire entry, key and payload during the quicksort phase. The time for the sort was extremely high, so solutions to make the sort more efficient were explored. It was decided to use pointers so that time wouldn't be wasted moving the entire key and payload with every swap, and this drastically increased the speed of the sort. After the entries are all loaded into a string, each entry is assigned a pointer. The quicksort then uses the address stored by the pointers to compare the 10 byte keys. If an entry swap is required, the pointers are switched. To increase the speed of the sort, the i7 was utilized to its fullest extent by running 8 copies of the program on the system to use all 4 cores and 4 hyper threaded strings.

To keep the comparison relevant, this phase of the sort needed to match the AsAP2 SAISorts first phase as closely as possible, and there were many aspects to the way the sort program runs that keeps things similar. The AsAP2 program assumes that 2 GB of information would be loaded to RAM, then streamed to the processor, followed by a final merge to 10 GB. To mimic this, the entire 2GB of entries are loaded into the machine's RAM at the same time. This way the program is not held back by hard drive access times. The first phase is executed after all of the entries are loaded, creating sorted lists the size the user defined to match a case of the AsAP2 sort. The sorted lists are then left in the RAM for the second phase to use.

4.3.2 Phase Two / Binary Merge

The second phase of the sort is effectively a binary merge. Two sorted lists are saved into separate variables, and merged back into the array to create one sorted list out

of the two. The program does an entire pass through all of the entries, merging two lists to one. After this first pass, there are half as many lists that are twice the size. The program then continues to do passes through the lists, until it gets down to just one sorted list. As the focus of this work is on the first phase of the database sort, this phase two binary merge was used with the SAISort.

The second phase begins with 2 GB of many small sorted lists (depending on the number of entries sorted per list in the first phase). During this phase of the sort, pointers are not used as a binary merge does not have as many swaps as a quicksort. After the binary merge portion of the program has done enough passes, there is one 2 GB sorted list left. To complete the entire sort, the 2 GB would be saved back into the hard drive, and another 2 GB would be loaded into the RAM. This would continue until there are five lists of 2 GB in the hard drive. From there, a final merge would take place. This final merge would use the RAM to buffer as much of each of the five lists as possible, and save the final 10 GB sorted list back to the hard drive.

Chapter 5

Results and Analysis

The algorithm was created for the first phase of a database sort, so the results highlight the first phase but still show the whole picture with results for a whole system running both phases. The metric compares the C++ implementation of the algorithm to the AsAP2 implementation for the first phase. Because of the way the C++ implementation was created, the comparisons for each AsAP2 algorithm are compared to the C++ implementation for the same number of sorted lists and size of list. For completeness sake, and to show a clearer picture of how the sort compares to other database sorts, estimated JouelSort numbers of records sorted per joule are also shown. The comparison of records per joule has the same C++ merge for the second phase of both implementations as the buffer merge is not as energy efficient when looking at a complete system.

The programs created were run on an AsAP2 chip. Psuedorandom numbers were generated for the keys of the entries, then sent through the chip. The chip is not connected to a board that would allow the I/O speeds required to run the database sort in real time, so it was observed that when an unsorted list was input, a sorted list came out. Energy numbers for the SAISort were calculated by simulating the programs with NC-Verilog to find the average runtime, then energy numbers were used for the AsAP 2 chip along with the rest of the components of the system.

To find the energy consumption of the C++ program, the created program was run on a Core i7 laptop. Eight iterations of the program ran concurrently to use all of the threads

available to an i7 processor that has four cores utilizing hyper-threading. The runtime for the programs were found by using a stopwatch, averaging out the numbers of five tests. The stopwatch was started at the initialization of the first program, and stopped when the last of the eight programs was done sorting (all 8 programs always stopped within a second of each other). The energy consumption of the laptop was calculated using documented numbers.

5.1 Calculations of Energy Consumption

5.1.1 Calculation of Power Consumption

Even though the complete system of using an AsAP2 chip to sort and output to a hard drive was not physically implemented, power numbers were found for each of the components of a system that could implement the system if the AsAP2 were connected to a board with sufficient I/O speed. The JouleSort requires that all numbers be accounted for, idle power numbers were used for parts of the system that would not be active during the first phase SAISort. The added power for the on-chip shared memory is so small that it does not change the significant digits of the total, so this number is used in calculations with or without the shared memories. The numbers used are:

```
SAISort Phase One With Payload Attached:
  Core i7 in idle mode = 1.82 W
  RAM = 2 × 1.8 W = 3.6 W
  HD in idle mode = 8 × 0.85 W = 6.8 W
  AsAP2 at 68% active using all 164 processors at 1.06 GHz = 5.2 W
  On-chip shared memories 3 × 4.5 mW = 13.5 mW
  Total power consumption during sort 1.82 W 3.6 W + 6.8 W + 1.2 W + 0.0135 W
                                         = 19.6 W


SAISort Phase One With Payload And Key Separated:
  Core i7 in idle mode = 1.82 W
```

RAM = 2 × 1.8 W = 3.6 W

HD in idle mode = 8 × 0.85 W = 6.8 W

AsAP2 at 15% active using all 164 processors at 1.06 GHz = 7.3 W

On-chip shared memories = 3 × 4.5 mW = 13.5 mW

Total power consumption during sort = 1.82 W 3.6 W + 6.8 W + 7.3 W + 0.0135 W

= **19.6 W**

The algorithm is proposed to work on a chip that would be used as a co-processor, with a general purpose CPU as the main processor. While the co-processor is sorting the first phase of entries, the main processor could be utilizing the system resources for other activities. Because of this, the power used solely by the processors in the sorting algorithm are useful. Even if the co-processor takes a longer amount of time to complete its first phase sort, it still could be an effective solution if the power saved is enough.

The power used by the AsAP2 chip is very low, even with all 164 general purpose processors operating at 100% active with a 1.06 GHz clock at 1.2 V, the AsAP2 chip only takes 7.79 W of power [27]. The chip is also very power efficient with stalls, consuming a negligible amount of power because each processor can completely turn off their oscillator. The created algorithm does not operate at 100% active, as NOP operations were a necessity for the algorithm to function correctly. The algorithms also utilize an output phase that takes considerably less time than the sorting phase. The throughput is not changed because of the output phase, but it allows the processors to spend more time in a low power state. This means that the payload and key attached sort averages out to approximately 68% active for the used processors. The key and payload separated sort has more stalled time as the address generators are used extensively, which require added NOPs. The key and payload separated sort's output mode also expels all of the entries quicker than the attached sort, as there are less bytes of data in use. The average chip is active approximately 15% of the used time. This means that the payload attached and separated sorts require only 5.2 W and 1.2 W respectively.

In the key separated sort, the processors active percentage varies considerably as can be seen in Figure 5.1. The first processor being used sorts every entry that goes into the

chip, but with a random sampling of entries there is a high probability that the incoming entries keys are far apart. The sorting algorithm is only required to compare with a few saved entries before finding where it belongs in this situation, and the processor ends up being active around 7% of the time. The last processor only sorts ten entries before going into the output phase, so also ends up not being as active; around 5%. The center processors end up being used the most, because they still have a decent number of entries to sort, and the entries that make it to the center processor are all lower than the entries saved in the previous processor, so there is a higher likelihood that more saved entries are compared before the correct position is found. The center processors average around 20% active, and are the most active processors on the chip for the key separated sorting algorithm described.

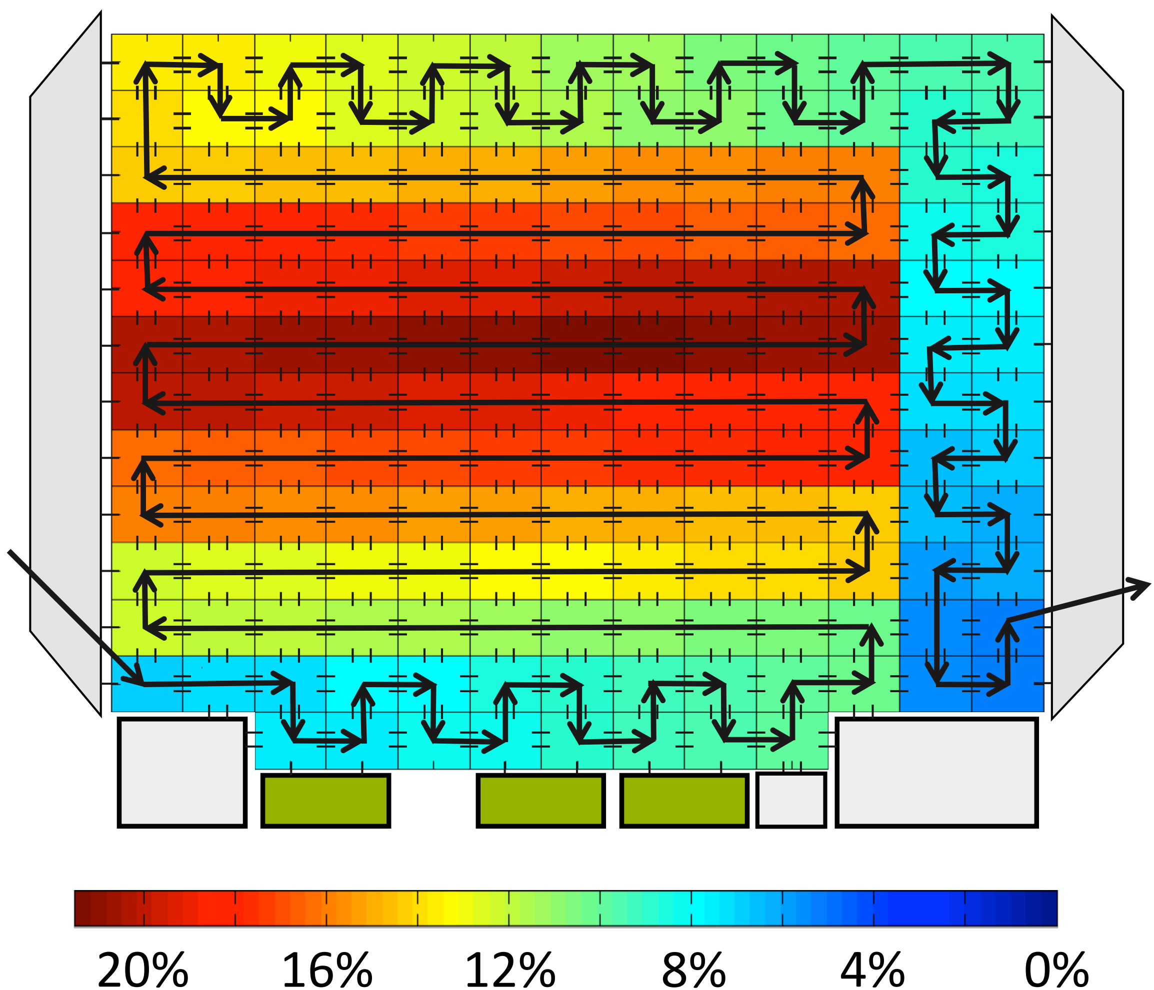

Figure 5.1: Heatmap of active percentage of each processor in the key separated sort

The C++ implementation utilized a core i7 processor. The numbers used for the C++ implementation were:

```
C++ implementation for phase one:
  Core i7 at 70% TDP = 31.5 W
  RAM = 2 × 1.8 W = 3.6 W
  HD in idle mode = 8 × 0.85 W = 6.8 W
  31.5 W + 3.6 W + 6.8 W = 41.9 W

C++ implementation for phase two:
  Core i7 at 70% TDP = 31.5 W
  RAM = 2 × 1.8 W = 3.6 W
  HD = 8 × 1.8 W = 14.4 W
  31.5 W + 3.6 W + 14.4~W = 49.5 W
```

5.2 C++ Performance Comparison to SAISort

The AsAP2 algorithm is slightly slower than the C++ implementation, though the power savings are considerable. As can be see from Table 5.1, it is slower by less than five seconds when not using the on-chip shared memory, and between 7.5 and 17 seconds for the shared memory merge implementations. Even though the sort times were only about double in the slowest case, the AsAP2 operates at approximately 1/6 of the power of the mobile i7 processor with all AsAP2 cores operating, so the total energy consumed for the AsAP2 sort is considerably less than the C++ implementation. The gap gets even larger when the AsAP2 is ran at a lower frequency, though the time to sort the first phase also increases.

In Table 5.2, the implementations with the payload and key separated are shown. In all of the situations tested, the Core i7 processor invariably achieved the shortest sort time, and the AsAP2 implementation achieved considerably lower power numbers. Because the AsAP2 implementation takes longer to output each entry than the RAM access time

	Records Sorted Per Phase One List	Time (Seconds)	Energy of Chip (Joules)	Total System Energy (Joules)
Core i7	329	11	347	461
AsAP2 @ 1.3 V No on-chip memories		15	79	261
Core i7	533	13	410	545
AsAP2 @ 1.3 V Payload stored in on-chip memories		30	**12**	379
AsAP2 @ 0.75 V Payload stored in on-chip memories		138	**3**	1,689
Core i7	783	15	473	629
AsAP2 @ 1.3 V Sorted list stored in on-chip memories		22.5	**109**	384
AsAP2 @ 0.75 V Sorted list stored in on-chip memories		103.5	**36**	1,301

Table 5.1: Time and energy to sort several quantities of unsorted records keeping the payload and key together

and transfer time, the implementation is processor limited. This is actually beneficial, as this means that if the algorithm efficiency is improved, or if the processor it is implemented can run at a higher clock frequency, the throughput would be increased. This also means that the main processor in the proposed setup could be actively using the RAM as well.

If a high throughput is not required, it can be seen that even larger energy reductions can occur. The voltage and clock frequency of each AsAP2 processor can easily be reduced. Because of the exponential relationship of power and frequency, if the chip is running at a low frequency, the power required is drastically reduced. As is observed in Tables 5.1 and 5.2, reducing the frequency to 260 MHz reduces the energy consumed by the chip is cut to almost 1/4 of the energy consumed at 1.06 GHz.

The most power efficient setup when compared to the C++ implementation is when the payload is stored into the shared memory. This low power is achieved because only 57 of the 164 processors are active to sort 533 entries, as 533 entries payloads are all that can fit into the shared memory. As mentioned earlier, the AsAP2 chip can effectively

shut off processors that are not being used, so that they take a negligible amount of energy. This allows for the significant energy reduction shown in Table 5.2, where the AsAP2 implementation takes around 1/33 of the energy the core i7 consumes.

	Records Sorted Per Phase One List	Time (Seconds)	Energy of Chip (Joules)	Total System Energy (Joules)
Core i7	1,641	12.5	394	913
AsAP2 @ 1.3 V No on-chip memories		30	**35**	405
Core i7	3,911	13.5	425	986
AsAP2 @ 1.3 V Sorted list stored in on-chip memories		77.7	**90**	1041
AsAP2 @ 0.75 V Sorted list stored in on-chip memories		357	**30**	4,398

Table 5.2: Time and energy to sort two different quantities of unsorted records keeping the payload and key separated for the initial sort

Simply having low power for the first phase does not necessarily make a good sort, because the second phase must be taken into consideration. The first phase of the sort sets up the second phase of the sort, where the number of merges required for the second phase are directly related to the size of the lists created in the first. While sorting 533 entries is considerably more efficient than the payload attached setups without on-chip memory, this is still less than half the output achieved by the payload separated setups.

The phase one algorithms that separate the key from the payload are the ones that provide the greatest efficiency to the phase two merge, and the AsAP2 implementation is more power efficient in these sorts as well. The number of records sorted can be considerably larger when the keys are separated from their payload, five times as many as the payload attached counterparts in fact. This allows the second phase merge sort to only use less passes through the data, as compared to the payload attached sorts.

The AsAP2 algorithm that sorts the key and payload separated has a lower throughput due to the added compares that take place in each processor when one 10 entries can be held, instead of two. As previously mentioned, the output mode is considerably quicker, and a large amount of the added time are in NOP instructions to allow the

compares to work correctly, so the processors are active only a fraction of the time which reduces the power consumption.

5.3 AsAP2 Implementation Performance Comparisons

To show a complete picture of how the algorithms run, it was necessary to create and run a second phase sort. The AsAP2 chip did not turn out to be the ideal platform to complete the merge function, as the chip's speed and memory limitations would not allow an efficient merge to be created that would function with large lists. The Buffer Sort algorithm mentioned earlier was completed to prove that the entire sort could be run on one chip, but it was decided that the most desirable approach would be to use the AsAP2 as a co-processor. In this way, a general purpose CPU could quickly and efficiently merge the lists that were sorted by the AsAP2 processor's phase one sort.

The merge sort was created using C++ and was run on a mobile Intel Core i7 laptop setup. The phase two sort which was created is a simple non-optimized program. After numerous tests, it was found that the C++ merge program sorts each pass through the data in approximately 7.5 seconds. As the lowest number of merges required for the second phase is 15, the second phase takes at least 112 seconds to complete with this program. This time is actually larger than the 87 seconds the JouleSort's algorithm takes to complete the entire sort. As the focus of this project is to create an efficient first phase sort, this second phase implementation is used to show how the sorts compare with each other, and is not submitted as a replacement option for current second phase sorting algorithms.

As can be seen from Table 5.3, the vast majority of energy and time for the total sort occurs during the second phase of the sort. It can also be observed from Table 5.3 that all of the attempted implementations still take less energy, and therefore have more records sorted per joule than the sort from Rivoire et al. [15].

With a more efficient phase two algorithm, the numbers will be considerably higher as the phase one algorithm created does set up the second phase well. For example, the AsAP2 algorithm that had the highest energy numbers while running at full speed was the implementation that kept the key and payload together and did not use the on-chip

Key and Payload Attached Sorts	Number of Passes Through the Data	Total Phase 2 Energy (Joules)	Total Sort Time (Seconds)	Total Sort Energy (Joules)	Total Records Sorted Per Joule
Core i7	19	7,053	154	7,514	13,309
AsAP2 @ 1.3 V No on-chip memories			158	7,314	13,672
Core i7	18	6,683	148	7,227	13,837
AsAP2 @ 1.3 V Payload stored in on-chip memories			165	7,062	14,161
AsAP2 @ 0.75 V Payload stored in on-chip memories			273	8,371	11,946
Core i7	17	6,311	143	6,940	14,410
AsAP2 @ 1.3 V Sorted list stored in on-chip memories			150	**6,695**	**14,937**
AsAP2 @ 0.75 V Sorted list stored in on-chip memories			231	7,612	13,137
JouleSort	N/A	N/A	**86.6**	8,600	11,628

Key and Payload Detached Sorts	Number of Passes Through the Data	Total Phase 2 Power (Joules)	Total Sort Time (Seconds)	Total Sort Energy (Joules)	Total Records Sorted Per Joule
Core i7	16	5,940	133	6,853	14,593
AsAP2 @ 1.3 V No on-chip memories			150	6,345	15,760
Core i7	15	5,569	126	6,554	15,257
AsAP2 @ 1.3 V Sorted keys stored in on-chip memories			162	**6,230**	**16,051**
AsAP2 @ 0.75 V Sorted keys stored in on-chip memories			342	8,386	11,925

Table 5.3: Number of passes for the second phase, time, and energy to sort varying quantities of unsorted records

memory, and this sort took 158 seconds and spent 7,314 joules of energy, with 19 passes through the data required for the seconds phase. If the entire algorithm was simply a binary merge, then 27 passes through the data would be required. This sort would take 202.5 seconds to complete while spending 10,034 joules of energy. This means that even the least efficient sort attempted saves approximately 3,700 joules of energy, compared to normal binary merge.

5.4 Hardware Variations

The AsAP2 chip was not designed with database sorting in mind. Because of this, the hardware of the chip is not necessarily optimal for the task. Looking at a few variables that could be changed for the next version of the chip, would allow an optimal setup to be discovered, and would allow for trends to be observed.

The two variables that were focused on, were varying the number of processors on-chip, and varying the quantity of on-chip shared memory. These two variables were chosen because they can vary the output of the first phase significantly. They both would also be fairly easy to implement, simply requiring more die space. As was shown in the previous section, the most efficient algorithm created for the chip was where the key and payload were separated before the first phase began. Because of this, the key and payload separated sort shall be the subject of the variations. This will yield the more interesting results, and the results would only be changed in scale with the key and payload attached sort, the general trends will remain the same for both.

For the models created, it was assumed that each processor on the chip was effectively the same as each individual chip on the AsAP2. That is to say that every chip can hold up to 10 entries, and that the most efficient method of communication is still chip to chip communication, so the power numbers for the algorithm presented in this thesis shall still be used for the calculations.

For the calculations, we are interested the outcome of the entire database sort. Because of this, the calculations look at altering the first phase of the sort, but still look at the power consumed for the second phase of the sort. The second phase being calculated

for is the C++ implementation described earlier.

5.4.1 Varying the Number of Processors

It was decided to look at how the efficiency of the sort is affected by the number of processors on the chip. For this simulation, it was assumed that there would be no on-chip shared memory utilized, only local memory that can hold 10 entries. Increasing the number of processors on the chip effectively increases by 10 the number of entries for each list in the first phase.

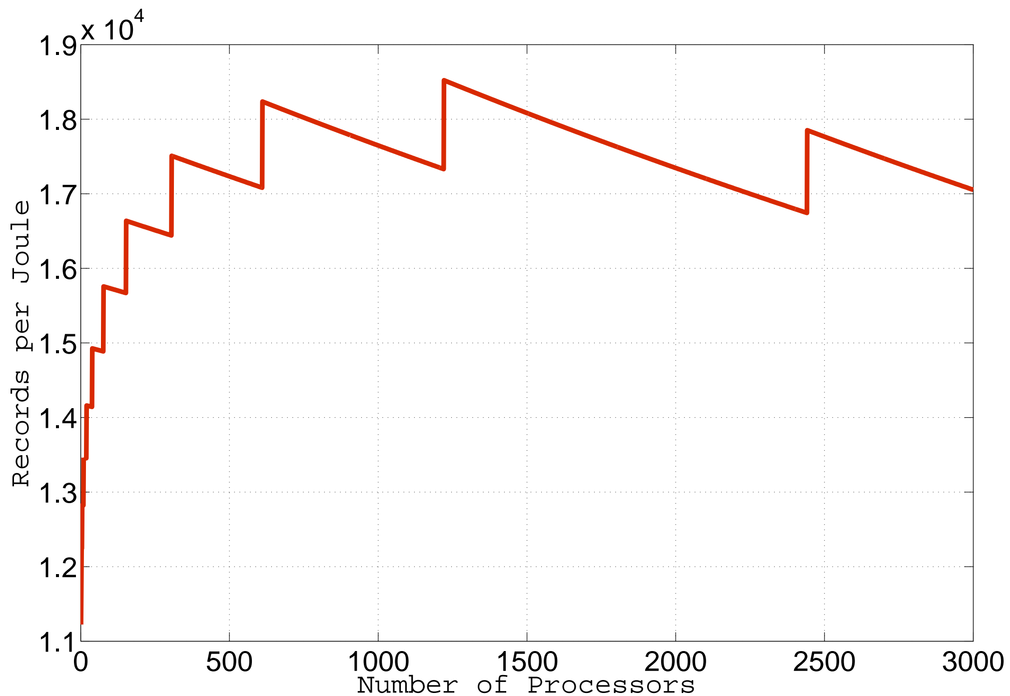

Figure 5.2: Graph of records sorted per joule with varying processors

It can be seen in Figure 5.2 that there are optimal peaks. This is caused by the nature of the second phase merge. When merging the lists into one, a number of passes through the entire data is required before one list is achieved. Varying the phase one list size without reducing the number of passes of the data in the second phase increases the energy consumed by the first phase, while not changing the energy consumed by the second phase. Because of this, the optimal points are where reducing the number of processors by

one would increase the number of passes through the data required.

As is shown in Figure 5.2, there is a clearly optimal number of processors. There is a drastic improvement of records sorted per joule when there are a small number of processors, up to about 300 processors, then the increase slows down. The most efficient number of processors is 1,221 at 18,520 records sorted per joule. Past 306, the energy required to power the first phase of the sort becomes more pronounced, so the graph slowly decreases.

5.4.2 Varying the Quantity of On-Chip Memory

The other variable being explored is how differing quantities of on-chip shared memory effects the records sorted per joule. Increasing the size of the memories increases the number of sorted entries per list in the first phase of the database sort, just like increasing the number of processors. For this simulation, the power consumption used was the same as the per byte memory usage of the AsAP2 processor. The number of processors sorted was held at a constant 164 for this simulation.

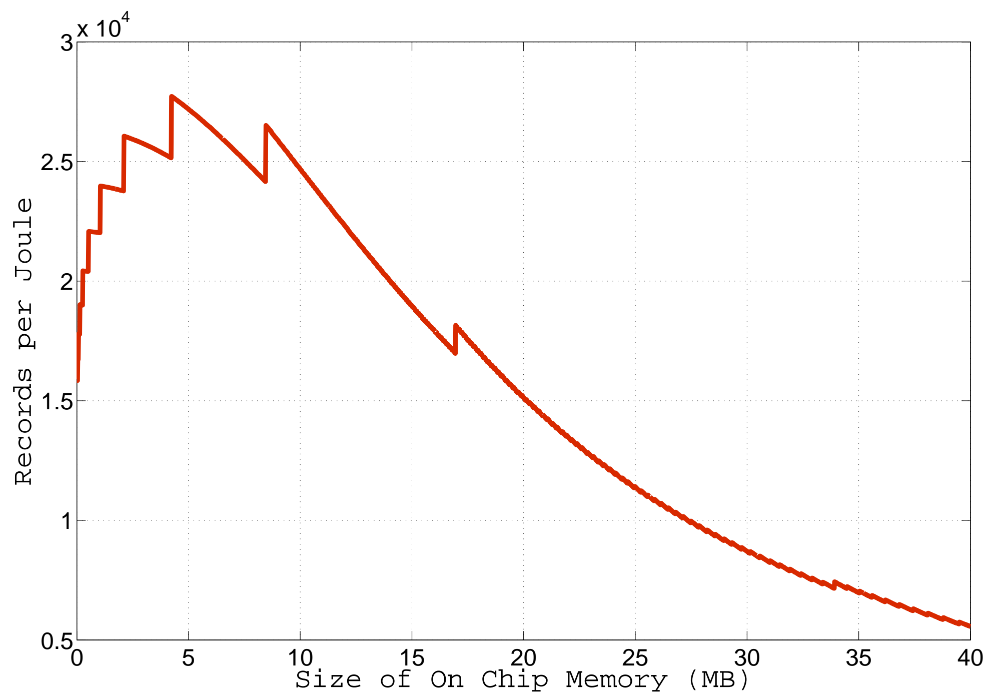

Figure 5.3: Graph of records sorted per joule with varying quantities of on-chip memory

As the graph, Figure 5.3 shows, there is the same saw-tooth pattern that was seen with the varying number of processors. The reason is still the same as well: at a certain point of increased memory, one less pass of the data is required in the second phase which causes a large jump in the efficiency.

The most efficient amount of memory also has a fairly clear maximum point of efficiency. As Figure 5.3 shows, there is a optimal point at 4.24 MB with 27,720 records sorted per joule. Because the sort is executed by sorting in chunks that would fill up the internal memory of the processors, the sorted lists are a fixed length. Adding on-chip memory would allow for multiple sorted lists to be saved, then they would be merged as they exit the chip. As it was proved earlier in this thesis that the AsAP2 processor has not been utilized efficiently to merge large lists, after only 4.24 MB of cache memory it would be more efficient to merge with the main processor of the system.

5.5 Feasibility of Implementation

The implementation suggested in this thesis would be fairly easy to implement. The AsAP2 chip could be attached to the main board of a database system, and could receive instructions and data from the general purpose CPU through the north or south bridge, as either would be able to supply the I/O speed required to the chip. A simpler implementation would also be to place the AsAP2 chip on die with the general purpose CPU.

Chapter 6

Future Work and Conclusion

6.1 Future Work

The efficiency of the phase one algorithm created shows that the AsAP2 chip is a viable option as a co-processor in a database system. The efficiency achieved also shines light on other operations that the AsAP2 might be able to efficiently execute. It would be interesting to see the an implementation for database searches and data calculations for example.

It would be also be useful to make the created sort an actual competitor for the Joule Sort benchmark. To achieve this, it would be necessary to physically implement the entire system. It would also require that an efficient second phase sort was used.

6.1.1 Searches

The general concept introduced here could be modified to allow for searches of data on a many-core system. To implement such a system, each processor on the chip could be given a specific field and data string to search for. For example processor 0,0 could be assigned to look in the "name" field for data matching "John Doe". This processor could be connected with other processors that are looking at different fields and/or data. The main processor could then simply input all of the entries to search into the system, each processor would look for entries that match their given criterion. If a match is found, the chip could either save the entry, save the entry address, or modify an attached key to show

which entry had the requested data.

If there are not enough different criterion to utilize all of the processors, multiple, identical chains could be created. Each chain would have all of the criterion, and each would have a processor at the input and output of the chip. The data could then be split into the number of chains created. It could even be taken down to each processor searching for the same criteria, with separate sections of data delivered to each. In this way, we could utilize the parallelism that a many-core chip allows for.

6.1.2 Data Calculations

The AsAP2 could also be effectively utilized to allow for data calculations on databases. Each processor on the chip could be programed with a specific field in an entry and an operation to implement on the field. Take, for example, take a user that wants to modify the pay of every employee on a database, for a cost of living increase. For this case we would program each processor to add a percentage of each employee's salary, and split up the list of employees across all of the processors.

6.1.3 Physical Implementation

Rivoire et al. [15] are very specific that to be considered for their benchmark all aspects of the system must be physically implemented so that all energy can be accounted for. Currently the AsAP2 is not connected to a board with sufficient I/O to function in a database sort. To remedy this situation, a development board that utilizes PCI express could be used. The I/O would be sufficient, and there are many good options out there that also have slots for RAM to buffer the AsAP2 chip as it sorts.

With the development board attached to a main-board through the PCIE slot, the CPU of the system and the AsAP2 to operate as co-processors as presented in this thesis. The system's CPU could be used as the main processor, and could complete the second phase of the sort. This would allow the system to run a normal operating system, which would make connecting the system to multiple hard drives very simple.

6.1.4 Efficient Second Phase Sort

This thesis focused on an efficient first phase sort for a database. The efficiency in comparison to a normal CPU was shown, which makes clear the possibility of a database sort being executed with a low power many-core system. To get a complete picture though, and to make the algorithm a competitor for the Joule Sort benchmark, it would be useful to have an efficient second phase sort, as the C++ program created and presented in this thesis was shown to not equal the efficiency of common database sorts. As there are plenty of sorts that operate quicker than the C++ code created and presented here, it should not be a large problem to find or create a second phase sort that would could be used with the first phase sort presented here.

6.2 Conclusion

It has been shown that a low power, relatively low clock frequency many-core chip can be effectively used to help create an efficient database sort. The energy consumption of the phase one sort presented here is considerably lower than that of a C++ program that sorted the same sized lists. While the time to sort for the presented algorithm was slightly longer than the C++ counterpart, the time is still reasonable. It is even more reasonable when looked in conjunction that the many-core system is proposed as a co-processor. This allows the main processor of the system to utilize system resources as the many-core processor is computing.

There is room for improvement of the sort, though this presented work lays the groundwork for other interesting future work. Because this many-core system was proven a viable option, new paths of research into other ways to exploit the parallel nature of many-core systems can be explored. It was also shown, that even with an un-optimized C++ merge program, the proposed algorithm was able to post decent energy consumption numbers.

Bibliography

[1] Ratnesh K. Sharma, Rocky Shih, Cullen Bash, Chandrakant Patel, Philip Varghese, Mohandas Mekanapurath, Sankaragopal Velayudhan, and Manu Kumar, V. On building next generation data centers: energy flow in the information technology stack. In *Proceedings of the 1st Bangalore Annual Compute Conference*, COMPUTE '08, pages 8:1–8:7, New York, NY, USA, 2008. ACM.

[2] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM.

[3] J. Melton and A. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufman, 1993.

[4] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25:73–169, June 1993.

[5] David Taniar and J. Wenny Rahayu. Sorting in parallel database systems. In *Proceedings of the The Fourth International Conference on High-Performance Computing in the Asia-Pacific Region-Volume 2 - Volume 2*, HPC '00, pages 830–, Washington, DC, USA, 2000. IEEE Computer Society.

[6] EPA. EPA report to congress on server and data center energy efficiency. Technical report, U.S. Environmental Protection Agency, 2007.

[7] Luiz André Barroso. The price of performance. *Queue*, 3:48–53, September 2005.

[8] Invest in Finland. Hp's finnish data center reaches high level of energy efficiency, 2008. This is an electronic document. Date of publication: October 29, 2008. Date retrieved: January 25, 2010. Date last modified: October 29, 2008.

[9] Google. Hamina data center, 2011. This is an electronic document. Date of publication: [Date unavailable]. Date retrieved: January 25, 2010. Date last modified: January 11, 2011.

[10] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2005.

[11] Dina Bitton, David J. DeWitt, David K. Hsaio, and Jaishankar Menon. A taxonomy of parallel sorting. *ACM Comput. Surv.*, 16:287–318, September 1984.

[12] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[13] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4:321–, July 1961.

[14] David Taniar and J. Wenny Rahayu. Parallel database sorting. *Inf. Sci. Appl.*, 146:171–219, October 2002.

[15] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD Conference*, pages 365–376. ACM, 2007.

[16] D. Truong, W. Cheng, T. Mohsenin, Zhiyi Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, Anh Tran, J. Webb, E. Work, Zhibin Xiao, and B. Baas. A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. In *VLSI Circuits, 2008 IEEE Symposium on*, Jun. 2008.

[17] B. Baas, Zhiyi Yu, M. Meeuwsen, O. Sattari, R. Apperson, E. Work, J. Webb, M. Lai, T. Mohsenin, D. Truong, and J. Cheung. AsAP: A fine-grained many-core platform for DSP applications. *Micro, IEEE*, 27(2):34–45, Mar. 2007.

[18] Zhiyi Yu, M.J. Meeuwsen, R.W. Apperson, O. Sattari, M. Lai, J.W. Webb, E.W. Work, D. Truong, T. Mohsenin, and B.M. Baas. AsAP: An asynchronous array of simple processors. *Solid-State Circuits, IEEE Journal of*, 43(3):695–705, Mar. 2008.

[19] Zhiyi Yu. *High Performance and Energy Efficient Multi-core Systems for DSP Applications*. PhD thesis, University of California, Davis, CA, USA, October 2007. http://www.ece.ucdavis.edu/vcl/pubs/theses/2007-5.

[20] Ryan Apperson Omar Sattari Michael Lai Jeremy Webb Eric Work Tinoosh Mohsenin Bevan Baas Zhiyi Yu, Michael Meeuwsen. Architecture and evaluation of an asynchronous array of simple processors. *Journal of VLSI Signal Processing Systems*, 53(3):243.

[21] Wayne H. Cheng. Approaches and designs of dynamic voltage and frequency scaling. Master's thesis, University of California, Davis, CA, USA, January 2008. http://www.ece.ucdavis.edu/vcl/pubs/theses/2008-1.

[22] Ryan Apperson, Zhiyi Yu, Michael Meeuwsen, Tinoosh Mohsenin, and Bevan Baas. A scalable dual-clock FIFO for data transfers between arbitrary and haltable clock domains. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 15(10):1125–1134, October 2007.

[23] Wayne H. Cheng and Bevan M. Baas. Dynamic voltage and frequency scaling circuits with two supply voltages. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1236–1239, May 2008.

[24] Michael J. Meeuwsen, Zhiyi Yu, and Bevan M. Baas. A shared memory module for asynchronous arrays of processors. *EURASIP J. Embedded Syst.*, 2007:21–21, January 2007.

[25] Michael J. Meeuwsen. A shared memory module for an asynchronous array of simple processors. Master's thesis, University of California, Davis, CA, USA, April 2005.

[26] 24bytes.com. Quick sort, 2010. This is an electronic document. Date of publication: June, 2010. Date retrieved: July 20, 2010. Date last modified: June, 2010.

[27] D.N. Truong, W.H. Cheng, T. Mohsenin, Zhiyi Yu, A.T. Jacobson, G. Landge, M.J. Meeuwsen, C. Watnik, A.T. Tran, Zhibin Xiao, E.W. Work, J.W. Webb, P.V. Mejia, and B.M. Baas. A 167-processor computational platform in 65 nm CMOS. *Solid-State Circuits, IEEE Journal of*, 44(4):1130–1144, Apr. 2009.