Dynamic Energy-Aware Database Storage and Operations

Peyman Behzadnia
University of South Florida, peyman.behzadnia@gmail.com

Follow this and additional works at: https://digitalcommons.usf.edu/etd

Part of the Computer Sciences Commons

Scholar Commons Citation

This Dissertation is brought to you for free and open access by the USF Graduate Theses and Dissertations at Digital Commons @ University of South Florida. It has been accepted for inclusion in USF Tampa Graduate Theses and Dissertations by an authorized administrator of Digital Commons @ University of South Florida. For more information, please contact digitalcommons@usf.edu.
Dynamic Energy-Aware Database Storage and Operations

by

Peyman Behzadnia

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Yi-Cheng Tu, Ph.D.
Ken Christensen, Ph.D.
Adriana Iamnitchi, Ph.D.
Bo Zeng, Ph.D.
Ming Ji, Ph.D.

Date of Approval:
March 26, 2018

Keywords: Database Management System (DBMS), Dynamic Power Management (DPM), Data Consolidation, Data Stream Management System (DSMS), Stream (Window) Join

Copyright © 2018, Peyman Behzadnia
Dedication

Dedicated to my beloved mother, father, brother, sister, nephew, niece, and my friend, Marjan.
Acknowledgments

I would like to deeply appreciate my advisor, Dr. Yi-Cheng Tu, for his academic advice and inspirations during my Ph.D. study. He is an excellent researcher, inspiring advisor and a perfect mentor that every graduate student would like to work with.

I would like to also thank my committee members Dr. Ken Christensen, Dr. Adriana Iamnitchi, Dr. Bo Zeng, and Dr. Ming Ji for their valuable advice and feedback in my research.
# Table of Contents

List of Tables ................................................................................................................................. iii

List of Figures ................................................................................................................................ iv

Abstract ......................................................................................................................................... vii

Chapter 1 Introduction .................................................................................................................... 1  
  1.1. Background and Motivation ....................................................................................... 1  
  1.2. Research Objectives ..................................................................................................... 6  
  1.3. Research Contributions ............................................................................................... 8  
  1.4. Outline.......................................................................................................................... 9

Chapter 2 Literature Survey .......................................................................................................... 11  
  2.1. DPM Techniques in Disk Storage Systems .............................................................. 11  
    2.1.1. Data Consolidation Techniques .................................................................. 12  
    2.1.2. Power-Aware Cache and Prefetching Algorithms ...................................... 16  
    2.1.3. Energy-Proportional Strategies in Distributed Systems ............................. 18  
    2.1.4. Other Techniques ........................................................................................ 20  
  2.2. Stream Joins ............................................................................................................... 21  
    2.2.1. Join Algorithms on Sliding Windows......................................................... 21  
    2.2.2. Shared-Memory Parallel Stream Joins ........................................................ 23  
      2.2.2.1. Handshake Join .......................................................................... 23  
      2.2.2.2. ScaleJoin .................................................................................... 26  
    2.2.3. Multi-Way Stream Join ............................................................................... 28  
  2.3. Database Joins ............................................................................................................ 30  
    2.3.1. Hash Joins ................................................................................................. 30  
      2.3.1.1. Canonical Hash Join ................................................................. 31  
      2.3.1.2. No Partitioning Join ................................................................. 31  
      2.3.1.3. Radix Join .............................................................................. 32  
      2.3.1.4. Parallel Radix Join ................................................................. 34  
    2.3.2. Sort-Merge Joins .......................................................................................... 35  
      2.3.2.1. Classical Sort-Merge Join ......................................................... 36  
      2.3.2.2. Parallelization of Sorting Phase using SIMD ......................... 37  
      2.3.2.3. Parallel Sort-Merge Join Algorithms ....................................... 41

Chapter 3 Dynamic Energy-Aware Disk Storage Management in Database Servers .......... 44  
  3.1. Proposed DPM Optimization Model ......................................................................... 44  
    3.1.1. Formulation of DPM Optimization Model for Multi-State Disks .......... 48  
    3.1.2. Two-State Optimization Model ................................................................. 49
List of Tables

Table 2.1  Characteristics of power-saving solutions using data consolidation techniques ..............................................................................................................15

Table 3.1  DPM optimization model parameters..........................................................................................................................46

Table 3.2  DPM optimization model decision variables..................................................................................................................47

Table 3.3  Hard disk specifications ..............................................................................................................................................59

Table 4.1  Intel Xeon Phi Coprocessor 5110P specifications ........................................................................................................88
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Power modes and their power consumption for the IBM UltraStar 7k6000</td>
<td>2</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Sliding-window join for two data streams</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Taxonomy of effective DPM techniques in disk storage systems</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Symmetric hash join algorithm for two data streams</td>
<td>23</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Overview of ScaleJoin’s architecture and parallelization approach [GN15]</td>
<td>26</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>ScaleJoin &amp; Handshake join throughput on two different systems [GN15]</td>
<td>27</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Multi-way join</td>
<td>28</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Canonical hash join</td>
<td>30</td>
</tr>
<tr>
<td>Figure 2.7</td>
<td>No partitioning join</td>
<td>31</td>
</tr>
<tr>
<td>Figure 2.8</td>
<td>Partitioned hash join</td>
<td>32</td>
</tr>
<tr>
<td>Figure 2.9</td>
<td>Radix join</td>
<td>33</td>
</tr>
<tr>
<td>Figure 2.10</td>
<td>Classical sort-merge join</td>
<td>35</td>
</tr>
<tr>
<td>Figure 2.11</td>
<td>Sorting network</td>
<td>37</td>
</tr>
<tr>
<td>Figure 2.12</td>
<td>Comparator implementation via min/max operators</td>
<td>37</td>
</tr>
<tr>
<td>Figure 2.13</td>
<td>Bitonic merge network</td>
<td>38</td>
</tr>
<tr>
<td>Figure 2.14</td>
<td>Multiple two-way merging</td>
<td>39</td>
</tr>
<tr>
<td>Figure 2.15</td>
<td>NUMA-aware sort-merge join with multi-way merge and SIMD</td>
<td>42</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>9-PWL function of response time model</td>
<td>50</td>
</tr>
</tbody>
</table>
Figure 3.2 Flowchart for one-to-one sequential pairing algorithm .........................................55
Figure 3.3 Flowchart for one-to-many sequential pairing algorithm ..................................56
Figure 3.4 Examples of gamma distributions with different parameters used in dynamic workloads ..................................................................................................................60
Figure 3.5 Experimental results under dynamic workloads with different mean data arrival rates ..................................................................................................................62
Figure 3.6 Average response time results under dynamic workloads with different mean data arrival rates ..........................................................................................64
Figure 3.7 Experimental results under static workloads with different data arrival rates ..........................................................................................66
Figure 3.8 Experimental results for large-scale storage system under dynamic workloads with different mean data arrival rates ......................................................................67
Figure 3.9 Average computational time results for large-scale storage system under dynamic workloads with different mean data arrival rates ..................................68
Figure 3.10 MPC robustness results under different prediction error rates – set 1 ...............69
Figure 3.11 MPC robustness results under different prediction error rates – set 2 ...............71
Figure 3.12 Effect of epoch length on energy saving ............................................................73
Figure 4.1 Chunk-based pairing hash join algorithm overview ...........................................81
Figure 4.2 Scalability evaluation under different window sizes and increasing number of threads ..................................................................................................................89
Figure 4.3 Comparisons in terms of scalability and throughput ..........................................90
Figure 4.4 Scalability evaluation for parallel multi-way hash join under different window sizes and increasing number of threads ..........................................................92
Figure 4.5 Latency evaluation of parallel schemas ..............................................................93
Figure 4.6 Latency evaluation of parallel schemas under different intensities of input streams ..................................................................................................................94
Figure 4.7 Latency evaluation of parallel multi-way hash join algorithm ..............................94
Figure 4.8  Energy consumption measurements for parallel stream join algorithms under increasing number of threads (cores)...........................................................96
Abstract

Energy consumption has become a first-class optimization goal in design and implementation of data-intensive computing systems. This is particularly true in the design of database management systems (DBMS), which is one of the most important servers in software stack of modern data centers. Data storage system is one of the essential components of database and has been under many research efforts aiming at reducing its energy consumption. In previous work, dynamic power management (DPM) techniques that make real-time decisions to transition the disks to low-power modes are normally used to save energy in storage systems. In this research, we tackle the limitations of DPM proposals in previous contributions and design a dynamic energy-aware disk storage system in database servers. We introduce a DPM optimization model integrated with model predictive control (MPC) strategy to minimize power consumption of the disk-based storage system while satisfying given performance requirements. It dynamically determines the state of disks and plans for inter-disk data fragment migration to achieve desirable balance between power consumption and query response time. Furthermore, via analyzing our optimization model to identify structural properties of optimal solutions, a fast-solution heuristic DPM algorithm is proposed that can be integrated in large-scale disk storage systems, where finding the most optimal solution might be long, to achieve near-optimal power saving solution within short periods of computational time. The proposed ideas are evaluated through running simulations using extensive set of synthetic workloads. The results show that our solutions achieve
up to 1.65 times more energy savings while providing up to 1.67 times shorter response time compared to the best existing algorithm in literature.

Stream join is a dynamic and expensive database operation that performs join operation in real-time fashion on continuous data streams. Stream joins, also known as window joins, impose high computational time and potentially higher energy consumption compared to other database operations, and thus we also tackle energy-efficiency of stream join processing in this research. Given that there is a strong linear correlation between energy-efficiency and performance of in-memory parallel join algorithms in database servers, we study parallelization of stream join algorithms on multicore processors to achieve energy efficiency and high performance. Equi-join is the most frequent type of join in query workloads and symmetric hash join (SHJ) algorithm is the most effective algorithm to evaluate equi-joins in data streams. To best of our knowledge, we are the first to propose a shared-memory parallel symmetric hash join algorithm on multi-core CPUs. Furthermore, we introduce a novel parallel hash-based stream join algorithm called chunk-based pairing hash join that aims at elevating data throughput and scalability. We also tackle parallel processing of multi-way stream joins where there are more than two input data streams involved in the join operation. To best of our knowledge, we are also the first to propose an in-memory parallel multi-way hash-based stream join on multicore processors. Experimental evaluation on our proposed parallel algorithms demonstrates high throughput, significant scalability, and low latency while reducing the energy consumption. Our parallel symmetric hash join and chunk-based pairing hash join algorithms achieve up to 11 times and 12.5 times more throughput, respectively, compared to that of state-of-the-art parallel stream join algorithm. Also, these two algorithms provide up to around 22 times and 24.5 times more throughput, respectively,
compared to that of non-parallel (sequential) stream join computation where there is one processing thread.
Chapter 1
Introduction

In this section, required background and motivation behind this research are provided. Next, research objectives are presented. Then, major contributions of this research are discussed followed by the dissertation outline.

1.1. Background and Motivation

Data centers consume massive and growing amount of energy. A report shows that, in 2014, data centers in the United States consumed an estimated 70 billion kilowatt-hours (kWh), representing about 1.8% of total U.S. electricity consumption [SH16]. Database management system (DBMS) is one of the most important servers in the software stack of modern data centers. Data storage system is one of the key elements of database and has been under many research efforts aiming at reducing its energy consumption. Storage systems used to consume a significant portion of energy in data centers and online transaction processing (OLTP) systems [GS03, ZC05]. This trend has been changed for storage systems in recent years towards smaller fraction of the total energy consumption in data centers. However, according to a report in 2016 [SH16], storage systems consumed around 10% of the total energy consumption in data centers in 2014, which is equivalent to around 6.7 billion kWh. This shows that power saving solutions in data storage systems can still be beneficial and result in remarkable energy saving in electricity bills of data centers. This motivates this research to tackle the problem of designing a power-aware disk storage system.
Note that the use of SSD drives simplifies the problem since they are highly energy efficient compared to HDDs, but, as of today, SSDs are still not in a position to replace all magnetic disks in large-scale storage systems as the primary storage medium, especially those handling today’s big data applications, since hard drives have advantages in terms of cost and capacity compared to SSDs [NT09, AG11]. Thus, the focus of this research is on traditional hard drives and trade-offs between performance and energy efficiency.

Making storage systems green has been addressed in many research efforts in the literature. In previous work, dynamic power management (DPM) techniques are normally used to save energy in disk storage systems. Such algorithms make real-time decisions on when to transition magnetic disks to lower-power modes with the price of longer response time to data access requests.

Many modern hard disks have two power states: active and stand-by. Disks in stand-by mode stop rotation completely thus consume significantly less energy than in active state. However, it incurs a considerable cost in response time and energy to spin up the disk to active mode in order to serve a request. Figure 1.1 shows the detailed specifications related to the power and transition time among different states of a typical multi-mode disk (model Ultra-star 7k6000 from IBM) [HG15].
Given the aforementioned penalty cost in response time and energy related to disk state transition, traditional DPM methods provide either little energy savings or suffer from significant performance degradation. More effective DPM techniques attempted to improve this limitation by extending the idle period of disks by either controlling the I/O intervals [ZZ05], [PS04], [LW04], [YX06], [ZD04], [ZS04], [WY08], [PG11] or consolidating data on subset of disk [PB04], [CG02], [WO07], [VK10], [OR10]. The first set of techniques usually considers single-disk systems and utilizes energy-efficient caching or pre-fetching techniques to prolong the idle periods in the I/O workload. The second set of methods basically consolidates the most frequently accessed data (called “hot” data in literature) on subset of disks to allow “cold” disks sleep longer. They usually perform corresponding inter-disk data migration in order to achieve the hot data consolidation goal.

As the major limitation, work of this type cannot effectively enough handle the dynamic workloads where arrival rate of data requests changes significantly with respect to time. Furthermore, they do not usually provide efficient disk state configuration or inter-disk data migration plans. The best known algorithm in the literature that tries to handle dynamic environment is named dynamic block exchange (BLEX) presented in [OR10]. However, we believe BLEX, again, does not strongly enough adapt to dynamicity in the workload since it performs insufficient inter-disk data migration. In this research, we tackle the limitation of the previous work.

Stream join is a dynamic and expensive database operation with many applications such as object tracking [HA03], video correlation [GW06], news item matching [GW07]. Stream join performs the join operation in real-time fashion on dynamic and continuous data streams [AB03, CC03, SA05, GB09]. The join operation is performed only on the most recent portion of each data stream, referred to as sliding windows. Figure 1.2 (adopted from [TM11] ) illustrates the sliding-
window join over two streams $R$ and $S$. Sliding windows are time-based or tuple-based. The former covers all tuples within the last $t$ seconds while the latter covers the last $n$ tuples in arrival order. Window slides forward as time proceeds or new tuples arrive respectively.

Stream joins have high computational time (such cost will be quantitatively shown in section 3.1 [GN15]) and potentially consumes more energy compared to other database operations. Thus, we also study energy-efficiency of stream join algorithms in this research. According to the state-of-the-art work that analyzed energy-efficiency in database servers [TH10], the most energy-efficient join algorithms are the highest performing ones. In particular, this work revealed that there is a strong linear correlation between energy-efficiency and performance of in-memory parallel joins as the number of cores grows in database servers. As a result, in this research, we tackle parallelization of stream join algorithms on multicore processors to achieve energy efficiency and high performance. In the literature, both shared-nothing [GJ12], [AB13] and shared-memory [GB09, TM11, RT14, GN15] parallelization techniques have been proposed for stream join operation. The former targets at scaling out in multi-node systems while the latter aims at scaling up the performance and throughput within an individual multi-core node. We focus on shared-memory parallelization techniques in this research. Among the stream joins, equi-join is the most frequent type of join existing in query workloads and symmetric hash join (SHJ) is a
popular hashing-based join algorithm that most effectively evaluates equi-based stream joins [XY07]. There are some research works in literature that proposed parallel nested loops-style stream joins on multi-core processors. However, the nested loops-based implementation of stream join, known as three-step procedure in literature (section 2.2.1), is not an efficient algorithm to execute equi-based joins. The reason is that it first needs to enumerate all combination of input tuples in a nested loop manner in the scan phase (first step) to check for the join condition. On the other hand, symmetric hash join algorithm provides hash tables as fast in-memory access structures that significantly reduce number of pairs to enumerate. As a result, symmetric hash join, as a hashed-based stream join, is known as the most effective and suitable join algorithm for evaluating equi-based joins on sliding windows. We are aware of no work on shared-memory parallel hash-based stream join processing on multicore processors. This motivates our research to devise a solution for this demanding problem.

Multi-way stream join is the major extension to the original binary stream join processing where there are more than two input data streams involved in the join evaluation. It is intuitive that the computational cost of multi-way stream joins is even higher than that of two-way stream joins since for each input tuple, there is more than one stream involved in its probe phase to find matching tuples. There is a multi-way hash join algorithm in literature, called eager multi-way hash join, explicitly designed for sliding windows. However, we are aware of no previous work in the literature on shared-memory parallel multi-way window joins, neither nested-loops style nor hashing-based style. This also motivates our research to design a shared-memory parallel multi-way hashing-based stream join on multicore processors.
1.2. Research Objectives

In this section, the major objectives are discussed to be achieved by this research. We aim at designing a dynamic energy-aware database disk storage. As mentioned earlier, the major limitation of previous DPM algorithms in literature is that they cannot efficiently enough handle the dynamic workloads where arrival rate of data requests changes significantly with respect to time. The best known algorithm in the literature that tries to handle dynamic environment is named dynamic block exchange (BLEX) presented in [OR10]. However, we believe BLEX, again, does not strongly enough adapt to dynamicity in the workload since it performs insufficient inter-disk data migration. As a major objective, this research aims at solving this issue by introducing an optimization model that integrates model predictive control (MPC) strategy to accommodate dynamic scenarios by enabling optimization actions in an online fashion. In particular, the current control action is obtained dynamically where, at each sampling instant, a finite horizon optimization problem is solved, and its optimal solution is applied as the current control decision. Such procedure repeats along the whole control process.

Furthermore, as a major goal, this research aims at designing a heuristic DPM algorithm that can be integrated to large-scale disk storage systems, where finding the optimal solution might be long, in order to provide fast and efficient power saving solutions. Also, we will conduct extensive set of experiments to empirically evaluate the effectiveness of the ideas in this research and compare them side by side with the state-of-the-art work in literature, BLEX algorithm.

As discussed in previous section, stream join is a dynamic and expensive database operation and the most energy-efficient stream join algorithms are the highest performing ones. Thus, we aim at reducing its high energy and computational costs through parallel design of stream join algorithms on multicore processors. Note that the goal of this research on parallel processing
of stream join algorithms is two-fold. The first goal, as the major claim of this research, is high throughput and scalability in performing stream join operation. The second goal is to also achieve the best energy-efficiency for systems running the stream joins. There is no work in literature on shared-memory parallel hash-based stream join. Thus, as the first research work, we aim at proposing a parallel symmetric hash join algorithm on multicore processors to achieve high performance in equi-based stream join execution while reducing the energy consumption. Another main objective of this research in this area is to introduce a novel parallel algorithm to further elevate data throughput and reduce the potential overhead related to inter-thread synchronization – required for concurrent access to shared data structures.

As mentioned earlier, this research also tackles parallel processing of multi-way equi-based stream join where there are more than two input data streams involved in the equi-join evaluation. It is intuitive that the cost of multi-way stream joins is even higher than that of two-way stream joins since for each input tuple, there is more than one stream involved in its probe phase to find matching tuples. are aware of no previous work in literature on parallel multi-way window joins. Thus, as a major objective, this research would be also the first to propose an in-memory parallel multi-way hash join algorithm for high-performance execution of multi-way equi-based window joins on multi-core processors. Also, we integrate the following desired parallelism properties in our parallel designs: high throughput, significant scalability, low latency, disjoin parallel, architecture independent, and hardware-specific optimization independent. We also conduct extensive set of experiments to empirically evaluate our parallel ideas in this research. We compare our two-way parallel algorithms with the state-of-the-art work in literature, ScaleJoin, in terms of throughput, latency, and energy efficiency.
1.3. Research Contributions

In this section, we summarize the major contributions that our proposed research provides to its field of science. Note that our contributions on stream join parallel processing is two-fold. The first contribution, as the main claim of this research, is to achieve high performance with low latency in performing stream joins. The second contribution is to also achieve the best energy-efficiency for the systems running stream joins. In summary, this research makes the following contributions:

1) A DPM optimization model integrated with model predictive control (MPC) strategy is introduced to minimize power consumption of the disk-based storage system while satisfying given performance requirements. It dynamically determines the state of disks and plans for inter-disk data fragment migration to achieve desirable balance between power consumption and query response time.

2) Via analyzing the optimization model to identify structural properties of optimal solution, a fast-solution heuristic DPM algorithm is proposed that dynamically determines efficient disk state configuration and inter-disk data migration. It can be integrated to large-scale disk storage systems holding today’s big data, where finding the most optimal solution might be long, in order to achieve near optimal power saving solution (near to that of the optimization model) within short periods of computational time.

3) Experimental simulations using extensive set of synthetic workloads are conducted to evaluate power-saving solutions proposed in this research. Experimental results are compared in terms of both power saving and response time with those of the best existing dynamic algorithm named dynamic block exchange algorithm (BLEX). Our
results clearly demonstrate remarkable energy savings while satisfying the given performance bound. The MPC-based optimization model and the heuristic DPM algorithm outperform the BLEX algorithm significantly in terms of both energy savings and response time in data access.

4) We are the first to propose shared-memory parallel symmetric hash join algorithm on multi-core processors that achieves high throughput and scalability in performing equi-based stream joins.

5) This research introduces a novel parallel hash-based stream join algorithm called \textit{chunk-based pairing hash join} that significantly elevates data throughput and scalability.

6) The proposed research is the first to propose an in-memory parallel hash join algorithm for multi-way stream environments to achieve high-performance execution of multi-way equi-based window joins.

7) Our high-performance parallel stream join algorithms achieve the best energy-efficiency for the systems running the stream joins. Also, our parallel ideas provide the following desired parallelism properties in addition to high throughput and scalability: low-latency, disjoin-parallel, and independency from architecture and hardware-specific optimization.

1.4. Outline

The remaining chapters of the dissertation are organized as follows: Chapter 2 presents a comprehensive literature survey that studies research proposals on energy-aware disk storage management using DPM techniques as well as state-of-the-art on shared-memory parallel join algorithms. Chapter 3 presents the power-aware solutions proposed in this research for database
disk storage systems including DPM optimization model and DPM heuristic algorithms followed by empirical evaluation and results. Chapter 4 illustrates the proposed parallel hash-based stream join algorithms in this research including parallel symmetric hash join, chunk-based pairing hash join, and parallel multi-way hash join followed by the experimental evaluation, results and comparisons. Finally, Chapter 5 provides the summary and future works, summarizing the major finding and contribution of this dissertation.
Chapter 2

Literature Survey

This chapter provides a comprehensive literature survey on previous work related to this research. In this survey, we study the research proposals on designing a power-aware disk storage system using dynamic power management (DPM) techniques. This survey presents a taxonomy of effective DPM methods for energy conservation in storage systems. Then, the existing power-saving solutions are reported and mapped onto this taxonomy. Furthermore, we explore the state-of-the-art on shared-memory parallelization techniques of stream joins as well as database joins on multi-core processors.

2.1. DPM Techniques in Disk Storage Systems

The energy conservation in disk-based storage systems is addressed in many research projects. DPM algorithms are the most popular techniques to target energy saving in this context. Basic DPM techniques attempt to transition disks to lower-power mode while experiencing relatively long idle periods. However, without a careful design, they usually provide little energy saving or suffer from severe performance degradation due to non-negligible extra response time and energy costs imposed by spinning disks up and down when stand-by disks should service requests. In this section, we present a taxonomy on the effective DPM techniques. We report the existing power-saving solutions in the literature and map them on to our taxonomy.

Intuitively, the core idea of an effective DPM algorithm is to prolong the idling period of disks in order to allow them sleep longer in the lower-power mode and thus, boost power saving.
opportunity. We classify algorithmic techniques extending disks idleness period into three different categories. Figure 2.1 shows taxonomy of the effective DPM techniques aiming at energy-efficiency or power-proportionality in disk storage systems.

2.1.1. Data Consolidation Techniques

The major approach taken in effective DPM algorithms is data consolidation or data packing that concentrates the frequently accessed data (hot fragments) into fewer number of disks (hot disks) in order to help other disks stay in idle mode longer.

Popular Data Concentration (PDC) presented in [PB04] utilizes the load consolidation idea. More specifically, the goal of PDC is to spread the data out across the disk array so that the first disk stores the most popular data, the second disk stores the next set of most popular data, and so on. The least popular data and the data that are never accessed will then be stored on the last few disks. In order to avoid overloading disk and performance degradation, PDC only migrates data onto a disk until the expected load on the disk is close to its maximum bandwidth for the workload. Then, PDC spin down the idle disks after a fixed period of idleness (called idleness threshold in literature) by periodically checking each disk after its last access time. A spun down disk is

![Figure 2.1 Taxonomy of effective DPM techniques in disk storage systems.](image)
reactivated again on the next access request to it. PDC is a logical I/O behavior-based data migration method which means that the unit of data is file (and not physical data item).

PDC is implemented in Nomad FS, a prototype energy-aware file server written in C++. Nomad FS is a user-level, event-driven server that works on top of the local file system. The server assigns a thread with each disk. This thread directly touches the disk, either for I/O operations or migrations. Based on PDC, the entire files are moved among disks although the server receives requests for 8-Kbyte file blocks. PDC works fine for specific workloads such as Web, proxy, ftp, and email server workloads, which access entire files at a time. As the major limitation, it is not clear how PDC adapts to different types of workloads as access frequency of data items are assumed static in this work while popularity of data fragments can significantly change with respect to time.

Another proposal of this category is presented in [OR10]. An efficient algorithm named dynamic block exchange (BLEX) is introduced that dynamically achieves load consolidation and performs necessary block exchange between disks. To the best of our knowledge, BLEX is the most effective algorithm in literature that tries to handle the dynamic I/O traces.

BLEX algorithm dynamically determines the number of hot disks and cold disks based on the observed workload. More specifically, the number of hot disks, called \( m \), is equal to total data request arrival rate on the storage system divided by the maximum arrival rate that is sustainable by a hot disk while satisfying the response time limit. This rate is called *sustainable rate or target threshold (TargetTH)*. Then, the first \( m \) disks in the sorted list of disks in decreasing order of their hotness level are set as hot disks and the rest of the disks are spun down to sleeping mode as cold disks. BLEX performs two types of data exchange between disks as follows: (1) when a data block is accessed on a cold disk, after serving the request by spinning up the disk, it exchanges the
accessed data block on the cold disk with an un-accessed data block on a hot disk. This exchange is done if the examined arrival rate of the accessed cold disk is lower than a threshold called LowTH which is an arrival rate threshold applied to each disk to determine whether to proceed with this type of data exchange. The logic behind of this type of data exchange is to reduce the temperature of cold disks to extend their idle time periods; (2) when a hot disk is overloaded, the accessed data blocks on the disk are exchanged with un-accessed data blocks on non-overloaded hot disk in order to balance the load between the hot disks. More specifically, given an arrival rate threshold called HighTH, if the observed arrival rate on a hot disk exceeds HighTH, it is marked as hotspot. Then, for each data request arriving to the hotspot, after serving the request, the accessed data block is exchanged with an un-accessed data block on a non-hotspot disk among other hot disks and this search starts from the hot disks with lowest arrival rate. Such procedure will continue whenever a request arrives to the hotspot disk until the arrival rate of the disk becomes equal or lower than the TargetTH. Then, the hotspot mark will be removed.

The performance of BLEX is evaluated by running experimental simulations developed in SIMPY [SI10] (simulation package written in Python) using real and synthetic workloads. According to [OR10], BLEX is capable of saving energy up to 50% with acceptable response time degradation. However, we believe BLEX might not efficiently enough adapt to dynamicity in workloads especially the ones with significant arrival rate change with respect to time such as database servers. The reason is that it performs insufficient data migration and maintains some data in stand-by disks and therefore, in order to adapt to dynamic changes in data request arrival rates, it potentially needs to pay some significant penalty related to spinning stand-by disks up and down. Other proposals exploiting data concentration are found in [CG02], [WO07], [VK10]. MAID [CG02] and PARAID [WO07] both directed their research efforts at the system level for
energy conservation. MAID, instead of migration, decides caching (duplication) of frequently accessed data into a subset of disks called cached disks in order to exploit data concentration idea. The cached disks are always active and other disks, called passive disks, are transitioned to lower-power mode when they experience a pre-determined idleness period. MAID’s write policy is designed to avoid spinning up idle disks. Data write requests missed in cache are written to the cached disks, and later will be written to the corresponding passive disks when they are spawn up to service read requests.

PARAID (Power-Aware RAID) is a RAID-based storage system that disks within each RAID group can “shift gears” (shift between power modes) based on the observed I/O work. In

Table 2.1 Characteristics of power-saving solutions using data consolidation techniques

<table>
<thead>
<tr>
<th>Solution</th>
<th>Approach</th>
<th>Evaluation tools</th>
<th>Traces</th>
<th>Disk Model</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDC</td>
<td>Data Migration</td>
<td>Own simulator</td>
<td>Synthetic, Humming bird, Pop</td>
<td>Seagate Cheetah, ST39205LC (SCSI)</td>
<td>Energy saving</td>
</tr>
<tr>
<td>BLEX</td>
<td>Data Migration</td>
<td>Own Simulator</td>
<td>Synthetic, OLTP, Cello99</td>
<td>Seagate ST3500630AS</td>
<td>Power saving, response time</td>
</tr>
<tr>
<td>MAID</td>
<td>Caching/Duplication</td>
<td>Own Simulator</td>
<td>Super computing center trace</td>
<td>IBM G60GXP</td>
<td>Energy, Response time</td>
</tr>
<tr>
<td>PARAID</td>
<td>Replication</td>
<td>Prototype</td>
<td>Web Trace</td>
<td>Fujitsu MAP 3367 (SCSI)</td>
<td>Response time, power consumption</td>
</tr>
<tr>
<td>SRCMap</td>
<td>Replication</td>
<td>Prototype - simulation</td>
<td>Blktrace [AJ07]</td>
<td>WD5000AAKB, WD360GD, WD2500AAKS</td>
<td>Power consumption, response time</td>
</tr>
</tbody>
</table>
particular, PARAID \((N, M)\) represents a RAID group consisting of \(N\) disks where \(M\) is the maximum number of disks allowed to be shifted to stand-by mode (at least \(N-M\) active disks). For example, PARAID \((4, 2)\) used in [WO07] provides three gear levels available for a RAID group. The lowest, middle and highest gears include two, three and four disks on high-power mode respectively. PARAID uses replication to exploit data concentration on active disks at each gear level within a RAID group.

Since data replication has significant space overhead (i.e., PARAID), Sample-Replicate-Consolidate Mapping (SRCMap) [VK10] suggest a more advanced replication model for data consolidation that replicates only the working data set. The number of replicas for each data set is variable based on the corresponding cost and benefits. This results in a much less space overhead which is the amount of free space available in the storage system. SRCMap samples a subset of blocks from each disk that constitutes its working set and replicates these on other physical disks. It activates the minimum number of storage volumes that guarantees availability of at least one replica and transitions other disk volumes to sleeping mode to enable energy-proportionality. Table 2.1 highlights the characteristic of the power-saving solutions described in this section.

2.1.2. Power-Aware Cache and Prefetching Algorithms

Strategies aiming at storing data in faster and volatile memories can greatly enhance energy efficiency in disk storage systems. To that end, most of the techniques in this category tackle caching and prefetching. Caching is useful because it reduces disk utilization in general, while prefetching data into memory usually helps to extend disk idleness period. The strategies of this type try to manage I/O intervals via power-aware caching and prefetching algorithms to extend disk inactivity periods. Therefore, what all techniques in this category have in common is that they
try to gather hot data in highly energy-efficient memory while usually maintaining cold data on hard disks.

In the context of caches, LRU (Least Recently Used) is the widely used algorithm for the data block replacement policy since it is simple and effective. There are power-aware cache replacement algorithms that extend LRU with focus on energy efficiency. For instance, Power-Aware LRU (PA-LRU) and Partition-Based LRU (PB-LRU) are proposed based on this idea and they consider workload features related to energy cost [ZZ05]. PA-LRU estimates the energy cost of a data block replacement and produces miss sequences that minimize energy consumption [ZD04]. PB-LRU partitions the entire cache to separate partitions, one for each disk. It divides the cache in a way to minimize total energy consumption of the storage system. The partition sizes are dynamically adjusted based on the observed workload [ZS04]. PB-LRU performance is similar to that of PA-LRU, however, with significantly less parameter tuning. This makes it an easier choice to adapt to existing storage systems. PA-LRU and PB-LRU show that they are capable of saving 22% more energy than LRU while providing better response times for OLTP workloads as well [ZZ05, ZS04].

The power-aware caching techniques discussed in [ZZ05] assume that the memory run at its maximum utilization, meaning that it is full and swapping data in and out based on its existing policies. Thus, these techniques do not impact on memory energy usage. However, as briefly discussed by Zhu in [ZZ05], the smarter caching techniques might incur more processor cycles and thus, more CPU energy. But, they claim that this overhead on CPU is much less than the energy saving benefits of their caching techniques. Also, it is mentioned that their algorithms can be even handled by slow but low-power processors inside the storage systems. Prefetching can be also extended with focus on energy conservation in disk storage systems without performance
degradation [PS04], [SK06]. A set of rules can be used towards energy-aware prefetching. One rule is to ensure that prefetch operations do not interrupt stand-by disks unless delay on operations compromises the performance. Another rule is that prefetch operations should take place if blocks are available for replacement. This prevents the disks from spinning down when the idle period would be too short to exploit energy saving. As a result, prefetching integrated with these set of rules decreases power consumption while maintaining the performance.

Other power-aware solutions that focus on caching and prefetching rely on multi-tiered caches and data storage [WY08, PG11]. They are based on using hybrid HDD and SSD drives as caching storage media. More specifically, caching and prefetching strategies are designed to focus on increasing SSD usage and decreasing hard drives so that disks can remain in sleeping mode for longer periods of time. This yields to both energy and performance benefits since SSDs are faster and power proportional.

Most of the methods described in this category are complementary. Caching and prefetching solutions usually do not degrade the performance. The only drawback is the cost such as cost for SSD-based solutions or implementation cost for algorithmic solutions.

Moreover, there is another type of power-aware solution that is similar to this category in sense of employing cache to control I/O intervals; however, it combines redundancy with cache usage at the same time. It tries to redirect some I/O requests to be served by redundant data available in cache or on high-power mode (active) redundant disks in order to let stand-by disks sleep longer. This scheme can efficiently adapt to RAID-based storage layouts [LW04, YX06].

2.1.3. Energy-Proportional Strategies in Distributed Systems

This section discusses the solutions that target energy-proportionality in large-scale distributed storage systems. The traditional data placement strategies existing in distributed
systems restrict achievement of power proportionality [LK10] . Thus, all techniques in this category try to achieve energy proportionality either by extending the existing data placement strategies or by proposing novel data layouts. Thus, energy-proportional data placement is the core of all existing solutions for distributed systems [KC14, KC11, CK11, 28, AC10, KB10, TD11, KM11, SK08, HS10, ND08, SK10].

In data parallel computing clusters, the distributed file systems maintain a set of replicas for each data block. They introduce the covering set as a group of nodes that together contain at least one replica for each data block needed for performing computing tasks. There are algorithms developed to discover a covering set that minimizes energy consumption while maintaining the remaining nodes into a lower-power mode [KC14, KC11, CK11].

Sierra [TD11] is an energy-proportional distributed system for general read and write workloads. It introduces a new power-aware layout that allows a significant fraction of the servers to be powered down without loss of availability, load balancing or fault tolerance. It uses a novel distributed virtual log to perform updates to objects when some replicas are turned off.

Rabbit [AC10] is a power-proportional distributed file system (PPDFS) that uses its novel equal-work data layout policies to support ideal power proportionality down to very low minimum number of powered-down nodes which is enough to store a primary replica of available datasets. Rabbit maintains near ideal power proportionality in the face of node failures.

GreenHDFS [KB10] is an energy-conserving, hybrid, logical multi-zoned (Hot and Cold zones) variant of Hadoop’s compute cluster. It relies on data classification-driven data placement to support substantially long periods of idleness in a subset of servers designated as Cold zones in the Hadoop cluster. It argues that zoning in GreenHDFS does not affect the Hot zone’s
performance adversely and the computational workload can be concentrated on the servers in the Hot zone without exceeding CPU utilization limits.

2.1.4. Other Techniques

Some methodologies integrate DPM algorithmic techniques with hardware facilities to provide more power saving opportunities. One clear-cut instance is to take advantage of different rotation speeds available in multi-speed disks. Hibernator [ZC05] and DRPM [GS05] dynamically configure disk speed based on the observed workload.

DRPM adjusts each disk speed accordingly based on the arriving data request rate to match the required performance. It provides a detailed analysis of disk power consumption and how rotation speed levels can affect energy consumption [GS05]. This strategy potentially can improve the energy proportionality of hard disks.

Hibernator dynamically partitions the disk array into different tiers and assigns different speeds to disk tiers. And, it consequently moves data blocks to proper disk tier based on block hotness level—which depends on the observed data request arrival rate. The main idea of the disk speed setting algorithm is to adapt the disk speed infrequently and keep it constant during a relatively long epoch. Thus, at the beginning of each epoch, the best speed configuration for each disk is determined based on the predicted workload for the disk and the response time limit. The rotation speed of a disk is kept the same throughout the entire epoch unless an unexpected workload change happens. In this case, there is a performance guarantee method that would be called to take over.

A research work conducted under the EU GAMES project [GR10] proposed an adaptive mechanism for energy-aware data storage control [CH11]. It focuses on application-driven storage control in file storage systems—files are data units—for energy efficiency. Also, another research
paper utilized the application level I/O behaviors for power saving in storage systems cooperated
with data-intensive applications [NN12].

2.2. Stream Joins

In this section, we first introduce stream join algorithms on sliding windows. Then, we will
discuss the parallel shared-memory stream join algorithms existing in the literature.

2.2.1. Join Algorithms on Sliding Windows

We follow the semantics of sliding-window join as described in the related literature [GB09,
TM11, GN15, GJ12]. A data stream is an unbounded sequence of tuples that share the common
schema based on application. In particular, each input tuple \( t \) has the schema \(< ts, Att_1, …, Att_n >\)
where \( ts \) represents generation timestamp and \( Att_1, …, Att_n \) are attributes based on the
application. Tuples are assumed to arrive in timestamp order in an input data stream. Due to
unbounded nature of input data streams, the notation of \textit{window} is introduced in the streaming
community so that database operators can be also evaluated over data streams Stream joins
compare tuples generated by two logical streams, \( R \) and \( S \), based on predicate \( P \). The join is applied
only to a finite subset of each input stream. There are different ways to define proper boundaries
for windows depending on the application. One way is to restrict the scope of the join operator to
recent portion of each data stream. This leads to \textit{sliding windows} which is the most common
windowing method in literature. At any point in time sliding window covers all tuples from an
earlier point in time up to the most recent tuple. Sliding windows are time-based or tuple-based.
The former covers all tuples within the last \( t \) seconds. The latter covers the last \( n \) tuples in arrival
order. Both types of windows slide forward as time proceeds or new tuples arrive, respectively.
We consider time-based sliding windows with \( TS \) time units that includes all tuples \{t | t.ts −
t.ts \leq TS\} where \( t \) is the latest received tuple in the stream. The time-based sliding window
concept can also be easily extended to tuple-based windows that hold a fixed amount of the last \( TS \) received tuples. When the predicate \( P(t_R, t_S) \) is satisfied for tuples \( t_R \in R \) and \( t_S \in S \), an output tuple \( t_O \) is produced combining \( t_R \) and \( t_S \) and setting the timestamp of the output tuple to the maximum between timestamps of \( t_R \) and \( t_S \).

The semantics of nested loops-based stream join is commonly implemented as the *three-step procedure*. For each incoming tuple \( t_R \) belonging to \( R \) (symmetrically for tuple \( t_S \)), this procedure (1) compares \( t_R \) with all \( t_S \) belonging to \( S \) based on the join predicate; (2) adds \( t_R \) to window for \( R \); (3) invalidates all expired tuples from window of \( R \).

Stream join evaluation has high computational cost. For instance, if we assume that both \( R \) and \( S \) receive \( N \) tuples per time unit and both windows for \( R \) and \( S \) hold \( N \times TS \) tuples on average. Because each tuple from \( R \) is compared with all tuples in window for \( S \) (and vice versa for tuples from \( S \)), the average number of comparisons per time unit is \( 2 \times TS \times N^2 \). For instance, if window size equals to 10 minutes and \( N \) equals to 500 tuples per second, a stream join must perform 300 million comparisons per second on average. Thus, this high computation cost of stream join has motivated many research efforts to reduce it through parallel design of stream join algorithms on multicore CPUs.

*Symmetric Hash Join* (SHJ) is one of the fundamental algorithms for stream join processing [XY07]. It is a hashing-based join algorithm, which also has been used to support highly pipelined processing in parallel database systems [WA91]. It supposes that the hash table for each data stream can be kept in the main memory. This algorithm symmetrically performs two phases for newly arrived tuples: build phase and probe phase. In particular, for each incoming \( t_R \in R \), (1) SHJ inserts it into the hash table for \( R \) (building phase); (2) then it probes the hash table for the partner stream of \( R \) to find joining tuples (probing phase). SHJ performs the same procedure
symmetrically for the tuples $t_S$ associated with the other data stream $S$. Figure 2.2 shows the symmetric hash join algorithm for two input data streams $R$ and $S$.

Note that sorting-based join algorithms, such as the sort-merge join, have been traditionally considered unsuitable for stream joins since sorting is a blocking operation that needs to see the entire input before producing any output [XY07].

2.2.2. Shared-Memory Parallel Stream Joins

In this section, we explore the proposed shared-memory parallelization techniques for stream join algorithms on multicore processors. In literature, shared-memory parallelization techniques for nested-loop join version of stream join algorithm have been proposed [GB09, RT14, TM11, GN15]. We will discuss the two major state-of-the-art parallelization techniques in this context. To our best of knowledge, the parallel shared-memory SHJ algorithm on multi-core CPUs has not been so far designed and implemented in the literature, which makes that an interesting and potential research work.

2.2.2.1. Handshake Join

Handshake join is a schema that describes and executes stream joins and is highly cooperative to parallelized design and execution. Handshake join can naturally take advantage of

Figure 2.2 Symmetric hash join algorithm for two data streams.
available hardware parallelism. It is based on the three-step procedure [KN03] corresponded to the nested loops-style join evaluation in data streams.

Traditional approaches (including CellJoin [GB09]) assume a central coordinator that divides and replicates data as required over the available cores in order to distribute the enumeration of join candidates. However, it is observed that this has become a bottleneck as the number of cores is increased. Given this limitation, handshake join does not depend on a centralized coordinator and aims to scale out to very high degree of parallelism.

Handshake join learns from soccer players and how all pairs of players from two opposing teams can be enumerated without any external coordination. Before the beginning of a game, players shake hands with all players from the opposing team by walking by each other in opposite directions and shaking hands with every player that they face. The handshake procedure used in sports games inspired the design of handshake join.

Two data streams, $R$ and $S$, flow by each other in the opposite direction and their tuples are pushed through the respective join window. When entering the window, each tuple pushes all existing tuples in the window one step to the side such that the oldest tuple falls out of the window and expires. Both join windows are lined up next to each other in a way that window tuples are pushed through in opposing directions similar to the players in soccer. Whenever two stream tuples $r \in R$ and $s \in S$ see each other, they shake hands, meaning that the join condition is assessed for $r$ and $s$, and a result tuple $<r, s>$ is added to the join result if the condition is fulfilled.

It is clear that many handshakes happen at the same time that can be parallelized over cores. Now, in order to parallelize handshake join over cores, each processing core is assigned one segment of the two lined up join windows. Tuple data is kept in local memory and all tuple comparisons of the segment are performed locally. The strategy used to process a segment by its
assigned core is called *immediate scan strategy*. In particular, when every tuple \( r \in R \) enters the segment, it is compared to all \( S \)-tuples in the segment immediately. Likewise, when a new tuple \( s \in S \) enters the segment, it is immediately compared to all \( R \)-tuples that are already in the segment.

Since a newly arriving tuple would synchronously push all tuples of the same stream through the respective window, all cores must simultaneously forward their oldest tuple to the respective left/right next neighbor. This requires an atomic operation over all participating processing cores and therefore an inter-core communication mode. Handshake join implementation is based on *asynchronous message passing* between neighboring cores. The aforementioned communication mode is known for its scalability advantages as number of cores increases [BB09]. In particular, one pair of FIFO queue between any two neighboring cores is implemented to perform data propagation along the chain of cores in either direction.

The aforementioned queue–based message passing scheme has potential risk of missed-join pair problem, which happens when tuples sent through message queues miss each other for joining while on the communication channels. In order to avoid the missing of the join candidates, *two-phase tuple forwarding* is introduced to the queue-based message passing scheme implemented in handshake join. More specifically, whenever a right core \( C_{\text{Right}} \) places a tuple \( t_i \) into its left send queue, it still keeps a copy of it in the local window join, but marks it as *forwarded*. The forwarded tuple \( t_i \) then is available for joining on \( C_{\text{Right}} \) until the second phase of tuple forwarding, which is initiated by an acknowledgment message from the left neighboring core \( C_{\text{Left}} \). After \( C_{\text{Right}} \) receives an acknowledgement from \( C_{\text{Left}} \), it removes tuple \( t_i \) from its local join window.
2.2.2.2. ScaleJoin

ScaleJoin is a shared-memory parallel stream join algorithm that is based on the three-step procedure [KN03] corresponded to the nested loops-style join evaluation in data streams. ScaleJoin is a deterministic, disjoint (independent) parallel, skew-resilient stream join that offers high-throughput and low-latency joining of tuples delivered by arbitrary numbers of streams. These features are essential to exploit shared-memory parallel stream joins in streaming applications. Novelty of ScaleJoin is a new data structure called ScaleGate that distills a minimal interface for satisfying the aforementioned features and parallelism requirements. It provides ready tuples (guarantying determinism) and allows processing cores to execute join comparisons in a disjoint-parallel way. ScaleJoin is implemented on top of ScaleGate. Given rate-varying and bursty streams, ScaleJoin keeps a balanced work among the processing threads by relying on the ScaleGate, guarantying skew-resiliency.

Determinism means that given the same sequences of input tuples, the same sequence of output tuples will be produced, independently of the tuples inter-arrival time and processing order.

Figure 2.3 Overview of ScaleJoin’s architecture and parallelization approach [GN15].

Figure is reused from [GN15]. Permission of use can be found in Appendix A.
The 3-step procedure in sequential stream join does not guarantee deterministic processing. ScaleGate guarantees determinism by merging the timestamp-sorted $R$ and $S$ physical streams into one time-stamp sorted physical stream of \textit{ready} input tuples, which is then consumed by processing threads. The threads act as \textit{reader} entities for this queue of ready tuples.

Figure 2.3 shows an overview of ScaleJoin’s architecture and parallelization approach. ScaleJoin provides parallel execution of an arbitrary number $n$ of Processing Threads ($PTs$) that each consumes and matches the input tuples delivered by $R$ and $S$ streams. Its processing includes three phases: (1) delivery of input tuples to $PTs$, (2) matching of tuples at $PTs$ and (3) collection of $PTs$’ output tuples. The first ScaleGate (SGin) is responsible for the first phase. It merges the $R$ and $S$ tuples generated by an arbitrary number of physical $R$ and $S$ input streams into a single timestamp-sorted stream of \textit{ready} tuples.

In the second phase, the aforementioned timestamp-sorted queue of ready tuples is consumed by $n$ $PTs$. The second ScaleGate (SGout) is responsible for the third phase. It merges the output tuples generated by each thread into a single timestamp-sorted stream of ready output tuples. Figure 2.4 demonstrates scalability and throughput (number of comparisons per second) of
ScaleJoin as number of processing threads increases. ScaleJoin’s performance is compared with that of Handshake join on two different systems [GN15].

Similar to stream joins, relational database joins are also one of the most important database operators and are very common in database query workloads. Also, database joins are normally known as both data and computation intensive, and thus contribute significantly to the execution costs in queries. There are two common join algorithms: hash join and sorted-merge join. In this section, we explore the design spaces for both in-memory hash join and sort-merge join on modern multi-core processors.

2.2.3. Multi-Way Stream Join

Multi-way stream join is considered as a major extension to the original binary stream join where more than two input data streams are involved in the join predicate evaluation. There is a multi-way stream join algorithm designed explicitly for sliding windows in [GO03] named _eager_ multi-way join with two versions: nested-loops join (NLJ) and hash join. The only difference is that the latter considers hash buckets for scan purpose of the join evaluation while the former version scans the entire sliding window.

Eager multi-way join algorithm executes multiple joins together in a series of nested for-loops and process newly arrived tuples from each window separately with possibly different join
orderings. Also, there is a join heuristic for this algorithm that attempts to minimize the number of intermediate tuples that are passed down to the inner for-loops. This means that the stream to which the new tuple belongs is processed in the outer-most for-loop, followed by the stream with the smallest number of selected tuples, and so on.

We display multi-way join for the following example. Let \( S_1, S_2, S_3 \) be three sliding windows to be joined with the global join ordering as \( S_1 \bowtie (S_2 \bowtie S_3) \). In Figure 2.5, the join order is shown as a join tree (left) and a series of nested for-loops (right). We refer to \( S_1 \) as being ordered first, \( S_2 \) as being ordered second and so on. The pseudocode on right shows the general strategy for multi-way join where the join is evaluated from top to down. The predicate is checked inside each for-loop in order to minimize the number of tuples that are passed down to the inner loops.

As we will exploit hash-based version of this algorithm for our parallel multi-way hash join algorithm (Chapter 4), we now explain detailed steps for eager multi-way hash join algorithm and its join ordering heuristic. Upon the arrival of a new tuple in one stream, it is inserted to the hash table of its origin stream as the build phase. In the probe phase, first, the expired tuples from the other streams are removed. Then, the join order for probing changes such that the stream to which the newly arrived tuple belongs will be ordered first, followed by the stream whose hashed bucket has the smallest size, and so on. In effect, the window at top of the order always have only one tuple and the join order changes in response to the origin of the new tuple. This is possible since it is assumed that there is a common join attribute across all input streams. This join heuristic, as mentioned earlier, attempts to minimize the number of tuples passed down to the inner loops to reduce the cost for scanning the buckets.
2.3. Database Joins

Relational database join is one of the most important database operators and is very common in database query workloads. Also, database joins are normally known as both data and computation intensive, and thus commonly contribute significantly to the execution costs in queries. This has motivated much research efforts to make join implementation more efficient through parallel design of database joins on multi-core CPUs. There are two common join algorithms: hash join and sorted-merge join. In this section, we explore the design spaces for both in-memory hash join and sort-merge join on modern multi-core processors.

2.3.1. Hash Joins

In-memory hash join algorithms proposed in the literature can be classified into two contradictory categories: hardware-oblivious and hardware-conscious [BT13]. The former hash join variants do not rely on hardware-specific parameters and they instead consider qualitative features of modern hardware [BL11]. They claim to achieve good performance on any similar hardware. They argue that tuning of hash join algorithms to particular underlying hardware makes them less portable and less robust. On the other hand, the latter variants of hash join claim that the

![Figure 2.6 Canonical hash join.](image)

Figure 2.6 Canonical hash join.

Figure is reused from [BT13]. Permission of use can be found in Appendix A.
best performance can be achieved by fine tuning the algorithm parameters (i.e., hash table sizes) to the underlying architecture [KS09], [LL13], [AK12].

2.3.1.1. Canonical Hash Join

The canonical hash join algorithm is the foundation for any modern hash join implementation [TO11, KT83]. As shown in Figure 2.6, it consists of two phases: build phase and probe phase. In the first phase (build), the smaller of the two input relations, $R$, is scanned to fill a hash table with all $R$ tuples. Then, in the probe phase, it scans the second input relation, $S$, and probes the hash table for each $S$ tuple to find matching $R$ tuples.

2.3.1.2. No Partitioning Join

A direct parallel version of the canonical hash join is proposed in [BL11] that is called no partitioning join. It does not rely on any hardware-specific parameters in contrast to the contrary camp and does not physically partition data—as partitioning is the core for hardware-conscious joins as will be discussed shortly. They argue that the partitioning needs multiple passes over the data and can be removed by depending on modern CPU characteristics such as simultaneous multi-threading (SMT) to hide cache latencies.

Figure 2.7 No partitioning join.

Figure is reused from [BT13]. Permission of use can be found in Appendix A.
Figure 2.7 illustrates no partitioning hash join. Both input relations are separated into equal-sized parts and these portions are assigned to a number of processing threads. In the build phase (first phase), all processing threads populate a hash table shared between all threads. After synchronization via a barrier, all processing threads enter the probe phase to find matching joins for their assigned $S$ portions. An important feature of this type of parallel hash join is that the hash table is shared among all processing threads. Thus, concurrent insertion accesses to hash table should be synchronized. Therefore, each bucket of the hash table is protected through a latch. Each thread should obtain the latch before insertion. The contention over the latch among threads is expected to be low since the number of hash buckets is usually large in the scale of millions. However, since the probe phase is a read-only stage, there is no need for synchronization and latches between threads while probing the hash buckets. Thus, the processing threads can concurrently read from the hash buckets.

### 2.3.1.3. Radix Join

Given that hashing in the main-memory results in cache misses due to its random access nature, the main idea behind hardware-conscious main-memory hash join implementation is tuning main-memory access of hashing by using cache in a more efficient way, which yields better query

![Partitioned hash join](image)

Figure 2.8 Partitioned hash join.

Figure is reused from [BT13]. Permission of use can be found in Appendix A.
performance. Therefore, *partitioning* the hash table into cache-sized blocks decreases cache misses and improves performance. This idea is refined in [MB02] by considering the influence of translation look-aside buffers (TLBs) during the partitioning. This resulted in *multi-pass partitioning* which is a key part of radix join.

- **Partitioned Hash Joins**: Figure 2.8 shows the partition idea for hash join. In the first phase of the algorithm (partition), the two input relations $R$ and $S$ are divided into partitions. In the build phase, a separate hash table is created for each partition of $R$ (assuming $R$ is the smaller relation). Each of these hash tables can fit into the CPU cache now. In the probe phase, partitions of $S$ are scanned and the respective hash table for each of these partitions of $S$ is probed for matching tuples.

  In the partitioning phase, input tuples are divided up based on their key value by using hash partitioning, $h_1$ hash function in Figure 2.8. And, another hash function $h_2$ is used to populate the hash tables. Therefore, each partition of $S$ can only match with the partition from $R$ with the same index number ($r_i \bowtie s_j = \emptyset$ for $i \neq j$).
Although partitioning the input tables avoids cache misses, it may cause another type of cache issue. The partitions have separate entries for virtual memory mapping since they usually reside on different memory pages. In modern processors, this mapping is cached by TLBs. Therefore, if the number of partitions is too large, the partitioning phase can experience TLB misses. The number of entries in TLB determined the maximum number of partitions allowed to be accessed concurrently in an effective way. Radix partitioning as described in the following can solve the aforementioned TLB miss problem.

- **Radix Partitioning**: Multiple-pass partitioning can prevent TLB misses. More specifically, in each pass, all partitions produced by the preceding pass are refined such that the partitioning fan-out does not violate the hardware limit given by the number of TLB entries. In reality, each pass considers a different set of bits from the hash function $h_1$. For common in-memory data sizes, two or three passes are enough to create cached-size partitions.

- **Radix Join**: Figure 2.9. Shows the complete Radix join. At first, both inputs are partitioned using two-pass radix partitioning (assuming two TLB entries are enough for this example). Second, hash tables are then built over each partition of table $R$. Finally, all partitions of $S$ are scanned and the respective partitions from $R$ are probed for finding the matches.

### 2.3.1.4. Parallel Radix Join

Parallelization of radix join can be done by dividing both input tables into sub-tables and assigning them to processing threads [KS09]. In the first pass, all threads create a shared set of partitions. As mentioned earlier, the number of partitions in this set is normally small since it is restricted by the hardware parameters. Thus, there is contention issue on these partitions since there are accessed by many individual processing threads. In order to solve this problem, within each output partition, a dedicated range is maintained for each thread. To do so, both input relations
need to be scanned two times. The first scan computes a set of histograms over the input data to
determine the exact output size for each thread and each partition. In the second scan, a contiguous
memory space is allocated for the output and each thread pre-calculates the allocated location
where it writes its output through computing a prefix-sum over the histogram. Finally, all threads
can execute partitioning with no need for synchronization.

After the first pass, there is normally enough independence in the rest of the system
between threads such that they can successfully perform their tasks without contention issue. Task
Queuing [KS09] model is normally used to distribute the workload between the processing threads.

According to the results in [BT13], the hardware-conscious algorithms keeps an edge over
hardware-oblivious in most systems and configurations. Also, hardware-conscious has been made
faster than previous implementations and more robust to wider set of parameters. Furthermore,
since they are shown to be easily tuned to the underlying hardware, this can significantly decreases
the argument of being difficult to port compared to their hardware-oblivious counterparts.
However, there are particular cases where characteristics of modern hardware such as aggressive
on-chip multi-threading make hardware-oblivious algorithms competitive.
2.3.2. Sort-Merge Joins

Sort-merge join is another popular join operation in relational databases next to the hash join. In this section, we investigate the design space for parallel sort-merge join on multi-core processors. Efficient implementations of parallel sort-merge join utilize hardware features such as large SIMD vectorization units. Also, they are tailored to the underlying hardware in terms of memory architecture such as memory bandwidth awareness and NUMA awareness.

First, we provide required background on the classic sort-merge join algorithm, as shown in Figure 2.10. Then, we describe design choice for different phases of sort-merge join, known as parallel sorting and parallel merging. After discussing parallel sorting and merging as building blocks, we explore variants of the overall sort-merge join algorithm that are both tailored for the memory architectures and SIMD on multi-core processors.

2.3.2.1. Classical Sort-Merge Join

Figure 2.10 depicts the classical sort-merge join algorithm consisting of two phases: sort and merge. In the first step, both input relations are sorted based on the join key. In the second step, both relations are checked for matching joins based on merge-join operation. This operation runs as follows: It scans both relations sequentially and there is a head pointer that keeps track of the current position in each table. The join condition is checked on the head tuples of both relations and, if satisfied, an output tuple is created. Then, the cursor on the relation with the smaller value proceeds and the aforementioned join condition evaluation is repeated, as described.

The algorithmic complexity of the sort-merge join mainly originates from the sorting phase. The complexity of sorting is $O(n \log n)$ where $n$ is the size of a relation. The merge phase has linear complexity, and thus is dominated by sorting step.
2.3.2.2. Parallelization of Sorting Phase using SIMD

As mentioned earlier, the major cost in sort-merge joins is sorting the input tables. Thus, we explore the techniques to implement sorting in a hardware-conscious fashion. The sorting algorithm that is normally used in sort-merge join is *merge-sort*, which is a divide and conquer type of algorithm. It consists of two building blocks: initial run generation, and merging of the pre-sorted runs [BA13]. Both building blocks can exploit SIMD as described in following sections.

- **Initial Run Generation**: In order to generate the initial runs, many chunks with small number of tuples should be sorted. The sorting algorithms that are capable of parallel processing of multiple chunks can benefit from this feature. Thus, *sorting networks* suit well with the SIMD execution model of multi-core CPUs [CN08, GB07, MT12].

- **Sorting Networks**: Figure 2.11 shows a sorting network for a four input items based on the notation in [KD98]. A set of four items, \(<9, 5, 3, 6>\), passes through the sorting network from

\[
\begin{align*}
  e &= \min(a,b) \\
  f &= \max(a,b) \\
  g &= \min(c,d) \\
  h &= \max(c,d) \\
  i &= \max(e,g) \\
  j &= \min(f,h) \\
  w &= \min(e,g) \\
  x &= \min(i,j) \\
  y &= \max(i,j) \\
  z &= \max(f,h)
\end{align*}
\]

Figure 2.12 Comparator implementation via *min/max* operators.
left to right through some *comparators*. Each comparator sends the smaller of its two input values to top and the greater one to the bottom. The data set is sorted after travelling through the five comparators.

The advantage of comparators is that they can be implemented via *min/max* operators only. In particular, the five comparators in Figure 2.11 can be implemented as a sequence of ten *min/max* operators as shown in Figure 2.12 (input variables are \( a, \ldots, d \) and output variables are \( w, \ldots, z \)). This implementation makes sorting networks parallelizable in an efficient way through SIMD. All variables in the code shown in Figure 2.12 can be instantiated with SIMD vectors of \( k \) items and all *min/max* runs can be switched by SIMD runs. Thus, \( k \) sets of items can be sorted in parallel by using SIMD.

- **Merging Pre-Sorted Runs:** Two pre-sorted runs can be merged into a larger sorted run by scanning both runs simultaneously and comparing the items on head. Despite the sequential nature of merging, it can take advantage from SIMD. The basic idea is described in [IH07] and has been used for the purpose of sorting in [CN08] and joins in [KS09].

![Figure 2.13 Bitonic merge network.](image)

Figure 2.13 Bitonic merge network.

Figure is reused from [BA13]. Permission of use can be found in Appendix A.

Figure 2.13 shows a bitonic merging network that combines two input lists of size four into an overall sorted output. The merge network consists of three stages, each has four comparators. Thus, assuming \( k \) equals to 4 for SIMD vector, each stage can be implemented by using one *max*
and one \textit{min} SIMD instruction. Shuffle instructions in-between stages put vector elements on their appropriate positions.

- \textit{Cache Conscious Sorting}: Given the cache hierarchies in modern hardware architectures, overall sorting process needs to be divided into several phases in order to optimize cache access as follows: (1) \textit{In-Register sorting}, with runs fitting in (SIMD) CPU registers; (2) \textit{In-Cache sorting}, where runs that can be kept in a CPU-local cache; and (3) \textit{Out-of-Cache sorting}, once runs that exceed cache sizes.

In phase (1), In-Register sorting associates to the run generation discussed in Section \textit{a}. In phase (2), In-Cache sorting merges runs until runs can no longer fit into CPU caches. In-Cache sorting corresponds to the bitonic merge networks discussed in Section \textit{b}. Phase (3) resumes merging until the data is completely sorted. However, given that runs have exceeded the size of the cache in this phase, memory references need to be fetched from off-chip memory.

The overall merge sort (sorting) algorithm can be summarized as follows:

1) Divide input into cache-sized chunks.
2) For each cache-sized chunk, perform In-Cache sorting (incorporates In-Register sorting in the first phase).
3) Out-of-Cache sorting.

Figure 2.14 Multiple two-way merging.

Figure is reused from [BA13]. Permission of use can be found in Appendix A.
- **Multi-Way Merging**: Out-of-cache merging is highly limited by the memory bandwidth. Thus, in order to reduce this memory bandwidth demand, more than two runs is merged at the same time which is called multi-way merging. Multi-way merging saves round trips to memory, increases CPU computation load and hence precious memory bandwidth. As a result, it brings balance between computation versus bandwidth.

  Multiple two-way merge units are implemented in [BA13] since it can still benefit from CPU-efficient, SIMD-optimized bitonic merging. Two-way merge units are connected by FIFO queues. The size of FIFO queues is determined such that all of them together can fit in the cache. Thus, external memory bandwidth is required just at the front of the merging tree. Figure 2.14 shows the multiple two-way merging.

- **Impact of NUMA**: Some merging passes unavoidably cross NUMA boundaries in practice. According to experiments shown in [LP13], NUMA interconnect bandwidth stands more and more behind the aggregate memory bandwidth in multi-sockets systems. Thus, there is a need to combat this problem in join implementation. Multi-way merging can also effectively address this issue in a scalable way, as will be shown in next sections.

- **Efficient Data Partitioning**: Since data partitioning and merging are duals of each other, sort-merge join algorithms can also be implemented using data partitioning. Sort-merge join algorithms based on data partitioning normally need partitioning to be performed according to different range values of the input. This range partitioning has commonality with hash join algorithms. Thus, the radix partitioning discussed in Section 2.3.1.3 can be also used for sort-merge joins. The radix join implementation can have new optimizations on modern processors, which will be discussed in this section. In the next section (2.3.2.3), the use of efficient data
partitioning in sort-merge joins will be discussed along with describing different types of overall sort-merge join algorithms.

- **Software-Managed Buffers**: As mentioned earlier in Section 2.3.1.3, the number of TLB entries is considered as an upper bound to the partitioning fan-out. In order to reduce the TLB limitation on the maximum fan-out, writes can be buffered first inside the cache. Thus, a set of buffers are allocated in the cache, one for each output partition and each buffer with room for $N$ input tuples. The buffers are copied to the final destination when they become full. Buffering causes extra copy overhead. However, all buffers can fit into a single memory page and into L1 cache for sufficiently small $N$. Hence, a single TLB entry will be enough unless a buffer becomes full and the copying process needs to happen.

### 2.3.2.3. Parallel Sort-Merge Join Algorithms

In this section, parallel sort-merge join algorithms are introduced. In particular, we discuss $m$-way sort-merge join, $m$-pass sort-merge join [BA13] and massively parallel sort-merge join (mpsms) [AK12].

- **Sort-Merge Join Algorithm – $m$-way**: The m-way algorithm is a highly parallel sort-merge join that exploits both data and thread parallelism and is NUMA-aware as well. The overall picture and the individual steps of the algorithm are shown in Figure 4.9. It is assumed that there four NUMA regions and one thread per region in Figure 2.15.

  At the beginning, input relations $R$ and $S$ are equally distributed across NUMA regions. In the first phase, each thread range-partitions its assigned NUMA-local chunk by the software-managed buffers technique described in previous section. The main reason behind partitioning is that it allows threads in the subsequent phases to work independently without any synchronization. In this phase, the partitioning fan-out is normally on the order of the number of threads (64-128)
and can be performed by using a single pass at the speed of total memory bandwidth of the machine. Then, each local partition is sorted by using the three-phase sorting algorithm described in previous section. In this phase, different threads are able to sort different partitions in an independent way.

Phase 2, as shown in Figure 2.15, requires re-arranging data between different NUMA regions, and thus, it is likely to be bound by the memory-interconnect bandwidth. Therefore, multi-way merging, as mentioned earlier in previous section, is used to reduce the inter-connect memory bandwidth limitation. Multi-way merging successfully can overlap the data transfer and merging and, thus making a balance between computation and bandwidth. Result of this phase is a globally sorted copy of $R$, shown as $R'$ in Figure 2.15. The same phases are applied to the other input relation $S$ (marked as phase 3 and 4 in Figure 2.15).

$R'$ and $S'$ are stored in the NUMA-local memory of each thread. In the last phase, each thread simultaneously checks the join between NUMA-local sorted runs using a single-pass merge join. Linear scan of both sorted runs is extremely fast and leads to matching pairs.
• **Sort-Merge Join Algorithm – m-pass:** The m-pass algorithm differentiates itself from the m-way algorithm only in phase 2. The m-pass performs successive two-way bitonic merging for merging NUMA-remote runs instead of multi-way merge in phase 2. The first iteration of merging is performed as the data is transferred to the local memory. After the first iteration, the number of runs becomes half of the initial total number of runs. The rest of the merging resumes by doing multi-pass merging technique in a repetitive way.

• **Massively Parallel Sort-Merge Join – mpsm:** The mpsm algorithm first **globally** range-partitions relation $R$ such that different ranges are allocated to different NUMA-regions. Next, each thread independently sorts its partition. This phase results in a globally-sorted $R'$. Relation $S$, in contrast, is sorted only **locally** and each thread sorts its own NUMA-local chunk without prior partitioning. Thus, in the last phase (merge-joining), a run of $R$ must be checked against all the NUMA-remote runs of relation $S$ for merge-joining.
Chapter 3

Dynamic Energy-Aware Disk Storage Management in Database Servers

In this chapter, we present our research ideas on designing a dynamic energy-aware disk storage system in database servers. First, a DPM optimization model integrated with MPC strategy is introduced that dynamically determines the state of disks and plans for inter-disk data fragment migration to minimize power consumption of the disk-based storage system while satisfying given performance requirements. Section 3.1 describes the DPM optimization model in detail. Second, a fast-solution heuristic DPM algorithm is proposed that can be integrated in large-scale disk storage systems for efficient state configuration and data migration. Section 3.2 illustrates the proposed heuristic DPM algorithm. Finally, our proposed power-aware ideas are evaluated by running simulations using extensive set of synthetic workloads. Section 3.3 presents in detail our experimental methodology.

3.1. Proposed DPM Optimization Model

In this section, we show the design of a DPM optimization model towards balance between energy consumption and performance. It is well-known that the arrival rate of data requests changes significantly in respect to time in I/O traces of database servers. This is particularly true in scientific database servers and OLTP servers. The SSDS SkyServer is a famous scientific database server that clearly shows significant changes in the server traffic rate [SD02]. The

---

1 This chapter was published in 27th International Conference on Database and Expert Systems Applications (DEXA) [BY16], and International Journal on Database Management Systems [BT17]. Permissions are included in Appendix A.
fluctuations in the data request arrival rate in SSDS server are observed to be up to 65% of the total arrival rate range. Also, [ZC05] shows arrival rate changes in an OLTP trace that demonstrates remarkable arrival rate changes with respect to time. The major problem of previous contributions is that they cannot efficiently enough adapt to dynamic I/O workloads. This research solves this issue by integrating model predictive control (MPC) strategy in an optimization model to enable optimization actions in an online fashion. Section 3.1.4 describes in detail how our optimization model integrates the MPC technique in order to capture the dynamic changes in data access frequency. In addition to the MPC strategy, another advantage of the DPM model is that it explicitly includes fixed penalty cost on disk status change to avoid excessive spin up and down operations that have expensive response time and energy costs. However, this is rather considered subjectively in BLEX algorithm – the best dynamic algorithm in literature.

Given the arrival rate changes in dynamic I/O workloads, we partition the planning horizon into multiple periods where the arrival rate in each period can be modeled by a constant. We formulate a model as a (nonlinear) mixed integer program (shown in Section 3.1) where the objective function is the overall cost from all energy consumption elements in the storage system during one epoch. At the beginning of each epoch, based on the observed workload and the predicted workload for the epoch, the model configures the optimal disk state setting and corresponding inter-disk data migration such that the energy consumption (aforementioned objective function) during the epoch is minimized while maintaining query response time quality. In order to avoid the disk overloading problem, the model performs load balancing between the overloaded disk(s) and other active disks at the beginning of each epoch.

The length of the epoch should be short enough to capture changing arrival rates and also long enough to accommodate disks transition cost and data migration periods as well as tolerable
number of on/off actions on disks in order to not damage their lifetime services. Considering arrival rate change patterns existing in database I/O traces, we verified different epoch length values to determine an efficient value that fulfills the above requirements. Based on our sensitivity analysis described in Section 3.3.4.5, the energy saving ratio is insensitive to the epoch lengths larger than 30 minutes. Therefore, we determined the epoch length to be 30-minute long since it captures arrival rate changes effectively while exploiting energy savings.

The assumptions in the model are as follows:

1) Any fragment can only be migrated once in a period.
2) Power rate of each disk depends on its current state (rotation speed).
3) The data migration cost (time) between any two disks is intuitively proportional to the total size of data fragments to be migrated and is independent of the source and destination disks.

Table 3.1 introduces the main parameters and indices used in the model development. Table 3.2 introduces the list of decisions variables used in the DPM optimization model including binary, integer and continuous variables.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>Index of disks, $i=1,...,I$</td>
</tr>
<tr>
<td>$j$</td>
<td>Type of data fragmentation based on request arrival rate pattern – data belonging to each type share the similar request arrival rate pattern given data correlations in queries, $j=1,...,J$</td>
</tr>
<tr>
<td>$\lambda_{j,t}$</td>
<td>Hotness level of fragment type $j$ in period $t$ – hotness level is determined by the observed data request arrival rate</td>
</tr>
<tr>
<td>$k$</td>
<td>State of disk</td>
</tr>
<tr>
<td>$Sc_i$</td>
<td>Storage capacity of disk $i$</td>
</tr>
</tbody>
</table>
Table 3.1 (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_j$</td>
<td>Migration cost of fragment type $j$</td>
</tr>
<tr>
<td>$b_j$</td>
<td>Block size of fragment type $j$</td>
</tr>
<tr>
<td>$e_{d_i}$</td>
<td>Energy to spin down disk $i$</td>
</tr>
<tr>
<td>$e_{p_i}$</td>
<td>Energy to spin up disk $i$</td>
</tr>
<tr>
<td>$p_{i,k}^t$</td>
<td>Power consumption of disk $i$ at $k$ spinning state in period $t$</td>
</tr>
<tr>
<td>$\Gamma$</td>
<td>Response time penalty parameter</td>
</tr>
<tr>
<td>$\text{maxfrag}$</td>
<td>Disk maximum number of data fragments</td>
</tr>
<tr>
<td>$\lambda_{\text{max}}^i$</td>
<td>Maximum data fragment hotness level (arrival rate)</td>
</tr>
<tr>
<td>$M$</td>
<td>Maximum no. of blocks in a disk</td>
</tr>
</tbody>
</table>

Table 3.2 DPM Optimization model decision variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{i,j}^t$</td>
<td>Integer - Quantity of $j$ type fragment on disk $i$ in period $t$</td>
</tr>
<tr>
<td>$y_{j,i_1,i_2}^t$</td>
<td>Integer - Quantity of $j$ type fragments migrated from $i_1$ to $i_2$ at the end of period $t$</td>
</tr>
<tr>
<td>$s_{i,k}^t$</td>
<td>Binary - Equals to 1 if disk $i$ is in state $k$ in period $t$</td>
</tr>
<tr>
<td>$u_i^t$</td>
<td>Binary - Equals to 1 if disk $i$ should be spun up in period $t$</td>
</tr>
<tr>
<td>$d_i^t$</td>
<td>Binary - Equals to 1 if disk $i$ should be spun down in period $t$</td>
</tr>
<tr>
<td>$\tau_i^t$</td>
<td>Continuous - Response time penalty of disk $i$ in state $k$</td>
</tr>
</tbody>
</table>
3.1.1. Formulation of DPM Optimization Model for Multi-State Disks

Our objective is to minimize the energy consumption within each epoch period. The total energy consumption during an epoch consists of four elements. The first part is the primary energy consumption of the disk storage that depends on disk states (rotation speed) and number of disks spinning in each state. It is independent of the migration operations. The second part is the energy consumed during the migration time which strictly depends on the total fragment size of migration. And, the rest of energy consumption includes energy costs for disk spin-up and spin-down operations. The objective function is shown in the following equation:

$$\min \sum_{i=1}^{T} \sum_{l=1}^{I} \sum_{k} p_{i,k}^l s_{i,k}^l + \sum_{j=1}^{T} \sum_{l=1}^{J} \sum_{j_1=1}^{l} \sum_{j_2=1}^{l} \sum_{(i_2 \in I, j_2 \in J_1)} c_{j_1} y_{j,j_1,j_2}^l + \sum_{i=1}^{T} \sum_{l=1}^{I} e p_i u_i^l + \sum_{i=1}^{T} \sum_{l=1}^{I} e d_i d_i^l + \sum_{i=1}^{T} \sum_{l=1}^{I} \sum_{k} \Gamma \cdot s_{i,k}^l T_i^k \quad (3.1)$$

The physical and logical constraints in the model are as follows:

1) Size of fragments stored in a disk can never exceed the disk capacity.

2) Disks must stay in a certain state during an epoch period.

3) During any epoch $t$, there must be at least one active disk serving the data requests.

4) Any fragment can only migrate once in a certain epoch $t$.

5) A disk in stand-by mode is not considered as source or destination for data migration.

6) There is a limit for data migration time ($H$) that represents the data transfer limit for any disk within an epoch. The migration limit by default is set to half of the epoch.

The following equations represent the aforementioned constraints respectively:

$$\sum_{j=1}^{J} b_j x_{i,j}^l \leq S c_i \quad \forall i, t \quad (3.2)$$

$$\sum_{k=1}^{K} s_{i,k}^l = 1 \quad \forall i, t \quad (3.3)$$

$$\sum_{k=1}^{I} s_{i,k}^l \leq I - 1 \quad \forall t \quad (3.4)$$
\[
\sum_{j \neq i} y_{j,i,j_2}^{t} \leq x_{i,j}^{t} \quad \forall i, t, j \text{ (} i_2 \neq i \text{)} \tag{3.5}
\]

\[
y_{j,i,j_2}^{t} \leq M \cdot \sum_{k=2}^{K} s_{i,k}^{t} \quad \forall i, t, j \text{ (} i_2 \neq i \text{)}
\]

\[
y_{j,i,j_2}^{t} \leq M \cdot \sum_{k=2}^{K} s_{i,j_2}^{t} \quad \forall i, t, j \text{ (} i_2 \neq i \text{)} \tag{3.6}
\]

\[
\sum_{j \neq i} (\sum_{t_2} y_{j,i,j_2}^{t} + \sum_{t_2} y_{j,i,j_2}^{t}) \leq H \quad \forall i, t, j \text{ (} i_2 \neq i \text{)} \tag{3.7}
\]

Also, the migration equation that links \(x_{i,j}^{t}\) and \(y_{j,i,j_2}^{t}\) is:

\[
x_{i,j}^{t} + \sum_{i \neq i_1, j \neq i_2} y_{j,i_1,j_2}^{t} = x_{i,j}^{t+1} + \sum_{i \neq i_1, j \neq i_2} y_{j,i_1,j_2}^{t} \quad \forall i, t \geq 1, j \tag{3.8}
\]

The following equations are also used in the model in order to determine the binary indicating variables related to spin up and down of disks:

\[
\sum_{k} k s_{i,k}^{t} - \sum_{k} k s_{i,k}^{t+1} \leq u_{i}^{t} \tag{3.9}
\]

\[
\sum_{k} k s_{i,k}^{t+1} - \sum_{k} k s_{i,k}^{t} \leq d_{i}^{t} \tag{3.10}
\]

### 3.1.2. Two-State Optimization Model

We develop DPM optimization model for two-state disk storage system. It is easy to obtain the model formulation for two-mode (active and stand-by) disk storage by setting two values for parameter \(k\) (1 or 2) in the general formulation provided in the previous section for multi-mode disk storage. The general DPM optimization model assumes 10 levels of data popularity (hotness level) for data fragments based on the observed data request arrival rate. We believe that having 10 levels is sufficient to accurately classify data blocks based on the hotness level (if more resolution would be needed, the model can certainly have more levels that indeed reduce the MPC computational time). An important feature of two-state optimization model is that the least and the second least popular data stay in original disks. This will help to minimize the migration cost.
3.1.3. Response Time Modeling

The expected response time of a disk is a function of its spinning state and the total data arrival rate. Thus, if we consider the state of disk constant, the response time of the disk is a convex function with respect to its relative hotness level (defined in the following equation) with increasing first derivative order. This function is modeled by using piecewise linear (PWL) functions in our optimization model since they are widely used to approximate any arbitrary function (especially convex functions) with high accuracy. The input of PWL function is relative hotness of a disk. The relative hotness of a disk is calculated by following equation:

$$\lambda_{i,t} = \frac{\sum_{j} \lambda_{j,t} x_{i,j}}{\lambda_{\text{max}}^{\text{maxfrag}}} \quad (3.11)$$

where $\lambda_{i,t}$ is the relative hotness level of disk $i$ in period $t$ and $0 \leq \lambda_{i,t} \leq 1$, $0 \leq \lambda_{j,t} \leq 10$ is the popularity of fragment type $j$ in period $t$, $\text{maxfrag}$ is maximum number of fragments in a disk and is upper bound for data popularity (hotness level). We define $L$ as the number of linear functions to approximate the response time. It is well known that PWL functions can represent arbitrary functions to any accuracy by simply increasing the number of segments ($L$) to the point of desired accuracy. Thus, we verified different $L$ values for approximation of the response time convex

Figure 3.1 9-PWL function of response time model.
function. We decided to use 9-PWL function shown in Figure 3.1 for two-state disk storage system since it approximates the convex function with high accuracy.

3.1.4. Model Predictive Control (MPC)

The presented optimization model is rather static while our actual system works in a dynamic on-line environment. Therefore, we extend the model to accommodate dynamic scenarios by using model predictive control (MPC) technique to solve this issue. MPC, also known as receding horizon control (RHC) or rolling horizon control, is a form of control strategy to integrate optimization. Specifically, the current control action is obtained in an on-line fashion where, at each sampling instant, a finite horizon optimization problem (which is (3.1) -(3.10) in our context) is solved and its optimal solution in the first stage is applied as the current control decision while remaining solutions will be disregarded. Such procedure repeats along the whole control process. Therefore, all controllable variables (such as disk status and response time) for the first period are implemented in the MPC. It has been observed that MPC is a very effective control strategy with reasonable computational overhead [GP89].

The prediction information on workload arrival rate is provided to the MPC optimization model. This information plays a key role in developing an accurate underlying mixed integer program for the DPM model since any mis-prediction of data request arrival rates could cause the model to produce a solution with a less desired quality. However, as observed in many other applications of MPC, since only the first stage solution will be implemented and remaining parts will be ignored, MPC control strategy is robust to poor predictions and has a strong adjustment capability [LJ05]. Also, experimental results in Section 3.3.4.4 clearly demonstrate strong robustness of MPC integrated in the optimization model against mis-predictions.
3.1.5. Solving Strategy

Our initial attempt to find solutions to the two-state model is to implement and solve the model in the well-known CPLEX solver. The solver is installed on a server which is connected to another server running the widely used disk simulator, DiskSim [GG08], which is utilized as an accurate and reliable simulation platform by many related works. In other words, the model solution is integrated in the disk storage system simulated in DiskSim. Technical details regarding the experimental simulations are provided in Section 3.3.

3.2. Proposed Heuristic DPM Algorithm

In this section, a fast heuristic DPM algorithm is introduced that efficiently determines disk power-mode states and inter-disk data migration based on the observed I/O workload in order to maximize power saving while satisfying the response time bound. The algorithm is designed for two-state disk storage systems since, as mentioned earlier, multi-mode disks so far have not been widely commercialized.

MPC is known for its overhead when the scale of the problem grows large [CH11]. Thus, the MPC integrated in the DPM optimization model incurs significant overhead for finding its optimal solution at each epoch in large-scale disk storage systems. There are some discussions on how to reduce the MPC overhead (i.e. via hardware implementation) presented in [WM11] that are not applicable to this research. Therefore, via analyzing the optimization model to identify structural properties of the optimal solution, we propose our heuristic DPM algorithm that provides near optimal power saving solution (near to that of optimization model) with fast computational time. As shown in Section 3.3, the heuristic algorithm significantly outperforms the BLEX in terms of energy saving and response time.

As shown in the following section, our heuristic algorithm supports merge and split operation between disks (in terms of their data content) and performs analytical comparison for
decision on each operation. In merge operation, the analytical comparison is considered between two cases: merging active disks to one disk; and keeping the current status of disks. It performs the similar comparison for split operation on disks. Such analytical comparisons definitely lead to more efficient energy management in disk storage—which accounts for an advantage of the heuristic algorithm over BLEX algorithm.

3.2.1. Assumptions and Properties of the Algorithm

We use the same epoch concept (and value) described in the previous section for the heuristic algorithm implementation. Furthermore, given that transition time between disk states (usually several seconds) is much smaller than epoch length (30 minutes), we can ignore the energy consumption caused by disk state change comparing to that consumed during an epoch. Therefore, it is assumed that disk status change will not incur energy consumption/cost. Also, it is assumed that data migration will not incur extra power consumption as it only imposes extra penalty cost on response time due to fixed data migration time—which is based on the total size of the transferred data.

We define a cost function for a disk called $f$. The properties of function $f$ are as following:

1) $f$ is defined for a single disk and it consists of two parts: fixed energy cost of an active disk during an epoch, say $\text{EnergyCost}$ (equals to 0 if it is in stand-by mode); and the response time penalty cost that depends on disk hotness level. Note that the hotness level of the disk means the weighted summation of its data hotness levels—similar to equation 3.11 in Section 3.1.3.

2) It monotonically increases with respect to disk hotness level.
3) $f$ is convex function with respect to the hotness level of disk. We use the 9 piece-wise linear function described in Section 3.1.3 to represent the convex function for the response time penalty part of function $f$.

The proposed heuristic algorithm has the following features:

1) The best data migration plan happens when the data is equally distributed on active disks. Otherwise, it can be proven that the objective function value is not optimal.

2) Inter-disk data migration is performed from disk(s) supposed to be cold to disk(s) supposed to be hot so that the data migration cost is minimized.

3) Once a disk is active, its energy consumption only depends on its current state and independent of hotness level of the disk.

3.2.2. Algorithm Description

In this section, we present the fast heuristic DPM algorithm called the *sequential pairing algorithm*. First, we define two types of possible data transfer between source and destination disks. Then, we present two versions of the sequential pairing algorithm corresponding to each type of data transfer assumed for fragment migration. We consider two different possible types of data transfer between disks as follows: one-to-one transfer in which data migration can be performed only between two disks; One-to-many (or many-to-one) transfer that allows data migration from one disk to multiple disks and vice versa.

3.2.2.1. One-To-One Sequential Pairing Algorithm

The one-to-one sequential pairing algorithm in general suggests two different types of data migration: *merge-migration* that merges all data of an active disk into another active one in order
to turn off the source disk; and *split-migration* that splits the data of an active disk evenly with another inactive disk in order to turn on the destination. Figure 3.2 shows a flowchart that describes the high-level flow between different steps of this algorithm. The one-to-one sequential algorithm is as follows:

*Step 1.* Sort disks according to their hotness level in ascending order. $L$ is the sorted list.

*Step 2. Merge-migration phase:* Pick the first sequential pair from the beginning of $L$ (if there is no available pair, continue to Step 3).

2.1. If the cost function is decreased by merging two disks, perform merge-migration and update $L$ with new hotness levels of disks. Repeat Step 2.

2.2. Otherwise, check step 2.1 for the next available sequential pair in the list. If there is no disk pair available. Go to Step3.

*Step 3. Split-migration Phase:* If there is any inactive disk, pair it with the disk having the maximum hotness level in sorted list $L$. Otherwise, go to Step 4.
3.1. If the cost function is decreased by splitting the data between the pair of disks, activate the stand by disk and perform split-migration. Update the sorted list $L$ with new hotness level values of disks and Repeat Step 3.

3.2. Otherwise, there is no split-migration needed and continue to Step 4.

**Step 4. Load-balancing Phase:** Evenly distribute the data between active disks.

### 3.2.2.2. One-To-Many Sequential Pairing Algorithm

Similar to one-to-one version of the algorithm, one-to-many sequential pairing algorithm also assumes two types of migrations: *merge* and *split*. In merge-migration, it migrates all data from one source disk *evenly* to many destination disks in order to turn off the source disk. And, in split-migration, it allows migration of data from multiple source disks to one destination disk in order to activate the destination disk (many-to-one). Also, similar to the previous version of the algorithm, one-to-many sequential pairing mainly consists of two phases: merge-migration and split-migration. Under the assumption of fragment migration to/from multiple disks, we can achieve the ideal situation in both merge and split phases. In both phases, we compute the ideal data allocation plan that provides the optimal number of active disks, say $n^*$, and the average

![Flowchart for one-to-many sequential pairing algorithm.](image)

Figure 3.3 Flowchart for one-to-many sequential pairing algorithm.
hotness level, say \( \bar{v} \), over \( n^* \) disks. The ideal data allocation plan can be easily obtained from following equation:

\[
\bar{v} = \frac{\sum_{j=1}^{n} v_j}{n^*} \quad (3.12)
\]

where \( n \) is the total number of current active disks and \( v \) represents the hotness level for each disk.

Next, we perform required migrations (depending on the phase) to reach average hotness level \( \bar{v} \) on all \( n^* \) active disks. Figure 3.3 shows a flowchart that demonstrates the high-level flow between different steps of this algorithm. The one-to-many sequential pairing algorithm is described in detail in the following:

**Step 1.** Sort disks according to their hotness level in ascending order. Let us assume \( L \) is the sorted list and \( n \) is the number of current active disks.

**Step 2. Merge-migration Phase:**

1. **Optimal data allocation plan:** decrement \( n \) by one and compare the new cost function with the current one. If the cost function is not reduced, go to Step 3. Otherwise, repeat decrementing \( n \) one-by-one as long as the cost function is reduced in each decrement. \( n^* \) is determined as \( n - k \) where \( k \) is the number of past successful decrements by which the cost function was reduced. Calculate \( \bar{v} \) by using equation 3.12.

2. **Merge-migration plan:** The data on the first \( k \) disks in the list \( L \) is merged to the rest of \( n^* \) disks, say optimal list, through one-to-many merge migrations. Select the first \( k \) disks in \( L \) and add to a list say \( K \). For each disk (in ascending order of hotness level) in \( K \), perform the following one-to-many merge-migration:
2.2.1. Select the disks from the optimal list whose current hotness levels are less than \( \bar{v} \). Perform one-to-many merge-migration. Turn off the source disk. Update the hotness level of destination disks.

2.3. **Load Balancing**: Evenly distribute the data among the disks in the optimal list such that hotness level \( \bar{v} \) is reached for each disk. Then, exit the algorithm.

**Step 3. Split-migration Phase:**

3.1. **Optimal data allocation plan**: increment \( n \) by one and compare the new cost function with the current cost function of \( n \) disks. If the cost function is not reduced, exit the algorithm. Otherwise, repeat incrementing \( n \) one by one as long as the cost function is reduced in each increment. \( n^* \) is set to \( n + k \) where \( k \) is the number of successful increments by which the cost function was reduced continuously. Calculate the optimal data allocation plan (\( \bar{v} \)) by using formula given in (3.12).

3.2. **Split-migration plan**: Add \( k \) newly activated disks with hotness 0 to the beginning of the sorted list \( L \). For each newly added disk, perform the split-migration as follows:

   3.2.1. Starting from the hottest disk in \( L \), select the disks whose hotness are more than \( \bar{v} \). Perform many-to-one split migration to achieve \( \bar{v} \) on destination disk. Update the hotness levels in \( L \).

3.3. **Load Balancing**: Evenly distribute data among the original \( n \) source disks such that hotness level \( \bar{v} \) is also reached on each of these disks. Then, exit the algorithm.

As shown in next section, the proposed heuristic algorithms adapt to large-scale disk storage systems effectively since they provide fast and efficient solution in terms of both energy saving and response time that are near to optimal solution provided by the optimization model.
3.3. **Empirical Evaluation**

We have conducted extensive set of simulations using broad range of I/O workloads to validate the effectiveness of our proposed power-saving techniques in this research. We have compared our solutions in terms of energy saving ratio and average response time with those of BLEX algorithm.

**3.3.1. Simulated Disk Storage System**

The disk storage used in our simulations consists of an array of conventional hard disks. Each disk is configured as an independent unit of storage. We have simulated the array of 15 disks in *DiskSim*, a widely used disk storage simulator by many related research works. The hard disk model used in simulations is IBM Ultrastar 7K6000 [HG15]. The main specifications of this hard disk are summarized in Table 3.3.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk model</td>
<td>Ultrastar 7K6000</td>
</tr>
<tr>
<td>Standard Interface</td>
<td>SAS</td>
</tr>
<tr>
<td>Rotational Speed (active)</td>
<td>7200 rpm</td>
</tr>
<tr>
<td>Rotational Speed (stand by)</td>
<td>3600 rpm</td>
</tr>
<tr>
<td>Disk Capacity Size</td>
<td>2 TB</td>
</tr>
<tr>
<td>Seek time (average)</td>
<td>8 milliseconds</td>
</tr>
<tr>
<td>Power in active mode</td>
<td>11 W</td>
</tr>
<tr>
<td>Power in idle mode</td>
<td>7.7 W</td>
</tr>
<tr>
<td>Power in sleep mode</td>
<td>1 W</td>
</tr>
<tr>
<td>Spin up time</td>
<td>15 Seconds</td>
</tr>
<tr>
<td>Spin down time</td>
<td>Immediately</td>
</tr>
<tr>
<td>Transfer Rate (MB/Sec)</td>
<td>202</td>
</tr>
</tbody>
</table>
3.3.2. Synthetic Workload Generator

We developed a workload generator written in C to synthesize disk access workloads. We follow the well-known $b/c$ model in generating a workload of a series of random data read operations ($b\%$ of all read operations is against $c\%$ of the data) [NJ00]. It is well known that database tuple access pattern is highly skewed and can be described as an 80/20 or even a 90/10 model [TP16, OR10]. Zipf distribution is used to generate a $b/c$ model for data request arrivals. We use 80/20 as the $b/c$ model in the simulations and set the skew parameter $\theta$ of Zipf to $\log 0.8 / \log 0.2$ to reflect that. Based on the statistics collected from the real workload [OR10], the data size required by each request is set to 8, 16, 24, or 32KB with even probabilities. Each data request has the following attributes: request time, disk number, starting block number, number of blocks to read. Given that data access frequency, in database workloads, can change

![Figure 3.4 Examples of gamma distributions with different parameters used in dynamic workloads.](image)
significantly with respect to time, we simulate such dynamic changes of data request arrival rate in our synthetic workloads [ZC05]. Gamma distribution is widely used to model time-varying event arrival patterns in database systems, discrete event simulations, and web-based applications [JS12], [JR11], [KX10], [TW12], [AL99]. Thus, we have utilized gamma distributions in our workload generator to simulate request arrival rate changes over time.

Gamma distribution has two parameters: $k$, $\theta$. $k$ controls the shape and $\theta$ controls the scale of a gamma distribution. Shape parameter $k$ affects the skewness of arrival rate pattern and scale parameter $\theta$ affects the width of arrival rate distribution. The mean arrival rate is determined by mean parameter $\mu$ ($= k\theta$). We have used many arrival rate patterns (up to 42) using gamma distributions with different parameters in our dynamic workloads. Figure 3.4 shows 6 examples of gamma distributions used in the experiments. The parameters for gamma distributions are randomly chosen. The typical length of dynamic workloads is 6 hours.

3.3.3. Experimental Platform

Our model is integrated in DiskSim. The BLEX algorithm is also implemented in the disk simulator, as the comparison target, based on its description in [OR10]. We extended DiskSim source code with a multi-speed disk power model where the disk power consumption rate is proportional to disk rotation speed. Also, it is augmented with extra features such as dynamic disk spin up (and down), disk state adjustment and inter-disk data migration.

The predicted access frequency (hotness level) for each fragment type for the next $k$ epochs is provided to the model along with the observed fragment type access frequencies in the previous epoch. The prediction is performed by the prediction and autoregressive modeling methods in MATLAB. In particular, an autoregressive model is developed based on the observed
data access frequency. Then, the prediction method forecasts fragments access frequency for the
next $k$ epochs ahead based on the identified model and the observed data frequency.

3.3.4. Simulation Results and Comparisons

In this section, we describe the experimental results in terms of energy saving ratio
(normalized with no power saving scheme applied) and average response time using extensive set
of synthetic traces. The workloads are classified in two categories: dynamic and static. The data
request arrival rate changes in respect to time in dynamic workloads while it is static over time in
static traces.

3.3.4.1. Dynamic I/O Workloads

We compare the performance of DPM optimization model and heuristic algorithms with
BLEX algorithm under comprehensive set of dynamic traces whose data request arrival rate
changes over time. As mentioned earlier, we believe that BLEX algorithm cannot strongly enough
handle the dynamic aspect of I/O. The reason is that it performs inadequate data migration and

![Figure 3.5 Experimental results under dynamic workloads with different mean data arrival rates.
(a) Energy saving results; (b) Total power consumption of the disk storage system using different
power saving schemes;](image)
keeps some data in stand-by disks and therefore, to be able to adapt to dynamic changes in data request arrival rates, it might pay some penalty related to spinning stand-by disks up and down. On the other hand, MPC control scheme integrated in the optimization model effectively captures the dynamic behavior of I/O workload traces over the time. The following experimental results demonstrate this characteristic of MPC control strategy which is the key advantage of our DPM optimization model over previous proposals.

- **Energy Saving Results:** Figure 3.5 (a) shows energy saving ratio for various dynamic I/O traces with different mean data arrival rates (request/second) under all power saving schemes. Results in Figure 3.5 (a) clearly show that the DPM optimization model and the heuristic algorithms significantly outperform the BLEX algorithm. Optimization model, Heuristic-m (one-to-many) and Heurisitic-1 (one-to-one) save energy up to 60%, 58% and 54% respectively. The MPC-based optimization model outperforms BLEX with difference of at least 16% and up to 23% in energy savings. It saves 19% more energy on average than BLEX.

According to the results in Figure 3.5 (a), one-to-many sequential pairing algorithm provides power saving solutions near to that of the optimization model. It significantly outperforms BLEX algorithm with difference of at least 14% and up to 21% in energy savings. Heuristic-m saves around 16% more energy on average than BLEX algorithm. One-to-one sequential pairing algorithm also provides better power saving results than that of BLEX. In comparison with BLEX, it provides 11% improvement on energy saving on average based on the results.

Figure 3.5 (b) shows the total power consumption of the disk storage system for each power saving method compared to that of no power saving (NPS) method applied to disk storage, where all disks constantly run in active mode (shown as a red bar in the figure). It shows that DPM optimization model is dominant in power saving. One-to-many sequential algorithm outperforms
the other version as well as BLEX algorithm in reducing the power consumption. It provides energy saving near to that of the optimization model. One reason is that it always attempts to evenly relocate data among ideal number of active disks. On the other hand, as it is shown in next section, it has an overhead, however acceptable, on average response time due to additional inter-disk data migration towards load balancing.

- **Average Response Time Results:** Saving power potentially incurs increase in query response time. Thus, it is important to measure the response time effected by power saving schemes to ensure that high quality of service for queries is still maintained. This will help us in understanding the limitations of our model and algorithms.

Figure 3.6 shows the average I/O response time for all power saving schemes under several workloads with different mean arrival rates. Note that the computational time to obtain the solution for all power saving schemes is up to a second, which is apparently ignorable compared to the epoch length (30 minutes). Thus, it is excluded from the average response time computations. The results show that optimization model provides better response time than all other schemes. One
reason, other than response time considerations in its optimal power-performance tradeoff, is that it takes into account the predicted information on data access frequency in next epochs for its solutions. Both versions of heuristic algorithm provide response time results near to that of the optimization model. BLEX has longer response time than other schemes. The reason is that it performs little data migration since it only migrates blocks in cold disks to hot disk when the blocks in cold disks are accessed.

3.3.4.2. Static I/O Workloads

In this section, experimental results in terms of power saving ratio and average response time are discussed for all power saving schemes under extensive set of I/O workload traces whose arrival rate is static over the time.

- **Energy Saving Results:** Figure 3.7 (a) demonstrates the power saving results for DPM optimization problem, heuristic algorithms and BLEX algorithm. The energy saving is normalized with the energy consumption of the storage system where all disks are always active with no power saving method applied. Energy saving ratio is shown for several I/O traces with different static data arrival rates (request/second). Based on the results shown in Figure 3.7 (a), DPM optimization model is dominant in saving energy up to 72%. The optimization model, even under static traces, saves up to 8% more energy than BLEX. The heuristic algorithms also show high performance near to the optimization model in terms of power saving. Specifically, Heuristic-m algorithm closely follows optimization model in saving energy.

- **Average Response Time Results:** Figure 3.7 (b) depicts the average response time results for all schemes under static arrival rates. Note that Figure 3.7 (b), similar to response time results in previous section, excludes the computational time taken to compute solutions for each power saving method. It shows that optimization model provides the best performance comparing
to all other methods. The heuristic algorithms also demonstrate reasonable response time close to that of optimization model. Both versions of the heuristic algorithm outperform the BLEX algorithm in terms of average response time. According to the experimental results shown in Figure 3.7, although our model is designed to handle dynamics, it is also general to cover scenarios with static workloads.

3.3.4.3. Large-Scale Disk Storage Simulation

We conducted experimental simulations using extensive set of dynamic I/O workloads to evaluate the performance of our proposed ideas, especially the heuristic algorithm, in large-scale disk storage systems. Thus, we have extended the aforementioned experimental platform to simulate a 100-disk storage system. Since there is larger number of disks involved in inter-disk data migrations at the beginning of each epoch, it incurs longer time for the entire disk storage to adjust to required configurations. Therefore, we have extended the epoch length to 60 minutes to amortize the longer inter-disk data migration periods in the simulated large-scale storage system. Our experimental results showed that the computational time for finding the optimal solution by
the DPM optimization model in the large-scale storage system takes on average 20-30 minutes as expected due to the MPC overhead, as discussed earlier. However, both heuristic algorithms demonstrate fast computational time less than a few seconds that can be apparently ignorable compared to the epoch length (60 minutes). The BLEX algorithm also incurs a few-second computational time similar to that of heuristic algorithms.

- **Energy Saving Results:** Figure 3.8 (a) shows energy saving results related to all power saving methods under broad range of dynamic I/O traces with different mean arrival rates. The energy savings are normalized to the energy consumption of disk storage where no power saving method is deployed. It clearly shows that the optimization model achieves the best power saving result, however, with long computational time. According to the experimental results, one-to-many sequential pairing algorithm provides power saving solution near to optimal solution provided by DPM optimization model and saves only 4 % less energy on average than optimization model, however, with fast computational time. It also significantly outperforms the BLEX algorithm with
around 16% more energy saving on average. Heuristic-1 algorithm also provides higher energy saving ratio than BLEX and shows 9% improvement on energy saving on average compared to BLEX. Based on the experimental results, the proposed heuristic algorithms, specifically Heuristic-m, demonstrate fast and efficient power saving solutions for large-scale storage systems.

- **Average Response Time Results:** We validate the effect of power saving methods on the average response time results in the 100-disk simulated storage system. Note that the average response time results shown in Figure 3.8 (b) exclude the computational time taken to compute the solutions for each power saving method. The quantitative results for the computational time are separately reported in Figure 3.9. It depicts the measured computational time for both versions of the heuristic algorithm as well as the BLEX algorithm for I/O workload traces used in this experiment. The measurements are in terms of the average computational time of each algorithm during an I/O trace. Figure 3.8 (b) clearly shows that both heuristic algorithms provide significantly better response time than that of BLEX algorithm except only for one trace with arrival rate 100 request/sec in which BLEX has slightly better response time than Heuristic-m and

![Average Computational Time Results for Large-Scale Storage System](image_url)
slightly worse than Heuristic-1. Heuristic-m and Heuristic-1 achieve around 10 and 9 milliseconds faster response time on average than that of BLEX algorithm respectively.

Based on the experimental results in this section, the proposed heuristic DPM algorithm demonstrates near optimal power saving solutions (near to that of MPC-based optimization model) in both energy saving and average response time with fast computational time. Thus, the heuristic algorithm, especially one-to-many sequential pairing, can be integrated in large-scale disk storage systems, where finding the optimal solution might be long, to achieve efficient energy saving solutions within short periods of computational time.

3.3.4.4. MPC Robustness against Mis-Predictions

In this section, via running extensive set of experiments in the systematic way, we evaluate the robustness of MPC technique integrated in the optimization model against mis-predictions in

![Figure 3.10 MPC robustness results under different prediction error rates – set 1.](image)

**Figure 3.10 MPC robustness results under different prediction error rates – set 1.**

3.3.4.4. MPC Robustness against Mis-Predictions

In this section, via running extensive set of experiments in the systematic way, we evaluate the robustness of MPC technique integrated in the optimization model against mis-predictions in
data request arrival rates. Note that the simulated disk storage used for experiments in this section is the 15-disk storage described in Section 3.3.1. As discussed in section 3.1.4, the prediction information on workload arrival rate that is provided to the MPC-based optimization model plays an important role in accurate development of the model. Therefore, any mis-prediction in data request arrival rate could cause the model to produce a solution with less desired quality. However, MPC strategy is robust to poor predictions and has a strong adjustment capability. Only the first stage solution will be implemented, and the remaining parts will be ignored. We evaluate the aforementioned feature of MPC by running extensive set of experiments.

First, we randomly place noises (mis-predictions) in the data request arrival rate predictions for future epochs—which are fed into the optimization model as an input. Then, we gradually increase the percentage of noises in the predictions to monitor how the solution given by the optimization model deviates from the optimal solution (correct) where predictions are error free and accurate. Deviation from the error-free solution leads to a non-optimal trade-off between energy saving and response time. Note that the intensity of each single mis-prediction (difference from the correct prediction value) is randomly determined. The following plots in Figure 3.10 show the solution produced by the model against different amount of prediction errors in multiple experiments. Note that model solution on disk state configuration is represented as the number of disks to be transitioned to stand-by mode.

The results demonstrate robustness of the MPC strategy integrated in the optimization model. According to Figure 3.10 (a), 3.10 (b) and 3.10 (c), the model output under mis-predictions matches the error-free solution under error rates up to 36%, 37% and 41% respectively. For further error rates up to 93%, the model solution deviates from the optimal solution with the minimum possible difference in terms of disk configuration (one-disk difference) in all these experiments. It
deviates with 2-disk difference in terms of disk configuration in only one experiment (Figure 3.10 (c)) where prediction error rate is greater than 95%. It is important to note that the trend of results seen in the experiments in Figure 3.10 was observed in all other experiments as well.

Second, we performed another set of experiments in a systematic way to verify the effect of prediction errors of data request arrival rate on the solution of the MPC-based DPM model. Given that the model receives the prediction information for multiple future epochs, it is intuitive to consider less confidence in predictions for farther epochs. In other words, prediction error rate

![Graph](a)

![Graph](b)

![Graph](c)

Figure 3.11 MPC robustness results under different prediction error rates – set 2.

increases for farther epochs in future. Therefore, we assume a fixed error ratio limit for predictions of each epoch and this error limit is relatively larger for farther epochs ahead. More specifically, we assume that error ratio limit increases linearly with a constant slope for data access predictions.
in farther epochs. Therefore, in this type of experiment, all predictions of data access frequency for all future epochs are imposed to errors, however, with different error bounds. Note that the distribution of error rates among all predictions related to a single epoch is even distribution based on the corresponding error bound for that epoch. Predictions are generated for up to four epochs ahead in this type of experiment. The error ratio limits for future epochs are represented as an error set. As an instance, error set \([20\% - 30\% - 45\% - 67.5\%]\) represents four error ratio limits corresponding to predictions for the next four epochs in future. For example, based on this error set, prediction errors related to the first epoch in future can vary between \(-20\%\) and \(+20\%\) (from the error free prediction value) with even distribution among all predictions for this epoch. Similar to the previous experiment, we monitor the effect of mis-predictions on the quality of the solution produced by the model by applying a wide range of error sets to each experiment. The error sets vary from set \([5\% - 7.5\% - 11.2\% - 16.9\%]\) up to set \([29.5\% - 44.2\% - 66.3\% - 99.5\%]\) in each experiment.

Figure 3.11 clearly shows strong robustness of MPC optimization model against poor predictions. The model solution under errors matches the optimal error-free solution for all error sets that equal or less than the set \([20\% - 30\% - 45\% - 67.5\%]\). It deviates from optimal solution with the minimum possible difference in terms of disk configuration (one-disk difference) for all other error sets greater than set \([20\% - 30\% - 45\% - 67.5\%]\) in all experiments shown in Figure 3.11. Note that the same trend of results was observed in all other experiments of this type performed for MPC robustness verification. Based on the experimental results in this section, it can be concluded that the MPC strategy integrated in our DPM optimization model is strongly robust against poor predictions and has powerful adjustment capability. As mentioned in Section
3.1.4, the reason is that only the first stage solution will be implemented and remaining parts will be ignored.

3.3.4.5. Effect of Epoch Length on Energy Saving

We explore the effect of the epoch length on the energy saving ratio by running extensive set of experiments using various number of database dynamic I/O workloads with different arrival rate change patterns. For each particular I/O workload, a large number of different epoch lengths are chosen and the corresponding energy saving ratio are measured separately. In order to synthesize the arrival rate change pattern of I/O traces, we have used Gamma distributions (parameters $k, \theta$) as described in Section 3.3.2. The epoch length should be long enough to accommodate the disk state adjustments and data migration and also it should be short enough to

Figure 3.12 Effect of epoch length on energy saving. (a), (c): Dynamic data request arrival rate patterns; (b), (d): Effects of epoch length on energy saving ratio for (a) and (c) respectively;
capture the dynamic data request arrival rate changes. Therefore, we intuitively introduce a reasonable lower and upper bound for the epoch length as 10 minutes and 240 minutes respectively. In each experiment, the epoch length is incremented by 15-30 minutes for a particular I/O trace and the corresponding energy saving ratio is measured separately in order to observe the effect of epoch length on energy savings.

Figure 3.12 (a) shows a dynamic arrival rate pattern in a database I/O workload used in our experiments. Figure 3.12 (b) shows the effect of the epoch length on the energy saving ratios for this trace. It is observed that the energy saving ratio does not fluctuate significantly under different epoch lengths more than 30 minutes. In other words, there is no observed correlation between the epoch length and the energy saving for the epoch lengths greater than the aforementioned threshold (30 minutes). As a result, the energy saving ratio is insensitive to the epoch lengths more than 30 minutes. Figure 3.12 (c) shows another dynamic arrival rate change pattern related to a different dynamic I/O trace used in our experiments. Similar to the previous experiment, the epoch length is incremented by 15-30 minutes for this particular I/O workload and the corresponding power saving ratio is measured separately to monitor the effect of epoch length on energy savings. Figure 3.12 (d) shows that energy saving does not change significantly for the epoch length values greater than 30 minutes. In other words, similar to the previous experiment shown in Figure 3.12 (b), energy saving ratio is insensitive to the epoch length larger than 30 minutes. It is important to note that we captured the same trend of results on all other dynamic I/O workloads in our experiments. The observed effect is more noticeable in the traces whose arrival rate does not have remarkable changes over the time. Therefore, according to our experimental results, we confidently determine 30-minute long epoch as an efficient choice that is well-responsive to data request arrival rate changes and also exploits energy savings.
Chapter 4

Shared-Memory Parallel Hash-Based Stream Joins in Continuous Data Streams

In this chapter, we propose our parallel hash-based stream join algorithms on multi-core processors. First, we present our parallel design and implementation for symmetric hash join algorithm that achieves high throughput, scalability, and energy saving in performing equi-based stream joins. Section 4.1 describes parallel symmetric hash join algorithm in detail. Second, we introduce our novel parallel hash-based stream join algorithm called chunk-based pairing hash join that significantly elevates data throughput and scalability. Section 4.2 illustrates chunk-based pairing hash join. Third, we propose an in-memory parallel hash join algorithm for multi-way window joins where there are multiple input data streams involved in the join evaluation. Section 4.3 discusses our parallel multi-way hash join algorithm. Finally, Section 4.4 present the empirical evaluation for our parallel solutions in terms of scalability, latency, and energy efficiency.

4.1. Proposed Parallel Symmetric Hash Join

In this section, we present our parallelization design for symmetric hash join algorithm on multi-core CPUs to achieve high-throughput with low-latency in processing equi-based joins in data streams. Also, our fast parallel SHJ algorithm can significantly reduce the energy consumption of the systems running the stream joins since it provides significant performance speed up. We first overview the input data stream properties. Then, distribution mechanism for delivery of input tuples to processing threads is discussed. Next, we discuss the algorithmic
implementation of processing threads. Last, we summarize the properties of our parallel symmetric hash join design.

Equi-based stream join compares tuples received from two input streams \( R \) and \( S \) using equality condition on a common join attribute \( A \) as \( A_r = A_s \) where \( A_r \) and \( A_s \) are representations for the join attributes related to stream \( R \) and \( S \) respectively. It is assumed that tuples are locally ordered based on timestamp within each data stream and globally ordered across the input data streams as well. This is called synchronous data streams which is the focus of this research. While defining two input data streams, there might be an arbitrary number of sources for each input stream that each delivers tuples in timestamp order.

A major challenge to make stream joins truly scalable is to thoroughly examine the execution flow, detect the potential bottlenecks, and develop a solution to eliminate the bottlenecks. We identify two major problems in high-performance and scalable stream join execution in the presence of multi-threaded architectures: (1) we need a load balancing mechanism that evenly distribute the load among processing threads under changes on input load; (2) we need to avoid a centralized coordinator that is responsible for dispatching input tuples to processing threads as it can potentially become a bottleneck in the join execution flow. We address these problems in our solution for parallel hash-based stream join execution.

We develop an efficient method for delivering and distributing the input tuples to processing threads. We first merge tuples in \( R \) and \( S \) into a single timestamp-sorted input queue called consumption line shared among all processing threads and they can consume tuples from this available consumption line. This avoids a centralized coordinator for task distribution among processing threads.
We desire each tuple to be processed by only one single processing thread to create a disjoint parallel design that eliminates dependency among threads. Also, we want each processing thread to run a fair share (approximately \( \frac{1}{n} \) where \( n \) is number of threads) of the overall size of the consumption line in order to keep the workload balanced. To achieve these goals, we distribute \( R \) and \( S \) tuples in a round-robin fashion between processing threads. More specifically, each thread, say \( TH_i \), keeps a counter of the number of processed tuples in the consumption line and consumes a new input ready tuple from the line if \( \text{counter} \% n = i \). This setup also helps to build resilience against skewed input streams since it ensures approximately equal distribution of input tuples among threads and consequently achieves balanced work. Once a processing thread \( TH \) receives an input tuple \( t \) from stream \( R \) (symmetrically for stream \( S \)), it performs both the build and probe phase related to the equi-join evaluation for \( t \). Note that both hash tables related to streams \( R \) and \( S \) fit and reside in the main memory. First, \( TH \) adds \( t \) to the hash table for stream \( R \) particularly at the hash bucket corresponding to the join attribute value of \( t \). Second, \( TH \) probes the hash table for stream \( S \) to find matching tuples with the same join attribute from the partner stream \( S \).

Tuple expiration strategy is an important issue that affects the design of windowed algorithms for processing continuous queries over sliding windows. There are two types of strategies on this issue: eager expiration and lazy expiration. The former scans the entire window (or hash table in equi-join) upon arrival of each new tuple and removes the expired tuples. However, the latter invalidates expired tuples periodically. The eager approach has high cost while the lazy procedure needs more memory for expired tuples waiting to be removed and might also cause incorrect join results.

In our parallel symmetric hash join algorithm, we integrate a near-eager tuple invalidation strategy in the build phase for each new tuple. In particular, after a newly arrived tuple is added to
its hash bucket, it also invalidates the expired tuples from its hash bucket. We call it near-eager as it avoids scanning the entire hash table given the expensive cost for that. The error incurred by this approach is very small since high tuple arrival rate is common in data stream applications such as the common benchmark (Section 4.4) used in literature for parallel stream join performance evaluation. To avoid the effect of this small error the accuracy of join results, we also filter the output tuples if the probed tuple is observed to be expired.

Since hash tables for $R$ and $S$ are shared data structures among all processing threads, concurrent insertion accesses to hash tables should be synchronized. Each hash bucket is protected through a latch and thus, each thread should obtain the latch before insertion. Proper reader-writer locks with writer-preference type are used for buckets in hash tables to implement inter-thread synchronization.

Note that inter-thread synchronization mechanism naturally imposes an overhead on latency. However, as will be shown in Section 4.4 on experimental evaluation, the synchronization overhead on the performance of parallel symmetric hash join is reasonable and it still achieves high throughput and relatively acceptable latency. We still tackle the synchronization overhead in Section 4.2 where we propose a novel parallel design for hash-based stream join that minimizes the inter-thread synchronization overhead and significantly boosts up the throughput and scalability.

The following algorithm the steps performed by a thread to process a new tuple assigned to it in detail. Note that the mechanism discussed for invalidation of expired tuples is performed by the processing thread within the build phase and after the insertion of the tuple to the proper
hash bucket. Also, the Acquire_lock and Release_lock functions implements the reader-write synchronization mechanism with preference given to write accesses.

Our effective subtle design for parallel symmetric hash join, despite the synchronization overhead, achieves the following desirable characteristics:

1) High-throughput and speed up while providing low-latency processing in generation of output tuples. It also outperforms ScaleJoin as will be shown in Section 4.4.

2) Disjoin-parallelism in terms of independency between threads in concurrent processing of tuples.
3) Architecture-independent for multi-core processors. It also does not rely on any hardware-specific optimization feature as well.

4.2. Proposed Chunk-Based Pairing Hash Join

In this section, we introduce our novel parallel hash-based stream join algorithm called chunk-based pairing hash join that significantly boosts up throughput compared to the parallel symmetric hash join while reducing latency as well through minimizing the inter-thread synchronization overhead imposed by concurrent access to shared data structures (hash tables) in memory.

A significant part of contention over hash table buckets is for when threads in need of read-only access during their probing phase need to wait too long on threads with write-access request for their build phase. Motivated by this fact, in our parallel design, we tackle at generalization of hashing procedure to two groups of tuples as chunks instead of individual tuples so that we can separate probe-related accesses from those of build phase. In particular, we divide the tuples within each sliding window for input streams into multiple chunks. Once two new chunks of tuples arrive in sliding windows for R and S streams, the chunk-based pairing hash join algorithm performs two group-based steps on the newly arrived chunks as follows:

1) All processing threads enter to chunk-based build phase and populate both hash tables for R and S by inserting newly arrived tuples into buckets.

2) After synchronization through barriers, all processing threads enter the chunk-based probe phase and find matching joins for each tuple in the chunks by probing the hash table for the partner stream in a symmetric fashion.
Figure 4.1 shows chunk-based pairing hash join and its two phases. The algorithm performs the same procedure on the next two incoming chunks of tuples arriving in the sliding windows. In the group-based build phase, there is an unavoidable contention over latches among threads for tuple insertion and invalidation actions. As all threads in the group-based build phase need write-access to hash tables, the latches are implemented as normal locks on hush buckets to synchronize their concurrent access. Each thread needs to acquire the lock to be able to perform its actions and
release it after it is done. In contrast, in the group-based probe phase, all processing threads need to probe hash tables and can freely access hash tables at the same time with no synchronization. In other words, the significant portion of synchronization overhead mentioned earlier is now eliminated in our novel parallel design. This will minimize the overall synchronization overhead in hash-based stream join to a negligible amount and also significantly increases throughput.

The only drawback is that arrival of new tuple chunks incurs a latency in join evaluation and output generation. However, as will be show in the next section, despite this latency, the response time for chunk-based pairing is shorter than parallel SHJ for large number of threads and also shorter on average considering all thread number spectrum in our experiments. Also, It achieves significantly higher throughout compared to parallel SHJ.

In our implementation, we consider fixed-size chunks in terms of number of tuples. We address this parameter as $\lambda$. Note that length of chunks in terms of time for two data streams $R$ and $S$ can be different if they have different patterns for tuples arrival rate. The size of chunk should be large enough to exploit the benefit of group-based probing phase while small enough to avoid too much latency on average response time. We have experimented wide range of values for $\lambda$ to figure out a fixed size that efficiently achieves this desired feature. We have set the chunk size to 300 tuples in our implementation for this algorithm.

Note that the distribution mechanism for delivering the build or phase related tasks among the threads in chunk-based build and probe phases is the same round-robin schema as explained in previous section for parallel symmetric hash join. The only difference is that an assignee thread only performs the build or probe task of the given input tuple during a group-based phase. The tuple expiration policy is the same as the strategy used for parallel SHJ.
As mentioned earlier, chunk-based pairing hash join elevates performance of our parallel symmetric hash join in terms of both throughput. Also, it provides other desirable parallelization properties discussed in Section 4.1 for parallel symmetric hash join which are disjoin-parallel, architecture independent, and hardware-specific optimization independent.

4.3. Proposed Parallel Multi-Way Hash Join

In this section, we present our parallel design and implementation for multi-way hash-based stream join processing on multi-core processors. First, we discuss the methods for delivering and distributing input tuples from multiple data streams to processing threads. Second, we describe the algorithmic implementation of processing threads. Then, we discuss the properties of the parallel multi-way hash join design.

The tuples inside of each data stream are locally ordered based on timestamp. Also, it is assumed that input tuples across the multiple streams are globally ordered in respect to timestamp. This is called synchronous data streams meaning that all tuples arrive with increasing value of the timestamps. On the other hand, there is another type of data streams as asynchronous data stream in which tuples across all data streams are unordered while they are locally ordered within each stream based on timestamp. The latter type has its own challenges and multi-way join algorithms which is out of the scope of this research.

Similar to our parallel symmetric hash join algorithm, we first merge arriving tuples across all input data stream into one single queue of input tuples shared among all processing threads. This gate-in process avoids a centralized coordinator for threads and eliminates the well-known bottleneck overhead related to it. Also, as another reason, this merge process will enhance creating a balanced distribution of tuples among threads as will be described next. Processing threads are able to process tuples from the shared line of merged input tuples. Each thread has a unique
identifier and keeps a counter for total number of tuples arrived in the gate-in line and increments it once a new tuple is added to the line. Each newly arrived tuple is processed only by one process whose identifier equals to modular division of the counter value over total number of threads. This implements round-robin distribution of data stream items among processing threads and thus ensures balanced work among threads. As a result, this parallel design builds skew resiliency against fluctuating or bursty data streams and keeps the workload balanced.

All hash tables of data streams reside in the main memory. Once a tuple arrives in the merged queue line, the assignee thread performs all steps related to its join evaluation. In other words, there is independency among threads and the parallelism is therefore disjoint which is a desired feature for parallel paradigms. First, as the build phase, the thread refers the hash table of the origin stream to insert the input tuple to the proper hash bucket based on the tuple’s join attribute value. Second, in the probe phase, the assignee thread probes hash tables for all other data streams based on the join ordering heuristic to find the matching tuples and generates the output tuples.

As mentioned earlier, the original join ordering heuristic orders the joins in ascending order of intermediate tuples (bucket sizes) in consecutive binary joins - to leave as little work as possible for inner loops. This means that the stream to which the new tuple arrives is ordered first followed by the stream with the smallest selected bucket size, and so on. In our parallel implementation for multi-way hash join, we relax the join order heuristic to reduce its time complexity. Instead of sort operation to achieve the desired join ordering, we choose the stream whose probed hash bucket for equi-join has the largest size and put it in the inner-most for-loop. The newly arrived tuple is similarly located in the outer-most for-loop. The other streams can be chosen in any order for the rest of the intermediate for-loops. The resulting join order from our heuristic has the same cost as
the join order with strictly ascending order of bucket sizes since the total number of times the largest bucket (inner-most for-loop) is scanned is the same. However, our new heuristic has linear time complexity compared to the original heuristic with time complexity of sorting operation \((n \log(n))\).

Similar to parallel binary hash join algorithms discussed in this section, we need to protect hash tables through latches in our parallel multi-way join design in order to handle contention over these shared data structures – similarly reader-writer locks with writer-preference type used on buckets for this purpose. However, note that the inter-thread synchronization overhead in the context of multi-way stream join is less than that of two-way hash join because the number of shared hash tables among threads are relatively more. Also, in multi-way join context, the total number of read accesses to hash buckets is more than that of write accesses as the probe phase requires more than one hash bucket to be read. Thus, this incurs less overall synchronization latency since relatively more threads in need of read-only accesses can concurrently probe the hash buckets with no synchronization.

In order to help reduce the overall synchronization overhead in our parallel multi-way join hash algorithm, we make a subtle modification to the tuple expiration policy of the original eager multi-way join. As mentioned earlier, a newly arrived tuple invalidates the expired tuples from corresponding hash buckets of all other streams when it enters the probe phase. Instead, we incorporate tuple expiration in the build phase. In particular, after a newly arrived tuple is inserted to its hash bucket, it also invalidates the expired tuples from the hash bucket of its origin stream. This keeps the probe phase as a read-only-access step and thus help with less synchronization latency. Our tuple expiration strategy might incur a small error in a specific scenario in which an
expired tuple is still not invalidated by a new tuple from its own stream and a tuple from other streams starts probing this expired tuple. In order to avoid the error for this scenario, we filter the expired tuples from the output join tuples in the probe phase.

Algorithm 2 shows how multi-way hash join is computed by an assigned thread when a new tuple \( t \) arrives at stream \( i \). We use the notation \( B_{i,t} \) to represent the hash bucket in the \( i^{th} \) window to which join attribute of tuple \( t \) maps. Window size for \( i^{th} \) window is represented as \( W_i \) and \( ts \) represents timestamp for tuples. We assume there are \( n \) streams \( S_1, ..., S_n \) participating in the multi-way hash join. We define the global join order as \( S_1 \bowtie (S_2 \bowtie ... \bowtie S_n) \) and represent it as \((S_1, ..., S_n)\).

The proposed multi-way hash-based join algorithm provide high-throughput and low-latency processing of equi-based join evaluation in continuous data streams. Also, similar to our parallel binary hash-based stream joins introduced in this chapter, the parallel multi-way hash join is disjoin parallel, architecture independent, and hardware-specific optimization independent.

4.4. Empirical Evaluation

In this section, we present our experimental evaluations that demonstrate the effectiveness of our proposed solutions in this research on parallel processing of equi-based stream joins. We
first introduce the experimental setup including the multi-core architecture and the benchmark used in our experiments. We evaluate our proposed parallel hash-based stream join algorithms in terms of throughput and scalability as well as latency. We study scalability of our proposed algorithms in terms of number of tuples processed per second. We continue by measuring the processing latency in terms of average end-to-end response time for input tuples. As will be shown, our proposed parallel algorithms in previous section achieve high-throughput, significant scalability, and low-latency in equi-join evaluation in data streams. Our two-way parallel joins outperform the best existing parallel stream join algorithm, ScaleJoin, in terms of throughput and scalability.

We have also compared our two-way parallel hash join algorithms (parallel SHJ and chunk-based PHJ) in terms of scalability and latency with those of the state-of-the-art parallel stream join in literature called ScaleJoin. We side-by-side compare our C++-written hash algorithms with C++-based implementation of ScaleJoin in our experimental platform which is introduced in the next section.

4.4.1. Experimental Setup

We follow the common benchmark used by previous work on parallel two-way stream join processing for our empirical evaluation [GN15], [RT14], [TM11], [GB09]. $R$ tuples consists of attributes $< ts,x,y,z >$ and $S$ tuples composed of attributes $< ts,a,b,c,d >$. $ts$ stores the timestamp for a tuple which shows its generation time. The tuples are locally ordered within each stream and globally ordered across the input streams based on their timestamp. Attributes $x,y,z$ are of types int, float and char respective, and attributes $a,b,c,d$ are of types int, float and double and bool respectively. Values for attributes $x,y,a,b$ are derived from the uniform distribution on the interval $[1-10000]$. For each pair of tuples $t_R,t_S$, if the equality condition on the integer
attributes $t_R.x = t_S.a$ is satisfied, then an output tuple $t_O$ is generated combining $t_R$ and $t_S$ and setting timestamp of $t_O$ to maximum between $t_R.ts$ and $t_S.ts$. The tuple injection rate is the same for the two input streams and equals to 500 tuples per second. According to the benchmark, the tuple injection rate is steady and window size is fixed-size during the course of running stream joins.

As for the multi-way join set up, we have considered four input streams for multi-way hash join evaluation over a common join attribute. Since there is no such benchmark for multi-way window joins, we also use the aforementioned benchmark to evaluate our proposed parallel multi-way hash join algorithm. Our proposed parallel algorithms are evaluated on a system equipped with a card of type Intel Xeon Phi Coprocessor 5110P. Our experiments are executed in native mode on the Phi card. The detailed specification of this hardware is presented in Table 4.1.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Collection</td>
<td>Intel® Xeon Phi™ x100</td>
</tr>
<tr>
<td>Processor Number</td>
<td>5110P</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>60</td>
</tr>
<tr>
<td>Processor Base Frequency</td>
<td>1.05 GHz</td>
</tr>
<tr>
<td>Cache</td>
<td>30 MB L2</td>
</tr>
<tr>
<td>Max Memory Size</td>
<td>8 GB</td>
</tr>
<tr>
<td>Max # of Memory Channels</td>
<td>16</td>
</tr>
<tr>
<td>Max Memory Bandwidth</td>
<td>320 GB/S</td>
</tr>
<tr>
<td>Instruction Set</td>
<td>64-bit</td>
</tr>
<tr>
<td>Intel Turbo Boost Technology</td>
<td>No</td>
</tr>
<tr>
<td>Instruction Set Extensions</td>
<td>IMCI</td>
</tr>
</tbody>
</table>
4.4.2. Throughput and Scalability Evaluation

Similar to the previous state-of-the-art research works, we first assess the scalability of parallel hash-based stream join algorithms for two streams \( R \) and \( S \) (each having one source) over increasing number of threads and for different widow sizes. Window sizes are 5, 10 and 15 minutes. The number of threads varies from 1 (non-parallel run) up to 80 threads. We measure the throughput in terms of number of tuples processed per second.

- **Parallel Symmetric Hash Join Scalability**: Figure 4.2 (a) shows throughput results for parallel symmetric hash join algorithm. Parallel SHJ achieves significantly high-throughput rate up to approximately 60,000 t/s, 48,000 t/s and 33,000 for window sizes of 5, 10 and 15 minutes respectively. Despite the fact that inter-thread synchronization overhead grows with increasing number of threads, parallel symmetric hash join still demonstrates almost perfectly linear scalability up to 50, 45 and 30 processing threads for window sizes of 5, 10 and 15 minutes respectively. After the aforementioned number of threads, throughput declines as synchronization overhead increases.

![Figure 4.2 Scalability evaluation under different window sizes and increasing number of threads. (a) Parallel symmetric hash join; (b) Chunk-based pairing hash join algorithm;](image)
overhead becomes too large and effects the throughput rate negatively. However, as mentioned earlier, parallel SHJ holds a great scalability up to 50 threads.

- **Chunk-Based Pairing Hash Join Scalability:** Results for our chunk-based design are shown in Figure 4.2 (b). The proposed algorithm obtains significantly high throughput rate up to approximately 71,600, 56,100, and 39,300 tuples per second for window size 5, 10 and 15 minutes respectively. Also, chunk-based pairing hash join achieves perfectly linear scalability for up to 65, 60 and 50 number of threads. Thus, it boosts up both the throughput and scalability compared to parallel symmetric hash join by minimizing inter-thread synchronization overhead. After the aforementioned number of threads, there is a decline, however slightly, in throughput as synchronization overhead becomes very high and unavoidable.

- **Comparative Results for Scalability:** Figure 4.3 depicts comparative results in terms of scalability and throughput between parallel symmetric hash join (SHJ), chunk-based pairing hash join (PHJ), and ScaleJoin. The window size is 5 minutes in this experiment. The difference in terms of throughput between chunk-based PHJ and parallel SHJ increases almost linearly with increasing number of threads up to 50. with 50 processing threads, chunk-based join achieves approximately

![Figure 4.3 Comparisons in terms of scalability and throughput. Parallel SHJ vs. Chunk-based PHJ vs. ScaleJoin](https://example.com/figure4.3.png)
66,300 t/s as throughput rate while parallel SHJ sustains rate of 59,800 t/s, thus resulting in around 6500 more tuples processed per second by chunk-based pairing hash join. A significant throughput difference is observed when the number of threads exceeds 50. After this point, although very high inter-thread synchronization overhead kicks in parallel SHJ, chunk-based pairing hash join remains unaffected and even keeps increasing the throughput rate under large number of threads up to 65. At this point, it achieves a significantly high throughput rate of 71,600 tuples processed per second. After 65 threads, an inevitable and very high inter-thread synchronization overhead arises that even lowers the throughput rate for our chunk-based design. However, even under this high synchronization overhead, chunk-based pairing hash join shows gradual and slight decrease in throughput results such that it still obtains significant throughout rate of approximately 63,100 tuples per second for 80 threads. According to the analytical results in Figure 4.3, our proposed chunk-based pairing join, as mentioned earlier in previous sections, significantly boosts up throughput and minimizes the effect of inter-thread synchronization overhead through its novel chunk-based pairing design.

Since ScaleJoin is the state-of-the art and the best parallel stream join algorithm (nested loops-style join), we also compare our proposed ideas on parallel two-way hash-based stream join processing with those of ScaleJoin in terms of scalability and throughput. Chunk-based pairing hash join and parallel SHJ both outperform ScaleJoin significantly in terms of achieved data throughput rate.

The difference between parallel SHJ and ScaleJoin increases linearly as well in a significant way up to 50 threads. For 50 threads, parallel SHJ achieves approximately 11.5 times more throughput rate than that of ScaleJoin. In spite of a decline in throughput for parallel SHJ after 50 threads, it still outperforms ScaleJoin. Even for 80 threads, parallel SHJ sustains 2,000
more processed tuple per second compared to ScaleJoin although ScaleJoin has its maximum achieved throughput rate at 80 threads.

- **Parallel Multi-Way Hash Join Scalability**: We examine throughput and scalability of our parallel multi-way hash join algorithm under different sliding window sizes and increasing number of processing threads. Figure 4.4 demonstrate high throughput and great scalability for the parallel multi-way hash join. It achieves high throughput up to around 6,700, 8,150, and 10,550 tuples per second for window size 15, 10, and 5 minutes respectively. With increasing number of threads, this algorithm shows great scalability for large number of threads up to around 60, 65, and 75 threads for window size 15, 10, and 5 minutes respectively.

**4.4.3. Latency Evaluation**

Low-latency plays an important role in stream join evaluation in time-sensitive applications, such as option pricing, that can tolerate latency only in the magnitude of several seconds. To this end, we measure the average end-to-end latency for tuples under each number of processing threads for throughput results shown in the previous section. Also, it is important to measure average response time for our chunk-based pairing hash join algorithm since formation of new chunks of tuples imposes an overhead in terms of latency. However, this overhead is countervailed
with the fact that chunk-based design performs the build and probe phases much faster than parallel SHJ. This results in even shorter response time compared to parallel SHJ for large number of threads. This is another advantage of chunk-based design in addition to boosting data throughput.

Figure 4.5 shows the end-to-end latency for chunk-based pairing hash join, parallel SHJ and ScaleJoin algorithms under different number of processing threads for the window size of 5 minutes. As it is shown, ScaleJoin has less latency than our proposed ideas. The reason is that it performs nested loops-style join and thus there is no contention over any shared data structure that might impose synchronization overhead on latency. On the other hand, our parallel hash-based stream join algorithms by nature need to share hash tables between threads and thus contention over these resources is inevitable and imposes an overhead on latency. Our proposed hash-based joins still have latency less than 1.5 seconds which is still acceptable and considered low even for time-sensitive stream applications while they achieve significantly higher throughput and scalability. Note that under ascending number of threads, the latency increases linearly for all algorithms in Figure 4.5. The reason for our parallel hash join algorithms is that the synchronization overhead on latency grows as number of threads increases. The reason for ScaleJoin is that the output rate per thread decreases as number of threads grows and this results
in longer time for output tuples to become ready and released by the output gate architecture of ScaleJoin [GN15]. Observe that chunk-based algorithm has lower latency compared to parallel SHJ for large number of threads since as mentioned earlier, synchronization has less impact on it thanks to its chunk-based design and synchronization-free probe phase. Chunk-based algorithm and parallel SHJ keeps the response time lower than 820 and 1,430 milliseconds respectively even under large number of threads as shown in Figure 4.5.

Figure 4.6 Latency evaluation of parallel schemas under different intensities of input streams. Chunk-based paring hash join (CHJ), Parallel symmetric hash join (PSHJ) and ScaleJoin.

Figure 4.7 Latency evaluation of parallel multi-way hash join algorithm. The window size is 5 minutes.
It is also important to measure latency under intense workloads to evaluate how parallel schemes scale-up to handle large inputs. To this end, we have conducted an experiment to measure response time under intense input data streams where the tuple injection rate grows very large. Figure 4.6 shows the results for this type of latency evaluation for chunk-based pairing hash join, parallel SHJ and ScaleJoin. Our parallel hash-based join algorithms show significantly better latency for very intense workloads compared to ScaleJoin which has exponential jump in latency for input rates more than $10,000$. As shown in Figure 4.6, chunk-based pairing hash join can scale-up to intense workload in terms of latency better than parallel SHJ and shows lower response time given its faster hash join evaluation.

Figure 4.7 shows the average end-to-end latency measurement related to our parallel multi-way hash join algorithm. It clearly shows low-latency for the given spectrum of processing threads. As mentioned earlier, our parallel design attempted to decrease the inter-thread synchronization existing in the multi-way hash join evaluation through a subtle modification on tuple expiration policy as explained in Section 4.3.

4.4.4. Energy-Efficiency Evaluation

As mentioned earlier, the contribution of this research on parallel processing of streams joins is two-fold. The first contribution, as the main claim of this research, is to achieve high throughput and scalability with low latency in evaluation of costly stream join operations. The second contribution is to obtain energy saving in running stream joins through significant computational speed up provided by our fast parallel hash join algorithms. Thus, we have also measured the energy consumption under our proposed parallel algorithms to observe the effect of our high-performance computing ideas on the energy consumption during equi-stream join
execution. In particular, we measure the energy consumption under increasing number of threads (cores) and benchmark with ScaleJoin as the baseline for energy-efficiency evaluation.

We continuously measure the power consumption during the course of running the stream joins. Fluctuations of power are observed in all the experiments – this is due to the different hardware activities at different times of the join. We measure average active power consumption for energy calculations. Note that active power is defined as the difference between recorded system power while processing the workload and that when the system is idle.

Figure 4.8 demonstrate the energy consumptions of parallel SHJ, chunk-based PHJ, and ScaleJoin under increasing number of threads. It clearly shows that both of our proposed parallel hash join algorithms significantly outperforms ScaleJoin in energy efficiency. All parallel algorithms in this figure show decreasing trend in energy consumption as number of threads grow due the speed up achieved through parallelism.
Chapter 5

Summary and Future Works

Power consumption has increased greatly in data centers. Database management system (DBMS) is one of the most important servers in software stack deployed in data centers. Data storage system, as an important element of database, has been addressed by many research efforts towards making it green and energy-efficient. In this dissertation, we presented our research ideas on designing a dynamic energy-aware disk storage system in database servers. We improved on the limitations of the previous work. We introduced a DPM optimization model extended with MPC strategy that can be adapted to disk-based storage systems with dynamic I/O workloads. Also, a fast-solution heuristic DPM algorithm is presented that can be integrated in large-scale disk storage systems, where finding the optimal solution might be time consuming, to achieve fast and efficient power-saving solutions. We evaluated our proposed ideas by running simulations using extensive set of synthetic workloads. The experimental results showed that our solutions achieve up to 1.65 times more energy savings while providing up to 1.67 times better query response time compared to the BLEX algorithm.

Stream join is a dynamic and costly database operation that performs real-time join evaluation on continuous data streams. Stream joins, also known as sliding-window joins, have high computational time and potentially consume more energy compared to other database operations. We tackle energy-efficiency of stream join algorithms in this research. Given that there is a strong linear correlation between energy-efficiency and performance of shared-memory
parallel join algorithms in database servers, we study parallelization of stream join algorithms on multicore processors to achieve energy efficiency and high performance. Equi-join is the most common type of join in query workloads and symmetric hash join (SHJ) algorithm is the best algorithm to evaluate equi-joins in data streams. To best of our knowledge, we are the first to propose a shared-memory parallel symmetric hash join algorithm on multi-core CPUs. Also, in this research, we introduce a novel parallel hash-based stream join algorithm called chunk-based pairing hash join. We also tackle parallel processing of multi-way stream joins where there are more than two input data streams involved in the join operation. To best of our knowledge, we are also the first to propose an in-memory parallel multi-way hashing-based stream join on multicore processors. Empirical evaluation shows that our proposed parallel algorithms achieve high throughput, great scalability, and low latency while achieving the best energy-efficiency for systems running stream joins. PSHJ and CPH algorithms achieve up to 11 times and 12.5 times more throughput, respectively, compared to that of ScaleJoin. Also, these two algorithms provide up to around 22 times and 24.5 times more throughput, respectively, compared to that of non-parallel (sequential) stream join computation where there is one processing thread.

As a future work, other database operators in streaming environment can be explored for parallelization on multicore processors. As another future work, implementation of our parallel algorithms can be extended to large-scale distributed streaming environments. Finally, an entire system can be designed and implemented that can process all queries on streams in parallel fashion.
References


[GG08] http://www.pdl.cmu.edu/DiskSim/


[LJ05] http://cepac.cheme.cmu.edu/pasilectures/lee/LecturenoteonMPC-JHL.pdf


Appendices
Appendix A. Copyright Permissions

Below is the permission for use of figures from [GN15] in Chapter 2.

<table>
<thead>
<tr>
<th>Thesis / Dissertations Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:</td>
</tr>
<tr>
<td>1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.</td>
</tr>
<tr>
<td>2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.</td>
</tr>
<tr>
<td>3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author’s approval.</td>
</tr>
</tbody>
</table>

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication].

2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis online.

3) In placing the thesis on the author’s university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity’s name goes here]’s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.
Below is the permission for use of figures from [BT13] in Chapter 2.
Below is the permission for use of figures from [BA13] in Chapter 2.
Below is the permission for use of material from [BY16] in Chapter 3.
Below is the permission for use of material from [BT17] in Chapter 3.

---

Your policy on re-use permission
2 messages

Peyman Behzadnia <pbehzadn@mail.usf.edu>  Fri, Feb 23, 2018 at 12:40 AM

To: secretary@aircse.org

To whom it may concern,

I have published a paper in International Journal on Database Management Systems (IJDMS) by AICCC Publishing Corporation. I would like to re-use the content of my journal in my PhD dissertation. Please let me know how I can obtain the re-use permission for this purpose? Thank you.

Best,
Peyman Behzadnia

secretary@aircse.org <secretary@aircse.org>  Fri, Feb 23, 2018 at 2:17 AM

Reply-To: secretary@aircse.org

To: Peyman Behzadnia <pbehzadn@mail.usf.edu>

Hi there,

Good day! You are free to reuse the contents of your paper for your dissertation. No worries.

Thanks!
Best Regards,

John Linton
+61 4 65813561

************************
Secretary
AIRCC Publishing Corporation
http://www.aircse.org
http://www.aircse.com

************************

From: Peyman Behzadnia <pbehzadn@mail.usf.edu>
To: secretary@aircse.org
Sent: Friday, February 23, 2018 4:46 PM
Subject: Your policy on re-use permission

[Quoted text hidden]
About the Author

Peyman Behzadnia completed his BSc in Computer Software Engineering at Iran University of Science and Technology (IUST), Tehran, Iran, in 2010 and his MSc in Computer Science at the University of South Florida, Tampa, Florida in 2012. He is currently a PhD candidate in Computer Science and Engineering at the University of South Florida. His research interests include database management systems, high-performance computing, and data mining.