A HYBRID DATA PREFETCHING ARCHITECTURE FOR DATA-ACCESS EFFICIENCY

BY

YONG CHEN

DEPARTMENT OF COMPUTER SCIENCE

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the Illinois Institute of Technology

Approved _________________________

Adviser

Chicago, Illinois
July 2009
© Copyright by

YONG CHEN

July 2009
ACKNOWLEDGEMENT

Ph.D. study is a long but fruitful journey for me and my family. I have many people to thank for their consistent support and help.

First of all, I am so grateful to my advisor Professor Xian-He Sun for his valuable guidance and insightful advice. I sincerely appreciate his suggestions, very helpful comments and constant encouragement.

I would like to sincerely thank Professor Zhiling Lan for her valuable suggestions on improving my work and her consistent encouragement. I would like to thank Professor Gady Agam and Professor Mark Anastasio for their interest in my research and taking their valuable time to serve on my thesis committee. I sincerely appreciate their efforts.

I am grateful to my colleagues and research collaborators, Dr. Surendra Byna, Dr. Ming Wu, Dr. Rajeev Thakur and Professor William Gropp. I thank them for their valuable support and help on improving my research work. It is my great pleasure to have them with my Ph.D. journey.

I would also like to acknowledge that the research study in this dissertation was supported in part by National Science Foundation, ACM/IEEE High-Performance Computing Fellowship, and Illinois Institute of Technology Fieldhouse Fellowship.

Finally, I thank my wife, my parents and my son for their invaluable and constant support, encouragement and understanding.

I thank everyone who made my Ph.D. journey enjoyable and memorable.

Y.C.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENT</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>x</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Growing Gap Between Computing and Data Access</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Memory Hierarchy Model</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Data Prefetching and Limitations</td>
<td>7</td>
</tr>
<tr>
<td>1.4 Hybrid Adaptive Prefetching Architecture Solution</td>
<td>9</td>
</tr>
<tr>
<td>1.5 Dissertation Organization</td>
<td>10</td>
</tr>
<tr>
<td>2. DATA ACCESS: BOTTLENECK OF COMPUTING</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Modeling Scalability of Emerging Multicore Architecture</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Modeling Scalability of Parallel and Distributed Architecture</td>
<td>26</td>
</tr>
<tr>
<td>2.3 Data Access: A Bottleneck of Scalable Computing</td>
<td>34</td>
</tr>
<tr>
<td>3. HYBRID ADAPTIVE PREFETCHING ARCHITECTURE</td>
<td>36</td>
</tr>
<tr>
<td>3.1 Hybrid Adaptive Prefetching Architecture</td>
<td>36</td>
</tr>
<tr>
<td>3.2 Cache-Memory Latency Reduction: A Hardware Approach</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Memory-Disk Latency Reduction: A Software Approach</td>
<td>39</td>
</tr>
<tr>
<td>3.4 Integration of Hardware and Software Approach</td>
<td>41</td>
</tr>
<tr>
<td>4. IMPROVING CACHE-MEMORY STAGE DATA-ACCESS</td>
<td>43</td>
</tr>
<tr>
<td>4.1 Data-Access History Cache Design and Methodology</td>
<td>44</td>
</tr>
<tr>
<td>4.2 DAHC-based Data Prefetching Mechanism</td>
<td>49</td>
</tr>
<tr>
<td>4.3 Adaptive Hardware Data Prefetching</td>
<td>55</td>
</tr>
<tr>
<td>4.4 Simulation Methodology</td>
<td>65</td>
</tr>
<tr>
<td>4.5 Experimental Results and Performance Analysis</td>
<td>69</td>
</tr>
<tr>
<td>4.6 Application and Impact</td>
<td>79</td>
</tr>
</tbody>
</table>
5. IMPROVING MEMORY-DISK DATA-ACCESS EFFICIENCY .......... 81
   5.1 MPI, MPI-IO and Parallel I/O .................................................. 81
   5.2 Pre-execution Based I/O Prefetching ...................................... 85
   5.3 Post-execution Based I/O Prefetching .................................... 99
   5.4 I/O Prefetching and Caching Library Support ...................... 106
   5.5 Experimental Results and Performance Analysis .................. 110
   5.6 Application and Impact .......................................................... 121

6. RELATED WORK ............................................................................. 123
   6.1 Data Prefetching at Cache-Memory Level ............................. 123
   6.2 Data Prefetching at Memory-Disk Level ............................... 127

7. CONCLUSION AND FUTURE WORK ............................................ 131
   7.1 Research Contributions .......................................................... 131
   7.2 Future Work ........................................................................... 134

BIBLIOGRAPHY .................................................................................... 136
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Simulator Configuration</td>
<td>69</td>
</tr>
<tr>
<td>4.2 Simulation Results for Matrix Multiplication</td>
<td>70</td>
</tr>
<tr>
<td>4.3 Primary Prefetching Algorithms Selected Adaptively at Runtime</td>
<td>77</td>
</tr>
<tr>
<td>5.1 Aggregate Sustained Bandwidth on NFS and PVFS</td>
<td>115</td>
</tr>
<tr>
<td>5.2 Aggregate Sustained Bandwidth of 5 by 5 Tiles 2D-Convolution on PVFS</td>
<td>116</td>
</tr>
<tr>
<td>5.3 Aggregate Sustained Bandwidth of 10 by 10 Tiles 2D-Convolution on PVFS</td>
<td>116</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Performance Improvement Trend of Processor, Memory and Disk</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Memory Hierarchy Model</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Data Prefetching: Fetching Data in Advance</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Symmetric Multicore Processor</td>
<td>16</td>
</tr>
<tr>
<td>2.2 Fixed-size Speedup of a Multicore Architecture</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Fixed-time Speedup of a Multicore Architecture</td>
<td>20</td>
</tr>
<tr>
<td>2.4 Memory-bounded Speedup of a Multicore Architecture</td>
<td>22</td>
</tr>
<tr>
<td>2.5 Fixed-size, Fixed-time and Memory-bounded Speedup of a Multicore Architecture</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Overview of Hybrid Adaptive Prefetching Architecture</td>
<td>38</td>
</tr>
<tr>
<td>3.2 Hybrid Adaptive Prefetching Architecture: A Systematic Solution to Improving Data-Access Efficiency</td>
<td>42</td>
</tr>
<tr>
<td>4.1 DAHC General Design and High-level View</td>
<td>45</td>
</tr>
<tr>
<td>4.2 DAHC Blueprint: PC Index Table, Address Index Table and DAH table</td>
<td>47</td>
</tr>
<tr>
<td>4.3 DAHC Snapshot</td>
<td>48</td>
</tr>
<tr>
<td>4.4 Data-Access History Cache Serving for L1 Data Cache</td>
<td>49</td>
</tr>
<tr>
<td>4.5 Reference Prediction Table and State Transition Diagram</td>
<td>51</td>
</tr>
<tr>
<td>4.6 Markov Prefetching Correlation Table and State Transition Diagram</td>
<td>52</td>
</tr>
<tr>
<td>4.7 Markov Prefetching with DAHC</td>
<td>54</td>
</tr>
<tr>
<td>4.8 Example of Difference Table</td>
<td>55</td>
</tr>
<tr>
<td>4.9 Prefetching Algorithm Performance Table</td>
<td>64</td>
</tr>
<tr>
<td>4.10 Enhanced SimpleScalar Simulator</td>
<td>68</td>
</tr>
</tbody>
</table>
5.18 Bandwidth of BTIO Reads with Prefetching on NFS................................. 121
5.19 Bandwidth of BTIO Reads with Prefetching on PVFS .............................. 122
ABSTRACT

High-performance computing has crossed the Petaflop mark and has been moving forward to reach the Exaflop range. However, while computing resources are making rapid progress, there is a significant gap between processing capacity and data-access performance. Due to this gap, although processing resources are available, they have to stay idle waiting for data to arrive, which has a severe impact on the overall system performance. In the meantime, applications tend to be more and more data intensive. The data-access delay, not the processor speed, has become the bottleneck of computing, especially for high-performance and high-end computing where performance is keen. There is a great need for research in improving data-access performance.

In this dissertation, we propose to improve data-access efficiency with a Hybrid Adaptive Prefetching architecture and associated innovative data prefetching techniques. The Hybrid Adaptive Prefetching architecture is built upon the memory hierarchy model, the for see engineering choice for masking the gap between computing and data-access speed, and enhances it with a hierarchical prefetching model to further mitigate the performance disparity and improve data-access speed. The fundamental idea behind the proposed solution is utilizing the excessive transistors on chip and available computing capability to build up specialized hardware and software approaches to accelerating data accesses, and thus to achieve a high sustained performance instead of a high peak performance. The Hybrid Adaptive Prefetching architecture reduces data-access latency via two stages, cache-memory stage by leveraging specialized hardware solutions and memory-disk stage by exploiting innovative software solutions. It improves data-access efficiency by harvesting the benefits of comprehensive, aggressive and adaptive
prefetching strategies. The goal of this dissertation is to exploit hardware, compiler and system support to provide a systematic solution to boosting data-access performance for high-performance and high-end computing. Extensive experimental testing has been conducted to validate the design and verify the performance gain, and the results have demonstrated significant performance improvement. The Hybrid Adaptive Prefetching architecture can benefit a variety of applications such as scientific simulation, data mining, geographical information system, multimedia and visualization applications, etc. It will have a broad impact on improving data-access efficiency for high-performance and high-end computing.
CHAPTER 1
INTRODUCTION

Computing speed keeps increasing rapidly, whereas data-access speed has been increasing very slowly. This disparity causes a huge gap between the computation performance and data-access performance, which results in a mismatch between the system peak performance and sustained performance. The long data-access delay is being gradually recognized as a crucial problem for today’s high-performance and high-end computing systems. This chapter reviews the background of this long data-access delay problem and discusses the motivation for this dissertation.

1.1 Growing Gap Between Computing and Data Access

As technology evolves, the uni-processor performance was improved by around 52% a year until 2004, as Moore’s law [61] predicted, and has been increasing by around 25% per year since then [37]. The increasing speed is slowed down recently because the uni-processor performance has reached many limitations, such as pipeline depth and power consumption. There is only very limited instruction-level parallelism available to be exploited in the processor instructions issue pipeline. In addition, if we keep increasing the frequency of uni-processor to scale the processor speed, the power consumption will become intolerable [37].

Instead, the multicore architecture presents a more cost-effective processor architecture and has become the trend for future high-performance processor chip [37]. This is governed by the Pollack’s rule [73], which states that if we integrate a certain number of transistors into a chip, the performance gain is roughly equal to the square root of the number of transistors (or the die size). This observation means that if we increase
the number of transistors to speedup uni-processor architecture, the performance gain is only the square root of the investment. It is expected to be better to separate the same amount of transistors to build several small and independent cores, and explore a task-level parallelism to improve the throughput and the performance gain. Pollack’s rule is an empirical rule, and has been confirmed by all major processor manufactures including Intel, Sun, IBM and AMD.

Many processor vendors have actively participated in manufacturing multicore architecture processors. For instance, Sun Microsystem, Intel and IBM joined in researching and developing multicore processor architecture back into year 2002. IBM has announced Cell processor with one master core and eight slave cores in 2005. The Cell processor has been widely used in video processing, gaming, scientific computing, etc. [35]. AMD has released quad-core processors, such as Phenom and Opteron processor families [102], in 2007. Sun has announced T2 processor with eight cores in 2007 [113]. The T2 processor has been widely adopted in building compute nodes and server nodes for high-performance/high-end computing. Intel has released a six-core processor, called Dunnington, in 2008, as its major successive processor products [103]. Intel has also announced a research prototype TeraFlops chip in 2007 [104]. This prototype chip has 80 cores and can deliver 1.01 TeraFlops computation capability. This performance is about equal to an ASCI Red supercomputer with ten thousands of processors of year 1996. The aggregate processor performance of a single chip nowadays becomes much higher than ever before and keeps increasing rapidly.

In the meantime, data-access performance (latency and bandwidth) increases very slowly. For instance, the memory speed and the disk speed are only increasing by roughly
9% and 7% respectively each year [37]. The performance disparity between processor and memory/disk keeps expanding. Figure 1.1 illustrates the trend of the processor, memory and disk performance improvement over recent three decades. The base case for the comparison is the processor, memory and disk performance of a VAX-11/780 machine in 1980. The performance improvement of processor and data-access speed over 30 years is normalized to the base case and is plotted in this figure. Notice that the vertical axis is a logarithmic scale. As the figure shows, the processor performance has been increasing with a much greater magnitude than data access does. With advances of the chip manufacturing technology and multicore architecture, the performance gap between the aggregate processor performance (or peak performance) and the memory/disk performance is huge and keeps increasing.

![Figure 1.1. Performance Improvement Trend of Processor, Memory and Disk](image)

Figure 1.1. Performance Improvement Trend of Processor, Memory and Disk
The reason behind the performance gap between computing and data access has several folds. Firstly, most advanced architectural and organizational efforts are focused on processor technology, instead of memory or storage device. Secondly, the underlying semiconductor technology improves dramatically, which results in much smaller and many more transistors to be built on chip for processing units, and thus have a high computational capability. Thirdly, the primary technology improvement for memory and storage devices focuses on higher density, which results in much larger memory and storage capacity, but the bandwidth and the latency are improved very slowly.

There are two well-known observations about this performance trend of processor and memory/disk: memory-bounded speedup [90] and memory-wall/disk-wall [98][37]. The memory-bounded speedup observation stated that the performance of a scalable application is limited by the memory performance. The memory-wall/disk-wall study analyzed the computer architectural advances and concluded that the data-access performance has become a dominant factor that decides the sustained performance of a computing system. Both studies revealed that the sustained performance is limited by data-access performance. Our preliminary studies [18][23][84][85][88] on the scalability of multicore architecture and general computing systems also confirmed that the performance of a scalable systems depends largely on data-access performance.

1.2 Memory Hierarchy Model

As the trend of architectural advances shows and the existing observation reveals, long data-access delay has become the major crisis and a critical problem in fully utilizing the computing capability. As a consequence, how to reduce data-access latency effectively becomes a key to achieve a high sustained performance.
The model of memory hierarchy has been the major solution that bridges the performance disparity between computing and storage for the past decades [37][39]. Each level of the hierarchy has the properties of higher bandwidth, smaller size, and lower latency than lower levels. The rationale behind memory hierarchy design is the principle of data locality and the cost-performance of different storage technology. The principle of data locality states that most programs do not access all code or data uniformly. Instead, most programs tend to reuse data and instructions which are accessed recently (temporal locality) or to access those items whose addresses are close to one another (spatial locality). Meantime, the engineering practice shows that the smaller hardware is faster and has a lower latency. The data locality principle, plus the guideline that smaller hardware is faster, led to a hierarchical design based on storage of different speeds and sizes.

Figure 1.2 illustrates a typical multilevel memory hierarchy design. This example includes register files level, primary (level one) data cache and instruction cache level, secondary (level two) cache level, main memory and disk memory level. The primary data cache and instruction cache are separate, while the secondary cache is unified for both data and instruction. The register files, primary and secondary caches are usually on the processor chip. The hardware implementation of them generally belongs to the Static Random Access Memory (SRAM) technique, which is a type of semiconductor memory and does not need to be periodically refreshed, as the word “static” indicates, because SRAM uses bistable latching circuitry to store each bit. The main memory implementation is generally a Dynamic Random Access Memory (DRAM) device, which stores each bit of data in a separate capacitor within an integrated circuit. Unlike SRAM,
DRAM needs to be refreshed periodically because real capacitors leak charge and the information eventually fades if without periodic refresh. The disk storage is a general category of storage mechanisms, in which data is recorded on planar, round and rotating surfaces, but generally refers to magnetic hard disk storage. Recently, other storage mechanisms have emerged, such as Solid State Drive and non-volatile flash memory storage. These storages have significant better random access performance than hard disk has, but the data throughput for a large and well-formed access is not significantly better as expected. In this dissertation, we consider the magnetic hard disk drive as the disk memory storage as it is the dominant storage device currently.

![Memory Hierarchy Model](image)

**Figure 1.2. Memory Hierarchy Model**

The speed of each level decreases substantially, while the size of each level increases dramatically. The register files level is the fastest memory hierarchy. Its access latency ranges from 0.2 ns to 0.3 ns, and its size ranges from 256 bytes to 512 bytes as of current technology. The latency and size of primary cache range from 1 ns to 2 ns, and 64KB to 256KB, while that of secondary cache range from 10 ns to 20 ns, and 256KB to 2MB currently. The main memory has a 100 ns to 200 ns access latency, and 256MB to 4GB capacity. The latency and capacity of disk memory level range from 10 ms to 50 ms, and 100GB to 500GB. Since fast memory is smaller and more expensive, the memory hierarchy is organized into several levels. The goal of the memory hierarchy model is to
provide a memory system with cost almost as low as the cheapest level and speed almost as fast as the fastest level.

1.3 **Data Prefetching and Limitations**

The memory hierarchy model alone, however, has been generally agreed as far from enough to bridge the expanding performance gap between computing and data access [37]. Especially, when applications lack locality due to a working set size larger than the cache and/or non-contiguous data accesses, multi-level memory hierarchy is ineffective. *Data prefetching* has been agreed as one of the most critical techniques to further reduce data-access delay and thus bridges the performance disparity, especially when applications lack temporal or spatial locality. As the term indicates, data prefetching is a technique to fetch data in advance, as shown in Figure 1.3. The essential idea is to observe data referencing patterns, then to speculate future references, and to fetch the predicted reference data closer to the processor before the processor demands them. Numerous studies have been conducted and many strategies have been proposed for data prefetching [2][14][16][27][28][30][34][40][44][49][51][64][76][81][99][101]. These studies concluded that prefetching is a promising solution to masking long data-access latency.

![Figure 1.3. Data Prefetching: Fetching Data in Advance](image)

Although many efforts are devoted to data prefetching research, the existing studies are far from mature yet. The detailed analysis of existing studies on data
prefetching is presented in prior survey studies [10][12] and Chapter 6. The major limitations of existing work have several folds, summarized as follows.

- Limited adaptive support. Most of existing data prefetching studies focus on a specific prefetching strategy or prefetching algorithm only. The adaptive support is very limited and usually only in the form of varying prefetching degree and prefetching distance adaptively. However, application features dominate the access pattern. There is no single universal prefetching algorithm suitable for all applications. An effective and general adaptive support for various algorithms and strategies is desired.

- Limited coverage and accuracy. Most of current prefetching studies are focused on the prediction based on past histories. These approaches, however, suffer limited coverage and accuracy when accesses do not follow regular patterns. Recent studies have started exploring speculative execution approaches that can improve coverage and accuracy greatly, but these approaches are still conservative. The effectiveness of data prefetching highly depends on the prediction accuracy and coverage, and some prefetching approach that provides high accuracy and wide coverage is desired.

- Limited systematic support. Many prefetching approaches are closely related and complementary to each other. They reside at different levels to serve data prefetching. It is beneficial to provide a systematic way and integrate different approaches at various levels together to deliver an effective latency reduction. It is desired that these complementary approaches work cooperatively and improve data access performance collectively.
1.4 Hybrid Adaptive Prefetching Architecture Solution

While computing capability is still increasing with a much faster pace than data-access performance, we argue, in this dissertation, that it would be beneficial to focus on reducing data-access latency to achieve a *high sustained performance* instead of building extensive computing capability to achieve a *high peak performance*. To achieve this goal, a comprehensive data prefetching architecture is the most important and realistic solution. This prefetching architecture can provide wide coverage and high accuracy, and support adaptive algorithms for various desires. It also should integrate various data prefetching strategies together to deliver an effective overall data-access latency reduction.

As the processor-memory/disk performance gap grows, application features demand faster access to data, and hardware/software technologies evolve, we propose a *Hybrid Adaptive Prefetching (HAP)* architecture to fully harvest benefits of data prefetching techniques. The proposed solution employs a hybrid architecture that improves data-access performance via two stages, cache-memory stage and memory-disk stage. It also explores a variety of hybrid prefetching techniques to mask data-access latency effectively and supports adaptive strategy to benefit various applications. It addresses the limitations of existing data prefetching technique we have identified and discussed previously well. We have investigated and solved technical challenging issues of the system design, and completed a prototype system to verify the performance gain of the HAP architecture. The experimental testing has shown promising results of the HAP architecture in bridging the processor-memory/disk performance gap. The Hybrid Adaptive Prefetching architecture, in essence, directs the computer architectural design and hardware/software evolution in two important directions: *trade computing power as*
data-access power, and reduce data-access delay with comprehensive data prefetching techniques.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 presents the preliminary study on the scalability modeling and analysis of emerging multicore and traditional parallel and distributed architecture. This preliminary study has revealed that data-access performance is a critical factor that limits the overall system performance. Chapter 3 presents a high-level overview of the proposed Hybrid Adaptive Prefetching architecture for improving data-access efficiency. Chapter 4 and Chapter 5 explore detailed solutions to reducing data-access latency at two stages, cache-memory stage and memory-disk stage, respectively. Chapter 6 reviews related work on data prefetching and compares them with the solution discussed in this dissertation. Chapter 7 concludes this dissertation and discusses potential future work.
CHAPTER 2
DATA ACCESS: BOTTLENECK OF COMPUTING

This chapter presents a prior study on the scalability modeling and analysis of recently emerged multicore processor architecture and traditional parallel and distributed computing architecture. We apply the scalable computing concept to analyze the scalability and inherent performance limitation of multicore processor architecture. We also analyze the scalability and performance limitation constraints of parallel and distributed system with a novel isospeed-efficiency model. These preliminary studies reveal that data-access performance plays a critical role in the performance of high-performance/high-end computing systems, and innovative techniques to reduce long data-access latency are essential in keeping the computing system scalable. These studies have been published in referred conference proceedings and journals in [18][23][84][85][88].

2.1 Modeling Scalability of Emerging Multicore Architecture

The emerging multicore architecture provides a new dimension to scale up the number of processing elements, i.e. cores, and therefore, the potential computing capacity on a single chip. It has been generally agreed that multicore architecture is the trend of future high-performance processors. While it is accepted that we have entered the multicore era in general, concerns exist on the inherent performance limitation and the scalability of multicore processors. Many believe that multicore architecture is not scalable citing Amdahl’s law [1]. In this section, we analyze the multicore processor architecture and evaluate the Amdahl’s law for multicore architecture. Upon that, we apply scalable computing principles to analyze multicore scalability under scaled computing conditions and from the data access (memory wall) perspective. These models
show that multicore architectures are scalable and not limited by Amdahl’s law. In addition to evaluating the future of multicore scalability, we identify what we believe will ultimately limit the performance of multicore systems: the memory wall [98], or the data-access bottleneck problem. In the following chapters, we present our proposed solution to mitigating the data-access bottleneck problem. We first revisit the speedup models of parallel processing, fixed-size (Amdahl’s law), fixed-time and memory-bounded speedup; we then extend them to multicore scalability analysis.

2.1.1 Speedup Models of Parallel Processing. Amdahl’s law [1] states that if a portion of a computation, $f$, can be improved by a factor $m$, and the other portion cannot be improved, then the portion that cannot be improved will quickly dominate the performance, and further improvement of the improvable portion will have little effect. Speedup is defined as sequential execution time over parallel execution time in parallel processing. Let $f$ be the portion of the workload that can be parallelized and $m$ be the number of processors; then the parallel processing speedup implied by Amdahl’s law is:

$$\text{Speedup}_{\text{Amdahl}} = \frac{1}{(1 - f) + \frac{f}{m}}. \quad (2.1)$$

When $m$ increases to infinity, the speedup upper bound is:

$$\lim_{m \to \infty} \text{Speedup}_{\text{Amdahl}} = \frac{1}{1 - f}.$$

Since most applications have a sequential portion that cannot be parallelized, by Amdahl’s law, parallel processing is not scalable. For instance, if 90 percent of an application can be parallelized and 10 percent cannot, then with 8 to 16 processors, the 10 percent sequential work will contribute about 50-80 percent of the total execution time, and adding more processors for parallel processing will have a diminishing effect.
A tacit assumption in Amdahl’s law is that the problem size, or the workload, is fixed to that which runs on the unenhanced system. The speedup emphasizes time reduction of a given problem. Amdahl’s law is thus also called the \textit{fixed-size speedup} model \cite{26}\cite{39}\cite{46}\cite{89}. In 1988, Gustafson introduced the concept of scalable computing and the \textit{fixed-time speedup} model \cite{36}. The fixed-time speedup model argues that powerful machines are designed for large problems and problem size should scale up with the increasing of computing capability. For many practical workloads (e.g. real time applications), the problem size scale-up is bounded by the execution time. Thus, the fixed-time speedup is defined as:

\begin{equation}
\text{Speedup}_{\text{FT}} = \frac{\text{Sequential Time of Solving Scaled Workload}}{\text{Parallel Time of Solving Scaled Workload}}
\end{equation}

(2.2)

Supposing the original workload, \( w \), and the scaled workload \( w' \), finish in the same amount of time with sequential processing and parallel processing with \( m \) processors, respectively; and assuming the scale of the workload is in the parallel processing part only; we have \( w' = (1 - f)w + fmw \). Therefore,

\begin{equation}
\text{Speedup}_{\text{FT}} = \frac{\text{Sequential Time of Solving } w'}{\text{Parallel Time of Solving } w'} = \frac{\text{Sequential Time of Solving } w'}{\text{Sequential Time of Solving } w} = (1 - f) + mf
\end{equation}

(2.3)

This equation is known as \textit{Gustafson’s law} \cite{36}. It states that the fixed-time speedup is a linear function of \( m \) if the workload is scaled up to maintain a fixed execution time. Gustafson’s law suggests that it is beneficial to build a large-scale parallel system as the speedup can grow linearly with the system size.

Many applications cannot scale up to meet the time bound constraint due to some physical constraints. In practice, the physical constraint is often the memory limitation. With this consideration in mind, Sun and Ni proposed the \textit{memory-bounded speedup}
model [90]. Let $w^*$ be the scaled workload under a memory space constraint. The memory-bounded speedup is defined as:

$$\text{Speedup}_{MB} = \frac{\text{Sequential Time of Solving } w^*}{\text{Parallel Time of Solving } w^*} \quad (2.4)$$

Assume that each computing node is a processor-memory pair. Increasing the number of processors, then, will increase the memory capacity as well. Let $y = g(x)$ be the function that reflects the parallel workload increase factor as the memory capacity increases $m$ times. That is $w = g(M)$, and $w^* = g(m \cdot M)$, where $M$ is the memory capacity of one node. We have $w^* = g(m \cdot g^{-1}(w))$. Thus memory-bounded speedup is:

$$\text{Speedup}_{MB} = \frac{(1-f)w + f \cdot g(m \cdot g^{-1}(w))}{(1-f)w + \frac{f \cdot g(m \cdot g^{-1}(w))}{m}} \quad (2.5)$$

Equation (2.5) looks complicated, but for any power function $g(x) = ax^b$ and for any rational numbers $a$ and $b$, we have: $g(mx) = a(mx)^b = m^b \cdot ax^b = m^b g(x) = g(m)g(x)$, where $g(m)$ is the power function with the coefficient as 1. Since many algorithms have a polynomial complexity in terms of computation and memory requirement, and we can always take the highest degree term to represent the complexity of the algorithm, we can simplify Equation (2.5) into:

$$\text{Speedup} = \frac{(1-f)w + f \cdot g(m)w}{(1-f)w + \frac{f \cdot g(m)w}{m}} = \frac{(1-f) + f \cdot g(m)}{(1-f) + \frac{f \cdot g(m)}{m}} \quad (2.6)$$

We provide a quick example to illustrate the calculation of $g(m)$ for matrix multiplication. The computation requirement of matrix multiplication is $y = 2N^3$ and the memory requirement is $x = 3N^2$, where $N$ is the dimension of the two $N \times N$ source
matrices. Thus: \( g(x) = 2 \left( \frac{1}{\sqrt{3}} \right)^3 = \frac{2}{3^3} x^2 \), and \( \bar{g}(x) = x^{3/2} \). Therefore, the memory-bounded speedup for matrix multiplication is:

\[
\text{Speedup} = \frac{(1 - f) + f \cdot \bar{g}(m)}{(1 - f) + f \cdot \frac{g(m)}{m} (1 - f) + f \cdot m^{1/2}}
\]  

(2.7)

In general, if we assume each element stored in memory will be used at least once, we have \( w^* \geq w' \), and the memory-bounded speedup is greater than or equal to the fixed-time speedup. Equation (2.6) is also known as Sun and Ni’s law [26][39][89][90]. It is a generalization of Amdahl’s law and Gustafson’s law, where Amdahl’s law is a special case with \( g(m) = 1 \), and Gustafson’s law is a special case with \( g(m) = m \). In general, the computational workload increases faster than the memory requirement, thus \( \bar{g}(m) > m \) and the memory-bounded speedup model gives a higher speedup than the fixed-size and fixed-time speedup. Memory-bounded speedup is natural for domain decomposition based applications and can be applied at different levels of a memory hierarchy system. It becomes more and more important with increasing awareness of the memory-wall problem [98].

2.1.2 Multicore Architecture Model and Assumptions. We follow the models of parallel processing to study the scalability and performance limitation of multicore architectures in terms of cores on a single chip. This subsection presents the multicore architecture model and assumptions we have made.

2.1.2.1 Assumptions for Multicore Architecture. To simplify the discussion, this chapter assumes that the multicore architecture under study is a symmetric multicore processor architecture. A multicore processor is symmetric if each of the cores in a chip
multiprocessor is identical. Figure 2.1 illustrates the memory hierarchy of a multicore processor we assume in this chapter. The assumed processor has $n$ cores. We assume that each core in a chip multiprocessor has a dedicated primary cache, L1 cache, and all cores share remaining levels of the memory hierarchy. This assumption matches with most of existing multicore processors that are either commercially available or in production.

![Figure 2.1. Symmetric Multicore Processor](image)

### 2.1.2.2 Hardware Cost Model for Multicore Architecture.

Hill and Marty [38] recently applied Amdahl’s concepts to multicore architectures and, citing hardware designs limitations, pessimistically concluded that the future of scalable multicore processors is questionable. Some others follow up with more limitations of multicore scalability based on Amdahl’s law [97]. Hill and Marty [38] gave a simple hardware model for multicore chips. We follow this hardware model to analyze the scalability of multicore architecture from scalable computing perspective.

This hardware cost model [38] assumes that a multicore chip under study can contain at most $n$ base core equivalents (BCEs) and each single BCE implements the
baseline core. This assumption comes from the fact that the microarchitects can only
dedicate limited resources on a chip. This cost model also assumes that microarchitects
have the technique to create a more powerful core with \( perf(r) \) sequential performance
with \( r \) BCEs, where the performance of a single BCE is assumed to be 1. The value of
\( perf(r) \) depends on the actual hardware technique and implementation, but in analysis, it
can be an arbitrary function.

2.1.3 Scalability of Multicore Architecture. We are now ready to analyze the
scalability of multicore architecture with the concept of scalable computing. We first
evaluate the Amdahl’s law (fixed-size model) and Hill and Marty’s study for multicore
architecture, then introduce the scalable computing perspective to analyze the scalability
of multicore architecture and study fixed-time and memory-bounded models.

2.1.3.1 Fixed-size Model for Multicore Architecture. Following Amdahl’s law, Hill
and Marty’s study [38] conclude that the speedup of a symmetric multicore architecture
is:

\[
\text{Speedup} = \frac{1}{1 - f \cdot \frac{r}{\text{perf}(r)} + f \cdot \frac{r}{\text{perf}(r) \cdot n}}
\]  

(2.8)

While it is not given in [38], here we provide the deduction of Equation (2.8) so it
can be better related to the scaled scalability analysis given in following section.

According to speedup definition:

\[
\text{Speedup} = \frac{\text{Enhanced performance}}{\text{Original performance}} = \frac{T_{\text{Original}}}{T_{\text{Enhanced}}}
\]

(2.9)

where the performance is the reciprocal of the execution time. Let us assume that the
problem size is \( w \). Thus the original execution time is \( T_{\text{Original}} = w / \text{perf}(1) = w \), where a
single BCE core has a performance of 1 as assumed by Hill and Marty [38]. The new
equation time of \( n \)-BCE multicore is
\[
T_{\text{Enhanced}} = \frac{(1-f)w}{\text{perf}(r)} + \frac{fw}{n \cdot \text{perf}(r)}.
\]
If we assume these
\( n \)-BCE resources are built into \( n/r \) cores, where each core has a \( \text{perf}(r) \) performance.
Therefore, the speedup is:
\[
\text{Speedup} = \frac{\text{Enhanced performance}}{\text{Original performance}} = \frac{w/\text{perf}(1)}{(1-f)w + \frac{fr}{n \cdot \text{perf}(r)}} = \frac{1}{1 - f + \frac{f \cdot r}{\text{perf}(r) \cdot n}}
\]
\( \text{perf}(r) \) is a constant for a given design. Let \( \text{perf}(r) = c \) and \( m = n/r \); then Equation (2.8)
becomes: \[\text{Speedup} = \frac{c}{(1-f) + \frac{f}{m}},\]
which is the Amdahl’s law that invariably results from a
fixed-size workload assumption.

![Figure 2.2. Fixed-size Speedup of a Multicore Architecture](image)

By Equation (2.8), the scalability of multicore is rather limited. Figure 2.2
illustrates the fixed-size speedup of multicore architectures, where \( c \) equals 1. The
horizontal axis represents the number of cores, scaled from 1 to 256. The vertical axis
represents the speedup value. This figure plots the fixed-size speedup results with \( f \) ranging from 0.5 to 0.999. As shown clearly from this figure, the fixed-size speedup model (Amdahl’s law) illustrates a very limited scalability of a multicore architecture, and the speedup is quickly restricted by the sequential portion of a problem under study. The scalability is acceptable only when the problem is highly parallelizable, such as the improvable portion is over 99.9%.

2.1.3.2 Fixed-time Model for Multicore Architecture. We take \( n \), the number of base cores, as the scaling factor. The scalability question is whether we should have a large \( n \).

Following Equation (2.8), let \( n = r \); we have

\[
\text{Speedup} = \frac{1}{1 - f} + \frac{f}{\text{perf}(r)} = \text{perf}(r)
\]

Let \( n = r \) be the initial point, and \( n = mr \) as the scaled number of cores. Following the fixed-time model assumption that the scaling is only at the parallel portion, for the fixed-time speedup model we have:

\[
(1 - f)w + \frac{fw}{\text{perf}(r)} = \frac{(1 - f)w}{\text{perf}(r)} + \frac{fw'}{\text{perf}(r)m}
\]

Thus, \( w' = mw' \). Hence, the scaled speedup, compared with \( n = r \) is:

\[
\text{Speedup} = \frac{\text{Sequential Time of Solving } w'}{\text{Sequential Time of Solving } w} = \frac{(1 - f)w}{\text{perf}(r)} + \frac{fw'}{\text{perf}(r)m} = (1 - f) + mf
\]

Equation (2.11) shows that multicore architectures are scalable under the scalable computing model, and their fixed-time speedup grows linearly with the scaling factor.

Figure 2.3 reveals the scalability of multicore architectures with the fixed-time speedup model. We compute the speedup following formula (2.11) under different scenarios where \( f \) ranges from 0.2 to 0.99, and plot the results in Figure 2.3. The fixed-
time speedup model, as shown in Figure 2.3, presents a more optimistic view of the multicore architecture. For instance, when $f$ equals 0.9, the speedup achieved is 922 with 1024 cores, where by Amdahl’s law, Equation (2.8), the speedup is around 10. When $f = 0.99$, the fixed-time speedup is 1013 with 1024 cores.

![Figure 2.3. Fixed-time Speedup of a Multicore Architecture](image)

The continued performance improvement of fixed-time speedup is due to the fact that it continuously has enough work for parallel processing. After scaling, the parallel work is $fw'$, and the total work is: $(1 - f)w + fw' = [1 + (m - 1)f]w$. Thus, the new parallel work over total work ratio is $f^* = \frac{mfw}{[1 + (m - 1)f]w} = \frac{f}{1 + \frac{m - 1}{m}f}$. When $m \to \infty$, the parallel work ratio approximates to 1. Under the fixed-time model, multicore architectures are scalable and not limited by the sequential processing term.

2.1.3.3 Memory-bounded Model for Multicore Architecture. We study the memory-bound model for multicore architecture in this subsection. The memory bound under the following analysis is the cumulated capacity of the L1 caches. Please notice that the
memory-bounded condition can be applied at different levels of the underlying memory hierarchy. For instance, should the capacity of L2 increases proportionally with the number of cores, the following analyses for L1 can be directly applied to L2.

Following a similar analysis of fixed-time model, and assuming the scaled workload under memory capacity constraint is \( w^* \), we have the speedup under memory-bounded model, when the number of cores is scaled from \( r \) to \( mr \), as:

\[
\text{Speedup} = \frac{\text{Sequential Time of Solving } w^*}{\text{Parallel Time of Solving } w^*} = \frac{(1 - f)w + fw^*}{(1 - f)w} \cdot \frac{\text{perf}(r)}{\text{perf}(r)} = \frac{(1 - f)w + fw^*}{m \cdot \text{perf}(r)}
\]

Assume \( y = g(x) \) is the function of computing requirement in terms of memory requirement, \( w = g(M) \), and assume \( g(x) \) is a power function. Therefore, following the previous section, the memory-bounded speedup is:

\[
\text{Speedup} = \frac{(1 - f)w + f \cdot \frac{g(m)}{m}}{(1 - f)w + f \cdot \frac{g(m)}{m}} = \frac{(1 - f) + f \cdot \frac{g(m)}{m}}{(1 - f) + f \cdot \frac{g(m)}{m}}
\]

Figure 2.4 demonstrates the speedup with the memory-bounded scaled speedup model for multicore architectures. This figure reports the speedup value of the matrix multiplication example. Similar to the fixed-time model, the memory-bounded speedup model reveals that a multicore architecture can scale up well as long as the workload size of the application can be allowed to grow with the number of cores. In addition, the results of the memory-bounded speedup model show that an even better performance can be achieved when the memory capacity constraint is used to scale the workload instead of the execution time constraint. As revealed in Figure 2.4, the scalability of a multicore architecture can increase steadily, in contrast with the fixed-time model. The memory-
bounded speedup model reflects situations where memory capacity is the constraint, in the case of the L1 cache of multicore architectures as we discussed; the fixed-time speedup model reflects situations where the execution time is limited by human patience or the workflow situation. Both models exhibit a promising view of large-scale multicore architectures.

Figure 2.4. Memory-bounded Speedup of a Multicore Architecture

2.1.3.4 Comparison of Fixed-size, Fixed-time and Memory-bounded Models. Figure 2.5 combines the fixed-size, fixed-time and memory-bounded speedup together for comparison. We pick three scenarios, $f$ with value 0.5, 0.9 and 0.99, for each speedup model. The speedups of fixed-size, fixed-time and memory-bounded models are represented with different line patterns. As illustrated in this figure, with the scalable computing viewpoint and the scaled speedup models, a multicore architecture can scale up well and linearly. The scalable computing notion and models demonstrate a much more optimistic view than Amdahl’s law does, and suggest large-scale multicore
architectures are of broad value. The direct comparison also verifies that memory-bounded speedup is likely to be higher than fixed-time speedup.

Figure 2.5. Fixed-size, Fixed-time and Memory-bounded Speedup of a Multicore Architecture

These results and analyses confirm that the scalable computing concept and two scaled speedup models, fixed-time speedup model and memory-bounded speedup model, are applicable to multicore architecture design. Hill and Marty’s conclusion on the scalability of multicore architecture [38] is essentially a corollary of Amdahl’s law. Their analysis and formulation are correct, but, as Amdahl’s law, only apply if users do not increase their computing demands when given more computing power.

These analyses have also revealed that sequential processing is not a limiting factor of multicore scalability, at least not in the sense of scalable computing. Nonetheless, we are having difficulties utilizing today’s multicore systems. A question we have to ask ourselves is: if sequential processing is not the limiting factor for scalability, then what is. We believe the limiting factor is data-access delay, or the so-
called *memory-wall problem*. We revisit the scalability problem considering data access as the factor limiting performance in the following subsection.

### 2.1.4 Memory Wall and Multicore Architecture Scalability

Multicore processor scalability is not necessarily the same as multicore-processor parallel processing scalability. For many applications, such as meta-tasks, high-throughput computing, or perfectly parallel applications, the sequential portion of the parallel workload is not the limiting factor for performance. Nonetheless, the performance of these types of applications is often limited on multicore architectures.

The memory-bounded scaled speedup model gives a performance upper bound where all the data are stored in the L1 caches. But, for any actual application with reasonable size, data may have to be accessed through the memory hierarchy, where long data-access delay occurs, a.k.a. the *memory-wall problem* [98], in addition to the contention of the shared L2 cache and data paths to the lower level of the memory hierarchy. The memory-wall problem is due to the disparity of technology advance between CPU speed and memory data access latency. In the following, we study the scalability of multicore architecture with data-access delay as the scalability overhead.

For data-access scalability analysis, we change the cost model slightly. We assume a task as two parts: data processing work, $w_p$, and data communication (access) work, $w_c$, and $w = w_p + w_c$. We assume $w_c$ is a function of $r$, but it is independent of the workload and the number of cores. As in previous section, the design choice is to choose an appropriate $r$ to optimize $\text{perf}(r)$ under the same assumption that the performance of a single BCE core is 1, and the scalability concern is on determining an appropriate number
of base cores, \( n \), for best performance. Following a similar deduction as given in Section 1, we have the fixed-size speedup as:

\[
\text{Speedup}_{FS} = \frac{1}{\frac{w_c}{\text{perf}(r)} + \frac{w_p \cdot r}{\text{perf}(r) \cdot n}}.
\]

For fixed-time speedup, taking \( n = r \) as the initial point, following the fixed-time principle where \( n = mr \), we have:

\[
\text{Speedup}_{FT} = \frac{\frac{w_c}{\text{perf}(r)} + \frac{w_p'}{m \cdot \text{perf}(r)}}{\frac{w_c}{\text{perf}(r)} + \frac{w_p}{\text{perf}(r)}} = \frac{w_c + m \cdot w_p'}{w_c + w_p}.
\]

Thus, \( w_p' = mw_p \). Therefore, the fixed-time speedup compared with \( r \) BCEs is:

\[
\text{Speedup}_{FT} = \frac{w_c}{w_c + w_p} + \frac{w_p'}{w_c + w_p} = (1 - f') + mf'.
\]

If we let \( f' = \frac{w_p}{w_c + w_p} \) then we have the familiar format \( \text{Speedup}_{FT} = (1 - f') + mf' \).

For memory-bounded speedup, when the number of cores is scaled from \( r \) to \( mr \), we have:

\[
\text{Speedup}_{MB} = \frac{\frac{w_c}{\text{perf}(r)} + \frac{fw_p^*}{m \cdot \text{perf}(r)}}{\frac{w_c}{\text{perf}(r)} + \frac{fw_p^*}{m \cdot \text{perf}(r)}} = \frac{w_c + f \cdot g(m \cdot g^{-1}(w_p))}{w_c + \frac{fw_p^*}{m}} = \frac{w_c + \frac{f \cdot g(m \cdot g^{-1}(w_p))}{m}}{w_c + \frac{fw_p^*}{m}}.
\]

For any power function \( g(x) \), we have a simplified formula as:

\[
\text{Speedup}_{MB} = \frac{w_c + f \cdot g(m)w_p}{w_c + \frac{f \cdot g(m)w_p}{m}}.
\]
Similar to the analysis in the previous section, it is likely that the memory-bounded speedup is greater than the fixed-time speedup since the computing requirement is generally greater than memory requirement.

These analyses and results reveal that, if we assume the data-access delay is a constant that does not increase with problem size and the number of cores, the scalable computing concept and models are still applicable to multicore architecture. While the assumption of fixed data-access time is not true under today’s technology, it is a technical issue not an inherent, immovable obstacle. There is no innate limitation to multicore architecture scalability, but the need for technical improvements is primarily in data-access performance.

Since the improvement of data-access performance cannot be limited at the microprocessor or cache level only, the scalability issues of multicore architecture involve the whole architecture design of a computing system. The memory-wall problem is a complicated technical issue, yet for the scalability of multicore we only need the data delay to be constant. With research and technology advance, we should be able to mitigate the memory-wall effect and provide a much better performance than that offered by today’s multicore architecture. The scalability analysis of multicore architecture demonstrates the great needs for research to overcome the technical hurdles, especially in reducing the data access delay, for computing systems.

2.2 Modeling Scalability of Parallel and Distributed Architecture

As computing systems evolving, understanding scalability and inherent performance limitation, such as data-access or communication performance, of parallel and distributed environments becomes timely important and necessary. In this section, we
present an *algorithm-system* approach and an *isospeed-efficiency model* for studying the scalability of general computing system, based on the isospeed metric proposed in [91]. Analytical and experimental studies are conducted to confirm the feasibility of the isospeed-efficiency scalability model, and the results have shown that the new model is practical and effective.

2.2.1 Isospeed-efficiency Scalability Model. In a high-performance/high-end computing environment, a code runs on a tightly coupled or distributed system. We often refer the code as the algorithm behind it to emphasize the importance of the scalability analysis of the algorithm. Thus, we choose the term *algorithm-system* combination, instead of *code-machine* combination [91], for the scalability study. To completely describe the attributes of a given algorithm-system combination, we need to characterize all computing features of the system including the CPU frequency, memory capacity and speed, network bandwidth, I/O latency and etc. In engineering practice, however, we cannot get into all the details; otherwise, the scalability model will be too complex to use. It is desired to balance the simplicity and the effectiveness. The model should be capable of catching the key features of an algorithm-system combination and hiding the details at the same time. For this reason, we introduce a new concept, *marked-speed*, to describe the aggregate computing power of a general parallel and distributed computing system.

2.2.1.1 Definition of Marked-speed. We first introduce the definition of marked-speed for a computing system and a computing node.

**Definition 1** The *Marked-speed* of a general computing system is defined as the combined *marked-speed* of all nodes in the system, where the *marked-speed* of each node
is defined as the (benchmarked) sustained speed, and speed is defined as work divided by execution time.

As defined, a general system’s marked-speed is the numeric summation of the quantitative marked-speed of all nodes that compose the system. It captures the essential of the computing power and represents the cumulative computational capability of a general parallel/distributed system, but does not represent other non-computation features like data access and network communication capability. The marked-speed can be calculated based on the hardware peak performance, which in general is much higher than an actual delivered performance. In practice, we can use standard benchmarks, such as Linpack [105], NPB [109] or an appropriate benchmark from the Perfect benchmarks suite [110], to measure each node’s sustained speed and calculate the whole system’s marked-speed. To guarantee the comparability, we should use the same benchmark for measurement. We will demonstrate the usage of marked-speed in the analytical study section. The marked-speed is a quantitative measurement of computational power [6][68][100].

Let $C$ denote the marked-speed of the computing system and $C_i$ denote the marked-speed of node $i$. In a heterogeneous environment, $C_i$ might be different from each other due to the heterogeneity of the nodes. In a homogeneous environment, all $C_i$ are the same. According to Definition 1, we have $C = \sum_{i=1}^{p} C_i$ in a general parallel/distributed computing environment with $p$ nodes. In a homogeneous environment, we have $C = \sum_{i=1}^{p} C_i = pC_1$ because all $C_i$ are the same.
2.2.1.2 Definition of Isospeed-efficiency Scalability. While the marked-speed for a given benchmark is a constant, the actual achieved speed of an application may vary with the system and problem size and may not be the same as the marked-speed. This is especially true for parallel and distributed processing where communication and data-access overhead is a major factor of actual achieved speed. We introduce another concept, speed-efficiency, to characterize the performance gain of an algorithm-system combination.

**Definition 2** The *Speed-efficiency* of an algorithm-system combination is defined as the achieved speed of the algorithm on the system divided by the marked-speed of the system.

Let $S$ denote the achieved speed, $W$ denote the work and $T$ denote the execution time, we have $S = \frac{W}{T}$. Let $E_s$ stand for the speed-efficiency. Thus, we have

$$E_s = \frac{S}{C} = \frac{W}{TC}.$$ 

In homogeneous environments, the speed-efficiency becomes the same as the definition presented in [91] because each node has the same marked-speed and the marked-speed of the system can be expressed by using the system size $p$.

Based on previous definitions and discussion, we propose the following *isospeed-efficiency scalability* for any algorithm-system combination on a general parallel/distributed computing system.

**Definition 3 Isospeed-efficiency Scalability.** An algorithm-system combination is *scalable* if the achieved speed-efficiency of the combination can remain constant with increasing the system ensemble size, provided the problem size can be increased with the system size.
The proposed isospeed-efficiency scalability model does not restrict the underlying system and is applicable to both homogeneous and heterogeneous systems. The method for increasing the system ensemble size includes increasing nodes or the number of processors within nodes, or upgrading to more powerful nodes. The approach to increasing the problem size depends on the algorithm.

2.2.2 Isospeed-efficiency Scalability Function. For a scalable algorithm or application, its communication and data-access requirement should not increase faster than its computation requirement. Therefore, we can increase the problem size to keep the speed-efficiency constant when the system size is increased. The increment of the problem size depends on the underlying computing system and the algorithm itself. This variation provides a quantitative measurement of the scalability. The marked-speed introduced previously is an appropriate representation of the computational capability, thus we use it to represent a general system and call a system with marked-speed $C$ as a system with system size $C$ in the rest of this dissertation.

Let $C$ be the initial system size of a specified computing system, $W$ and $T$ be the initial problem size and the execution time. Let $C'$ be the scaled system size, $W'$ be the increased problem size and $T'$ be the new execution time for the scaled problem size. We define the isospeed-efficiency scalability function as:

$$
\psi(C, C') = \frac{C'W}{CW'}
$$

where $W'$ is constrained by the isospeed-efficiency condition:

$$
\frac{W}{TC} = \frac{W'}{T'C'}
$$

In the ideal situation, there is no communication necessary, which means
If we apply the isospeed-efficiency scalability to a homogeneous environment, we have $C = pC_i$, and $C' = p'C_i$ because all $C_i$ are the same. The scalability function becomes:

$$\psi(C, C') = \frac{CW}{CW'} = \frac{p'W}{pW'}$$

This shows that the original homogeneous isospeed scalability model is a special case of the isospeed-efficiency scalability model.

**2.2.3 Theoretical Studies.** We have analyzed the isospeed-efficiency scalability model in theory for further understandings of scalability studies, and this subsection presents the analysis results.

**Theorem 1**: Suppose an algorithm has a balanced workload on each node and the sequential portion (which cannot be parallelized) of the algorithm is $\alpha$. If we can find a problem size to keep the speed-efficiency constant when the system size is increased, then the system is scalable and the scalability is:

$$\psi(C, C') = \frac{t_0 + T_o}{t'_0 + T'_o}$$

where $t_0$ and $t'_0$ are the execution time of the sequential portion, $T_o$ and $T'_o$ are the communication and data-access overhead of system $C$ and $C'$ separately.

**Proof**: The proof can be found in study [18].

Theorem 1 provides a method to calculate the scalability of an algorithm-system combination, and also shows an insightful understanding of the scalability. It reflects that the scalability is decided by both the sequential portion of the work and the communication and data-access overhead. When the problem size is scaled to keep the
speed-efficiency constant, the sequential portion of the work is increased, as well as the communication and data-access overhead due to scaled system size. Therefore, the scalability is likely to be smaller than 1 in practice.

**Corollary 1:** If an algorithm can be parallelized perfectly and has a balanced workload on each node, and if the communication and data-access overhead is constant for any problem size and system size, then the algorithm-system combination is scalable and the scalability is perfect with a constant value 1.

**Proof:** The proof can be found in study [18].

Corollary 1 analyzes the scalability of an ideal case. According to the previous discussion, the scalability of an ideal case is 1. Corollary 1 also reveals all the conditions that a perfectly scalable algorithm-system combination requires.

**Corollary 2:** If an algorithm can be parallelized perfectly and has a balanced workload on each node, and if we can find a problem size to keep the speed-efficiency constant when the system size is increased, then the algorithm-system combination is scalable and the scalability is

\[ \psi(C, C') = \frac{T_o}{T_o'} \]

**Proof:** The proof can be found in study [18].

Corollary 2 shows another meaningful understanding of the scalability and is useful in analyzing the scalability of an algorithm-system combination. It demonstrates that if an algorithm can be parallelized perfectly and has a balanced workload on each
node, then the scalability will only be decided by the communication and data-access overhead at different system sizes.

In practice, we usually compute the sequential portion of the algorithm on the same node before and after the system is scaled. The following theorem analyzes the scalability in this situation.

Theorem 2: Let an algorithm have a balanced workload on each node and the sequential portion of the algorithm be $\alpha$. Suppose the sequential portion of the algorithm is computed on the same node before and after the system is scaled. If we can find a problem size to keep the speed-efficiency constant for the initial system $C$ and the scaled system $C'$, then the system is scalable and the scalability is

$$\psi(C, C') = \frac{C \beta W - C' \beta W + C T_o}{C T'_o}$$

where $\beta = \alpha / C$, $C_i$ is the marked-speed of the node where the sequential portion of the algorithm is computed, $W$ is the initial problem size, $T_o$ and $T'_o$ are the communication and data-access overhead for system $C$ and $C'$, respectively.

Proof: The proof can be found in study [18].

2.2.4 Calculation of Isospeed-efficiency Scalability. The isospeed-efficiency scalability can be obtained in many ways. The most straightforward way is to compute the scalability. This method measures the execution time at different system and problem sizes and computes the scalability according to the isospeed-efficiency scalability definition. Another approach is to analyze and predict the scalability. This method examines the computational and communicational ratio of the algorithm, as well as the
communication and data-access latency of the machine, and then utilizes derived theoretical analysis results to predict the scalability based on measurements of base cases. This method can also be used to verify the computed scalability. Studies [18][88] have detailed discussion on scalability prediction. The third approach is to measure the scalability directly when scaling the problem size to maintain the isospeed-efficiency [91]. The experiments given in following subsection illustrate the computation of the isospeed-efficiency scalability.

2.2.5 Verification of Isospeed-efficiency Scalability Model. We have carried out experimental testing to verify the proposed isospeed-efficiency scalability model and the theoretical analysis results, and to demonstrate the isospeed-efficiency scalability is practically applicable. Two classical algorithms, Gaussian Elimination and Matrix Multiplication algorithms, and one real application, 2-D Convolution, were selected for testing. The experiments were conducted on a 64-node heterogeneous Sunwulf compute farm in the Scalable Computing Software (SCS) laboratory at Illinois Institute of Technology. The detailed algorithm, implementation, workload analysis, experimental testing results and analyses can be found in study [18].

2.3 Data Access: A Bottleneck of Scalable Computing

We have studied the scalability of multicore architecture and parallel and distributed architecture in this chapter. Based on the scalable computing concept where problem size can be increased with the computing capability, we have derived two sets of performance models for considering sequential processing and data access as the hampering factors of scalability. We have also introduced a novel isospeed-efficiency scalability model. This model can characterize the inherent nature and performance
limitation of general computing systems. The isospeed-efficiency scalability model reveals that if data-access overhead, including communication cost, can be kept fixed, and if an algorithm can be perfectly parallelized, the algorithm-system combination will be perfectly scalable. The model and the theoretical study disclose that data-access performance plays a critical role in the overall system performance. The scalability is primarily attributed to the data-access capability. The experimental testing has also confirmed this fact, as well as the practical usage of the isospeed-efficiency model in analyzing the innate performance limitation of computing systems.

These modeling and analytical studies have discovered that the need for technical improvements to keep the computing system scalable is primarily in data-access performance. The data-access bottleneck issue has to be addressed well to reach the potential of scalable computing. In the following chapters, we present our solution, a Hybrid Data Prefetching architecture, to tackle the data-access bottleneck issue and improve data-access efficiency.
CHAPTER 3
HYBRID ADAPTIVE PREFETCHING ARCHITECTURE

We propose a *Hybrid Adaptive Prefetching* architecture to address the critical data-access bottleneck problem and to improve data-access efficiency by fully exploring the benefits of data prefetching. The essential idea of the Hybrid Adaptive Prefetching architecture is to focus on reducing data-access latency to achieve a high sustained performance instead of a high peak performance. This chapter presents the high-level overview of the Hybrid Adaptive Prefetching architecture and its subsystems. The following chapters discuss the design, challenges, solutions, and experimental testing results in great detail.

3.1 **Hybrid Adaptive Prefetching Architecture**

While the performance gap between computing and storage keeps growing, as discussed in Chapter 1, the memory hierarchy model alone is not enough to bridge the gap. In addition, the goal of memory hierarchy model, to reach the speed of the fastest level while having the cost and capacity of the lowest level, can be rarely achieved without further assistance. We propose a solution to facilitating memory hierarchy model with a Hybrid Adaptive Prefetching architecture to accomplish the task of improving data-access efficiency and bridging the performance gap together. The Hybrid Adaptive Prefetching architecture, in essence, enhances the memory hierarchy model with a *hierarchical model of data prefetching*. 
The Hybrid Adaptive Prefetching architecture, or HAP architecture in short, is an idea of improving data-access performance via two-stage data prefetching, cache-memory stage and memory-disk stage prefetching. We separate these two stages because they require distinct techniques in order to achieve the best result on improving the data-access efficiency. We employ a specialized hardware solution for the cache-memory stage data prefetching because this stage needs a much faster solution. In contrast, we employ a software solution for the memory-disk stage data prefetching because this stage involves significant larger latency and can tolerate the software overhead well. In addition, the Hybrid Adaptive Prefetching architecture exploits comprehensive and adaptive prefetching mechanism to boost data-access performance at both cache-memory and memory-disk stages. We employ both heuristic prediction based and novel pre-execution analysis based prefetching for both irregular and regular accesses. We also adopt an adaptive prefetching mechanism based on the feedback information collected dynamically to be able to adapt to suitable algorithms depending on the specific application feature at runtime.

Figure 3.1 illustrates a high-level view of the Hybrid Adaptive Prefetching architecture. The heuristic prediction approach supports various history-based prefetching algorithms and is applied when accesses have regular or perceivable patterns. These prefetching algorithms are general, and we are able to apply them to the latency reduction at both stages. The heuristic prediction approach also provides support to adapt to a specific prefetching algorithm as necessary depending on actual access patterns, which addresses the limitation of existing data prefetching approaches that lacks effective and general adaptation. When application accesses do not exhibit perceivable patterns, the
heuristic prediction approach only has limited accuracy and coverage. We employ the pre-execution approach to improve the accuracy and coverage. The pre-execution approach works well even when accesses are totally irregular or random. In essence, the heuristic prediction and pre-execution analysis approaches cooperate and complement each other, and together, they apply to both cache-memory and memory-disk stages to improve data-access efficiency.

Figure 3.1. Overview of Hybrid Adaptive Prefetching Architecture

A general logical flow is shown in Figure 3.1 as well. The heuristic prediction approach, including the enhancement from the post-execution analysis, speculates future accesses via various prefetching algorithms and collected data access history. These prefetching algorithms include well-known algorithms such as sequential prefetching, stride prefetching, Markov prefetching, etc. The pre-execution approach takes a pre-
compiler or programmer’s support to conduct speculative execution at runtime, and speculates future requests. At each level, a prefetch generator produces prefetch requests and enqueue them into a prefetch queue. The prefetch requests are scheduled to fetch data from the source to the destination in advance to mask the data-access latency and improve the data-access efficiency.

### 3.2 Cache-Memory Latency Reduction: A Hardware Approach

At the cache-memory stage, we employ a specialized hardware approach to prefetching data from main memory to various cache memory levels. We introduce a generic and prefetching dedicated cache structure, named Data-Access History Cache (DAHC), and explores comprehensive and adaptive heuristic prediction based prefetching. The heuristic prediction based prefetching includes a variety of prefetching strategies. These strategies are effective and efficient when accesses exhibit perceivable patterns. For instance, if accesses are non-contiguous but have a constant stride, stride prefetching works effectively to mask the latency. If the probability of repeated access addresses is high, Markov prefetching works well. On the other hand, these access patterns are application dependent and no single prefetching algorithm can achieve the best result for all sorts of applications. The adaptive support is essential for a well-performing heuristic prediction prefetching. The Data-Access History Cache can support a variety of prefetching strategies, and based on it, we further introduce feedback-controlled adaptive prefetching. The adaptive prefetching utilizes runtime information and makes adaptation to proper algorithms depending on the specific application access characteristics. Chapter 4 discusses the Data-Access History Cache design, prefetching methodology, adaptive prefetching mechanism, and the simulation verification in detail.
3.3 Memory-Disk Latency Reduction: A Software Approach

We employ a software approach of data prefetching to improving memory-disk stage data-access efficiency. We investigate both pre-execution analysis and post-execution analysis based data prefetching at this level. The pre-execution prefetching utilizes available computing capability and pre-executes certain code fragment to identify future requests. It is a more general prefetching strategy and is complementary to heuristic prediction prefetching when accesses do not follow certain patterns. The pre-execution prefetching is beneficial because, as we discussed in Chapter 1, the cost of computing power has been decreasing rapidly. Computing capability is enormous but data access is the identified bottleneck. This trend provides the need and possibility to conduct pre-execution to reduce data-access latency efficiently. The post-execution utilizes the past access history of applications, and performs analysis to identify the essential features of data accesses from an application. We extract these features and represent them with an abstraction, which is employed for future runtime prefetching. Both pre-execution and post-execution analysis based prefetching are able to explore data access concurrency with the computation well, and thus effectively hide access latency at memory-disk level. In the mean time, the traditional concerns with memory-disk level prefetching strategies, such as increased memory pressure, buffer cache pollution and increased communication congestion, have been remedied well by new technologies such as much larger memory at low cost, dedicated memory portions for buffer cache, and much higher I/O bandwidth and disk-level buffer cache. All these technique trends provide promising opportunities for exploring the pre-execution and post-execution
analysis based data prefetching. Chapter 5 discusses the detailed design, challenges and solutions of the memory-disk stage approach of improving data-access efficiency.

3.4 Integration of Hardware and Software Approach

The specialized hardware and software approaches of the two-stage latency reduction mechanism are integrated naturally. The cache-memory stage latency reduction is a complete hardware solution, providing comprehensive data prefetching with the provision of specialized Data-Access History Cache and feedback-controlled adaptive prefetcher. When a processor manufacturer provides the integration of the specialized hardware approach on chip, the functionality of cache-memory stage latency reduction can be automatically exploited for applications. The software approach is implemented and delivered as a library solution that integrates with compilation analysis, operating system and middleware solutions. It takes advantage of these existing and new techniques to further build a comprehensive software approach of improving data-access efficiency.
As shown in Figure 3.2, the ambitious goal of the Hybrid Adaptive Prefetching architecture is to provide a systematic solution to boosting data-access performance by leveraging specialized hardware, compiler analysis, operating system and middleware support. As the experimental result verifies, the Hybrid Adaptive Prefetching architecture is beneficial for a variety of applications. It serves as an effective data-access accelerator for data-intensive, high-performance and high-end computing system. The Hybrid Adaptive Prefetching architecture is a promising approach to advancing the state-of-the-art of data-access technology.
CHAPTER 4

IMPROVING CACHE-MEMORY STAGE DATA-ACCESS EFFICIENCY

This chapter discusses the cache-memory stage latency reduction solution of the Hybrid Adaptive Prefetching architecture in detail. We study specialized hardware approach of data prefetching for improving data-access efficiency, and discuss related issues and solutions. These studies and results have been published in refereed conference proceedings and journals in [10][12][22][86][87].

To reduce cache-memory stage data-access latency by exploiting the benefits of comprehensive and adaptive data prefetching, we first propose a novel cache structure, named Data-Access History Cache (DAHC) [22]. The DAHC serves as a fundamental structure dedicated to data prefetching at cache-memory stage, and behaves as a cache for recent reference information instead of as a traditional cache for either instructions or
data. Theoretically, it is capable of supporting any history-based prefetching algorithms, especially adaptive and aggressive approaches. With DAHC, we study the methodology of supporting numerous data prefetching strategies. We also introduce a *feedback-controlled adaptive prefetching* considering diverse application features and maximizing the benefits of data prefetching at runtime. We present the specialized hardware design and analyze the hardware budget of the cache-memory stage solution of improving data-access efficiency. We have carried out extensive simulation experiments with an enhanced widely-used SimpleScalar simulator to validate the design of DAHC-based and feed-back controlled data prefetching, and to verify the performance gain. The simulation testing has confirmed that the HAP architecture and specialized hardware approach of cache-memory stage solution are capable of largely improving data-access efficiency.

### 4.1 Data-Access History Cache Design and Methodology

The main purpose of the specialized cache structure, Data-Access History Cache, or DAHC in short, is to track recent data-access history and maintain the correlations from different perspectives. These histories and correlations are valuable information for data prefetching, especially for aggressive and adaptive prefetching strategies.

The design rationale of DAHC is that heuristic prediction of data prefetching algorithms must rely on correlations within either program counter (PC) stream or data address stream, or both. Thus, DAHC is designed to have three specialized hardware tables: one data-access history table (DAH) and two index tables (PC index table and address index table). The DAH table accommodates access history details, while the PC index table and the address index table maintain correlations from the PC and data address stream viewpoints respectively. A prefetching implementation can access these
two tables to obtain the required correlations as necessary. Figure 4.1 illustrates the general design of DAHC and a high-level view of how it can be applied to support various data prefetching algorithms. In existing work of hardware data prefetching [16][27][28][31][40], only very limited correlations are maintained, which largely limits the prefetching accuracy, coverage and aggressiveness. Moreover, they only target a specific algorithm and lack the support for diverse prefetching strategies and adaptation among different strategies. The existing work has difficulty in applying to diverse applications and effectively reducing cache-memory stage data-access latency dynamically at runtime.

Figure 4.1. DAHC General Design and High-level View

The detailed design of DAHC is shown in Figure 4.2 through an example. The DAH table consists of PC, PC_Pointer, Addr, Addr_Pointer and State fields. PC and Addr fields store the instruction address and data address separately. The PC_Pointer and Addr_Pointer point to an entry where the last access from the same instruction or the last access of the same address is located. Therefore, PC_Pointer and Addr_Pointer link all
accesses from the instruction stream and data stream perspectives. This design offers the fundamental mechanism to detect potential correlations and access patterns. The State field maintains state machine status used in prefetching algorithms. Various algorithms could occupy different bits of this field for maintaining their own states. The length of this field is implementation dependent, and the usage is decided by prefetching strategies.

The PC index table has two fields, PC and Index. The PC field represents the instruction address, which is a unique index in this table. The Index field records the entry of the latest data access in the DAH table from the instruction stored in the correspondent PC field. It is the connection between the PC index table and the DAH table. The address index table is similarly defined. For instance, in Figure 4.2, the DAH table captured four data accesses, three of them issued by instruction 0x403C20 (stored in the PC field) and one by instruction 0x4010D8. The instruction 0x403C20 accessed data at address 0x7FFF8000, 0x7FFF8004 and 0x7FFF800C in sequence, which is shown through the Addr and PC_Pointer fields. The instruction 0x403C20 and 0x4010D8 are also stored in the PC index table, and the corresponding Index field tracks the latest access from the DAH table, which are entry 3 and 1 respectively. The address index table keeps each accessed address and the latest entry, as shown in the bottom left of the figure, thus connecting all the data accesses on the basis of the address stream. Both PC index table and address index table can be implemented in a variety of ways including a fully associative structure and a set-associative structure. Notice that DAHC design is general and it does not imply any restriction to the system environment. It works in Chip-Multiprocessing (CMP) or Simultaneous Multithreading (SMT) environment, as well as in multiple applications environment.
Figure 4.2. DAHC Blueprint: PC Index Table, Address Index Table and DAH table

Figure 4.3 shows a snapshot of DAHC after capturing more data accesses. The PC index table, address index table and DAH table are updated. The latest access entries for instruction 0x403C20 and 0x4010D8 become index 9 and 8, respectively. The address accessed and the corresponding entry are updated in the address index table. In this case, a complex structured stride pattern of (4, 8, 4, 8) is detected for instruction 0x403C20 after examining address 0x7FFF8000, 0x7FFF8004, 0x7FFF800C, 0x7FFF8010 and 0x7FFF8018; therefore, data at address 0x7FFF801C and 0x7FFF8024 could be prefetched to memory in advance to avoid cache misses when 0x7FFF801C and 0x7FFF8024 are accessed as predicted. Such a complex structured pattern is a general case of stride pattern. However, the conventional stride prefetching approach [16] is unable to detect it without DAHC support. This example also shows an address correlation between 0x100003F8 and 0x100003FA, which is often observed and utilized for prediction in the Markov prefetching algorithm [40]. The following section discusses data prefetching methodologies based on DAHC.
The DAHC provides a straightforward and an effective prototype design of a prefetching-dedicated structure. It is a cache for data-access information compared with conventional cache for instructions or data. The proposed DAHC can be placed at different levels for various desired data prefetching. For instance, it can be used to track all accesses to first level cache and to serve as a L1 cache prefetcher, as shown in Figure 4.4. It can also be placed at the second level cache and serves as a L2 cache prefetcher only. The straightforward design makes the implementation uncomplicated. The hardware implementation of the DAHC should be a specialized physical cache, like victim cache or trace cache. The PC index table and the address index table can be implemented with any associativity such as 2-way or 4-way. Since the index tables usually have less valid entries than the DAH table, it is unlikely that some entry is replaced due to a conflict miss. Even if a conflict miss occurs, it does not affect the correctness except discarding some access history. The DAH table can be implemented with a special structure where history information can be stored row by row and each row
can be located by using its index. The logic to fill/update the DAHC comes from the cache controller. The cache controller traps data accesses at the monitored level and keeps a copy of the access information in the DAHC. If the DAH table is full, a victim entry will be selected and evicted out. The PC index table and the address index table are updated as well for consistency. The required DAHC size for normal applications’ working set is trivial. For instance, if we suppose a DAHC with 1024 entries is implemented, which is a reasonable window size for a regular working set, then the required DAHC size is about 22KB. Our experiments simulated DAHC functionalities, and the conclusion is that DAHC is feasible in terms of hardware implementation.

![Image](4.4.png)

**Figure 4.4. Data-Access History Cache Serving for L1 Data Cache**

### 4.2 DAHC-based Data Prefetching Mechanism
4.2.1 Stride Prefetching. Stride prefetching predicts future references based on strides of recent references. This approach monitors data accesses and detects constant stride access patterns. Stride prefetching is usually implemented with a Reference Prediction Table (RPT) [16][31] as shown in Figure 4.5. RPT acts like a separate cache and holds data reference information of recent memory instructions. Since stride prefetching involves tracking the difference between two consecutive accesses and predicting the next access based on the stride, it is straightforward to design such an RPT table for stride prefetching implementation. Each entry in RPT is the instruction address, and it contains the last access address, the stride and the state transition information to predict future accesses. The right part of Figure 4.5 shows the state transitions. Once a pattern enters steady state or remains at steady state, which means a constant stride is found, a prefetch is triggered. The prefetched data address is simply calculated by adding the stride to the previous address.

Although RPT is effective for capturing constant stride of data accesses, it has several limitations. The first limitation is that RPT only calculates the stride between two consecutive accesses. It is hard to detect variable strides and impossible to find complex patterns, such as a repeating pattern of length $n$ (e.g., 2, 4, 8, 2, 4, 8, …). Those complex patterns are common in user-defined data types. The second limitation is that RPT only tracks the last two accesses and omits many useful history references; thus, the accuracy in detecting patterns is relatively low. Those issues are addressed well in the proposed DAHC structure. Since DAHC tracks a large set of working histories, it is capable of detecting variable strides. Those detailed histories can also be used to improve the
accuracy of stride detection. Moreover, DAHC makes detection of complex structure patterns possible, as discussed in previous examples.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Prev_addr</th>
<th>Stride</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>450</td>
<td>50000</td>
<td>4</td>
<td>Transient</td>
</tr>
<tr>
<td>520</td>
<td>60500</td>
<td>8</td>
<td>Stable</td>
</tr>
</tbody>
</table>

Figure 4.5. Reference Prediction Table and State Transition Diagram

Stride prefetching can be implemented with the DAHC as follows. First, when a data access happens at monitoring level and is tracked by added DAHC component and related logic (see Section 4.1 for more details), the instruction address is searched for in the PC index table. If the instruction address does not match any entry in the PC index table, which means it is the first time that we see this instruction address in current working window, no prefetching action is triggered. If the instruction address matches one entry (it will match only one entry because the entries in index tables are unique), we follow the index pointer to traverse previous access addresses and detect whether a strided pattern or a structured pattern is present. If a pattern is detected, one or more data blocks are prefetched to data cache or a separate prefetch cache. The prefetching degree and prefetching distance can vary depending on the actual implementation. Finally, a new entry with this data access is created and inserted into the DAH table. The PC index table and address index table are updated correspondingly. Notice that the approach described above is enhanced stride prefetching with detection of variable and complex stride...
patterns. The conventional stride prefetching [16][31] can be implemented by detecting constant strides only.

4.2.2 Markov Prefetching. Markov prefetching is another classical prefetching strategy. The Markov prefetching algorithm builds a state transition diagram through past data accesses. The probability of each transition from one state to another state is calculated and updated dynamically. The algorithm assumes the future data accesses might repeat the histories. Therefore, once a new data access is captured, the future references predicted from the state transition diagram are prefetched in advance. For instance, Figure 4.6 shows the correlation table and state transition diagram for the data access stream 7FFF8000, 1010FF00, 10B0C600, 7FFF8000, 7FF3CA00, 7FFF8000, 10B0C600 and 7FF3CA00.

![Figure 4.6. Markov Prefetching Correlation Table and State Transition Diagram](image)

The conventional Markov prefetching strategy treats all history accesses with the same weight. In practice, we usually give the highest weight to the latest access. This approach is essentially a combination of Markov model and LAST model [29]. The rationale is that the next data access is most probably the one that had followed the
current access in the nearest past. For example, if we have a sequence of accesses to address A, B, A, C, D, A, then it is likely that the next access is C. With DAHC support, Markov prefetching can be implemented as follows. First, the data reference address is searched for within the address index table. If the newly accessed address does not match any existing entries, it is simply inserted into the DAH table. The PC index and address index table are also updated. If it matches an entry in the address index table, then we insert it to the DAH table and walk through the DAH table following the index and address pointer as shown in Figure 4.7. Each address next to these entries we visit is a prefetching candidate because each of this address was immediately accessed following the present access address in histories. Similar as in stride prefetching, different prefetching degree and prefetching distance can be supported depending on the actual implementation. If the prefetching degree is greater than one, we fetch multiple continuous data addresses following these entries we visit. We can also increase prefetching distance to initiate multiple visits. Continuing with the previous example and as shown in Figure 4.7, if a new data access address is 0x10B0C600, then a new entry is inserted into the DAH table at index 7, and the address index table is updated. After we walk through the DAH table following index 7, pointer 5 and pointer 2, data at address 0x7FF3CA00 and 0x7FFF8000 are prefetch candidates if we set prefetching degree as one and prefetching distance as two. Notice that Markov prefetching builds state transition based on data addresses. It does not need to use the state field.
4.2.3 **Aggressive Prefetching Strategy.** Since the DAHC maintains recent accesses in detail and the correlation among them, it is more powerful than supporting traditional prefetching approaches such as stride prefetching and Markov prefetching. It can support many other history-based prefetching strategies including more aggressive prefetching algorithms. It is an easy task to implement aggressive strategies with the DAHC because the DAHC is designed to support aggressive strategies naturally. The Multi-Level Difference Table (MLDT) prediction algorithm is such a representative aggressive strategy [86]. This prediction strategy forms a difference table of depth $d$ of recent data accesses. Figure 4.8 demonstrates an example of the difference table. If a constant difference can be found in the first depth, which means a constant stride is found among data access histories, then the $k^{th}$ future access from access $A_r$ is predicted as $A_{r+k} = A_r + k \cdot B$, where $B$ is the constant difference among accesses. Some polynomial formula is used to predict the future access for general cases. For example, if a constant difference is found in the third depth, the future access is predicted as

$$A_{r+k} = A_r + k \cdot B_{r-1} + \frac{k \cdot (k+1)}{2} \cdot C_{r-2} + M_k \cdot D.$$
Here \( M_k = \frac{k}{3} \times (k-1) \times (k-2) + k^2 \), where \( k = 1, 2 \ldots \)

<table>
<thead>
<tr>
<th>References</th>
<th>A_0</th>
<th>A_1</th>
<th>A_2</th>
<th>A_3</th>
<th>A_4</th>
<th>A_5</th>
<th>A_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>First differences</td>
<td>B_0</td>
<td>B_1</td>
<td>B_2</td>
<td>B_3</td>
<td>B_4</td>
<td>B_5</td>
<td></td>
</tr>
<tr>
<td>Second differences</td>
<td>C_0</td>
<td>C_1</td>
<td>C_2</td>
<td>C_3</td>
<td>C_4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Third differences</td>
<td>D_0</td>
<td>D_1</td>
<td>D_2</td>
<td>D_3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.8. Example of Difference Table

MLDT strategy is similar to existing stride prefetching but is more aggressive since it searches references up to depth \( d \). The stride prefetching is the special case where depth equals one. In addition, this method finds sets of repeating differences and ultimately finds the actual pattern in the accessing structures with variable stride data access patterns. For variable stride patterns, MLDT searches for regularity among data references by finding a deeper difference table. It can also be extended to find repeating sets of strides (e.g. 4, 8, 4, 8, 4, 8, 4, 8, 4…) at each level of difference table. Our proposed DAHC provides an implementation approach for the MLDT prefetching algorithm. First, when we see a data access at monitoring level, we check this access’s instruction address with the PC index table. We update the DAH, PC index and address index tables as necessary. Second, we follow the index pointer and walk through the DAH table to find out previous accesses. These operations are similar as in stride prefetching case. The difference between MLDT prefetching and stride prefetching is that multiple level differences are calculated to detect if any constant stride, variable stride or complex structure pattern exists in each level, which means we perform a stride prefetching at each stride difference level. If a pattern is detected at some level, we stop
going to further levels. If we continue to the further level, we calculate the strides of next level and they become the strides we deal with. Therefore, we always work with one level of stride similarly as in the conventional stride prefetching case. Figure 4.3 shows an example where a complex structure pattern (4, 8, 4, 8) is detected when we perform the MLDT prefetching with the DAHC.

4.3 Adaptive Hardware Data Prefetching

In this section, we present the design of an intelligent adaptive prefetching [17]. This prefetching methodology leverages the powerful functionality provided by DAHC, and support multiple prefetching algorithms, but dynamically adapt to those algorithms that perform well at runtime. The essential idea is using runtime feedback and evaluation to direct the dynamic adaptation. We first present the evaluation metrics, and then discuss their hardware implementation and how to direct adaptation.

4.3.1 Evaluation Metrics. While extensive studies exist in data prefetching, few studies present a formalized metric to evaluate the effectiveness of prefetching algorithms. We analyze and sort out the essential and the most critical criteria to model prefetching evaluation and present a formal definition in this chapter. These metrics provide a comprehensive evaluation of hardware prefetching methodologies. These metrics can be independently used to evaluate a prefetcher in addition to commonly used metrics, such as an IPC (Instructions Per Cycle) speedup metric.

4.3.1.1 Prefetch Precision. The first and widely-adopted metric is termed *prefetching precision*, or *prefetching accuracy*. This metric characterizes how many percent of prefetches are actually accessed by demand requests, thus reflects how accurate the
prefetch requests and the prefetching algorithms are. We present a formal definition of prefetching precision as the follows.

**Definition 1.** Prefetching precision is defined as the ratio between the number of distinct prefetched cache lines that are accessed by at least one demand request after being prefetched in and before being replaced out over the number of total prefetched cache lines.

By this definition, the prefetching precision models the accuracy of the prefetcher, i.e. the percent of useful cache lines in the overall cache lines prefetched. Notice that we clearly define a useful prefetch as being accessed at least once when the prefetched cache line resides in the prefetch destination. Therefore, a repeated access to that cache line in the current lifecycle does not account into the total number of useful prefetches. However, if a cache line is prefetched again into the destination after being displaced, and get hit, this scenario will contribute to one useful prefetch. We refer these cache lines brought in by prefetch and accessed by demand requests as *prefetch hits*, in contrast with *demand hits*, those lines fetched by demand and hit again by other requests.

The prefetch precision should be considered as the most critical metric to evaluate or direct prefetch adaptation, as it describes the cost-efficiency of a prefetcher well. Taking prefetch precision into consideration, a too aggressive but not accurate enough prefetcher should be largely avoided because it might produce a large number of useless prefetches, which significantly wastes resources, such as power and cache line slots. Instead, this metric favors a prefetcher with high-confidence. The prefetch precision metric strongly suggests the prefetcher should focus on identifying the correct access pattern and make a highly-accurate prediction. This way maximizes the hardware
investment on the prefetecher, and achieves a high cost-efficiency. An ideal prefetecher will produce a prefetching precision with value 1. In practice, the prefetching precision has a range from 0 to 1.

Though the prefetch precision is critical and straightforward, it merely describes one aspect of the problem under study – the prefetch precision does not quantify how effective the prefetecher is, i.e. how many misses among the overall misses are hided. The next metric we formalize addresses this limitation.

4.3.1.2 Prefetch Coverage. The prefetch coverage metric is introduced to complement prefetch precision and quantify the other aspect of how well a prefetecher works. We formalize the prefetch coverage definition as follows.

**Definition 2.** Prefetch coverage is defined as the ratio of the number of misses reduced due to prefetches over the total number of misses that will occur without prefetching.

As the definition states, the prefetch coverage focuses on quantifying the ratio of the misses reduced, i.e. how wide a prefetecher covers the demand misses that are supposed to occur without the assistance of prefetching. A highly-accurate prefetecher does not necessarily provide a wide coverage. This is because such a prefetecher could be very conservative and takes action only when the prefetecher has a high-confidence prediction. This conservativeness results in a high precision but the effectiveness in terms of miss reduction ratio is low. The vice versa holds as well, i.e. a widely-covering prefetecher is not necessarily highly-accurate since the prefetecher could be very aggressive (such as with a great prefetch degree) to improve the coverage while sacrificing the
precision. In essence, the prefetch coverage and prefetch precision are complementary to each other, and together they quantify the effectiveness of a prefetcher from two aspects.

4.3.1.3 Prefetch Pollution. While prefetch precision and prefetch coverage can reflect the prefetch effectiveness, or the positive side, of a prefetching algorithm well, they do not characterize the negative side of an algorithm. Cache pollution [94] is considered the most critical downside of prefetching. When a cache line that is replaced by a prefetched line is later accessed by a demand request, cache pollution occurs. Such a cache miss will not happen if without the interference of prefetching. This scenario, cache pollution, is referred as a negative side-effect of prefetching. We present a formal definition to describe prefetch pollution.

**Definition 3.** Prefetch pollution is defined as a ratio of the number of additional demand misses caused by prefetching that will not occur without prefetch interference over the number of misses that will occur without prefetching.

According to this definition, prefetch pollution quantify the percent of extra demand misses due to prefetches, which means that those demand misses will not occur if prefetching is not adopted. The occurrence of these misses is due to the limited size of cache and the replacement of useful cache lines by prefetched cache lines. Together with prefetch precision and prefetch coverage, the prefetch pollution completes a three-tuple, \((\text{precision}, \text{coverage}, \text{pollution})\), to evaluate a prefetching algorithm. These three metrics are complementary to each other, and assess an algorithm from both positive and negative aspects. Some literatures separate other metrics, such as lateness [82]. We observed that these metrics are well covered in 3-tuple, \((\text{precision}, \text{coverage}, \text{pollution})\).
A separation of these additional metrics might be helpful, but might also cause confusion.

4.3.2 Evaluation Metrics: On-the-Road. We have presented the formal definition of a three-tuple metric to evaluate and direct a prefetcher in the previous section. We discuss the hardware design and realization of these metrics in this section.

4.3.2.1 Realizing Prefetch Precision Metric. To realize the prefetch precision metric, we need two statistics counters for each evaluated algorithm, one counter for the prefetch hits, and one counter for overall prefetches. We refer these two counters as \textit{prefetch\_hits} and \textit{prefetch\_total}. In addition, to collect the statistic of prefetch hits, we need to distinguish the cache lines prefetched from demanded. This requirement results in a major hardware storage budget. We assume each cache line in the prefetch destination (L2 cache in this dissertation) has one extra prefetch bit to represent whether this line is prefetched or fetched for each evaluated algorithm. When a cache line is prefetched into destination, this prefetch bit is set. If this cache line is ever accessed during its lifetime in the cache (after being prefetched and before being displaced), the \textit{prefetch\_hits} counter is increased and the prefetch bit is reset. By this way, the prefetched cache line is not counted as multiple hits even when it is accessed multiple times, which is consistent with the definition. A simple reason behind this decision is that the first hit acts like a regular demand request and fetches in data, and the future accesses will hit in cache. The actual saving of the prefetching is the first access. If a cache line is brought in by a normal demand request, the prefetch bit is not set. The combinatorial logic to maintain this prefetch bit, set and reset, is simple, and the hardware implementation of this logic is trivial. Notice that if a cache line is prefetched by multiple algorithms, the corresponding
prefetch bits will be set and a hit of this cache line will attribute to the statistics of each corresponding prefetecher.

4.3.2.2 Realizing Prefetch Coverage Metric. The \textit{prefetch\_hits} counter designed in above can also be used in calculating the prefetch coverage. This is because that the statistics the \textit{prefetch\_hits} counter collects is the number of misses reduced due to prefetches. To compute the prefetch coverage, we simply need another statistics, the overall misses that will occur without prefetching. We design another counter, named \textit{demand\_misses}, to collect the number of misses that occur even with prefetching. The \textit{prefetch\_hits} counter represents the number of misses saved by prefetch, and the \textit{demand\_misses} counter represents the number of misses that still occur. The sum of both two counters is the required total number of misses that will occur without prefetching. The prefetch coverage is computed as:

\[
\text{prefetch\_coverage} = \frac{\text{prefetch\_hits}}{\text{prefetch\_hits} + \text{demand\_misses}}.
\]

4.3.2.3 Realizing Prefetch Pollution Metric. It is more challenging to collect the prefetch pollution statistics than to collect prefetch precision and coverage. The reason is that we never know whether the replaced cache line due to a prefetch will be used in future or not. An optimal solution to collecting the prefetch pollution metric is tracing down all these cache lines and detecting whether any future requests will access these lines. If such a cache line is detected, which means that this cache line is replaced out due to a prefetch but is needed by a demand request, a scenario of cache pollution is detected. However, such an optimal solution will require infinite-size storage to keep all past cache lines replaced out due to prefetches. This approach is not feasible in practice. Motivated
from existing studies, we utilize a Bloom filter \cite{3,72,82} to estimate the percentage of cache pollution.

Suppose the cache line size is 64B, and a cache block address is 26 bits. The pollution filter splits 26 bits into two parts, high-order 13 bits and low-order 13 bits. These two parts are fed into an XOR logical unit, and a filtered address, with 13 bits, is the output. This filtered address is used to index a bit vector and set the corresponding bit in the vector. We use this filter to estimate the pollution. We track each cache line that is replaced out due to a prefetch and feed this cache line address into the filter. A corresponding bit of the bit vector is set. We also feed cache miss addresses into the filter, and if the corresponding bit is set, we estimate this cache line was in the cache but was replaced out due to a prefetch. After a cache pollution is detected, the corresponding bit is reset, as the cache line is fetched back into cache. We use another pollution counter for each evaluated algorithm to accumulate the prefetch pollution statistics.

4.3.3 Metrics Collection. We periodically collect these three metrics, precision, coverage and pollution discussed above in order to make adaptation decision. The proposed adaptive prefetching is designed to have two phases, metrics collecting phase or learning phase, and stably prefetching phase or working phase. In the learning phase, we run all supported prefetching algorithms simultaneously, and collect all statistics of each prefetching algorithm. In the end of learning phase, all three metrics are computed for prefetching algorithm evaluation. In the working phase, only the adaptively selected algorithm will be running, and all counters and pollution estimator are cleared and turned off. The decision to choose the working algorithm is discussed in the following section.
The switch between these two phases is controlled by phase timers, or phase counters. We view each cache miss as one time tick to the phase control. This is a feasible design choice to control the adaptive prefetcher behavior because the number of cache misses can fairly represent how the prefetcher should react. We empirically set the learning phase $1/8^{\text{th}}$ of the working phase, which means we collect statistics and make adaptive selection decision in one unit of time, while let the selected algorithm(s) work in eight units of time.

4.3.4 Adaptive Selection. After collecting statistics and computing the metrics, we make the decision to adaptively select the suitable prefetching algorithms. In order to make the decision, we evaluate the performance of prefetching algorithms, the precision, coverage and pollution as either high or low. This evaluation is done by comparing the runtime statistics against with empirically preset threshold. If it is above the threshold, we classify the statistics as high. In contrast, if it is below the threshold, we classify it as low. In simulation experiments, the threshold to distinguish a high/low prefetch precision, coverage and pollution are preset as 0.70, 0.3, and 0.2 respectively based on empirical experience. In practice, these thresholds can be tuned and determined in advance for any specific architecture. Figure 4.9 illustrates eight levels of prefetching algorithm performance.

Based on the prefetching algorithm evaluation performance table, we are able to dynamically identify and choose optimal prefetching algorithms. We introduce two different mechanisms to select the algorithms, best-strategy adaptive selection and multi-strategy adaptive selection.
The best-strategy adaptive selection always output the one performing best in the statistics collecting phase. This decision is made based on the algorithm evaluation and the performance table – we assign the lowest level with the highest priority. Within the same level, the accuracy has the highest weight, the coverage second, and the pollution last. This means that if we have multiple algorithms following into multiple levels, we choose the lowest level algorithms as the candidate. If we have multiple candidates, we favor the one with highest accuracy.

The best-strategy adaptation works with choosing a best strategy out of all supported algorithms, but the limitation is that it only chooses one algorithm even if multiple strategies are performing well too and can sometimes complement each other. Also this strategy always outputs one “relatively best” strategy, even though the best strategy might not work well enough at certain circumstances. We introduce another adaptation of multiple optimal strategy selection based on the evaluation and performance table. This adaptation choice uses the level as selection criteria. For instance, if we specify level 0 and level 1 as adaptation criteria, then we dynamically choose all

<table>
<thead>
<tr>
<th>Level</th>
<th>Precision</th>
<th>Coverage</th>
<th>Pollution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>1</td>
<td>H</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>2</td>
<td>L</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>3</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>4</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>5</td>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>6</td>
<td>H</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>7</td>
<td>L</td>
<td>L</td>
<td>H</td>
</tr>
</tbody>
</table>
these algorithms fall in these levels, and use them in the working phase. This selection can also control the quality of the selection. If none of algorithms following the specified criteria, then we do not have any algorithm performing in the working phase, until the algorithms are evaluated again in the next learning phase. The selection criteria are preset, for instance, as level 0, 1 and 2.

4.3.5 Hardware Cost. As discussed in previous sub-sections, we need three counters for each evaluated algorithm and two counters for all algorithms to collect the required statistics in order to direct the adaptation. Each counter can be implemented with a 32-bit register. The overall required storage for counters will be 56 bytes, if assuming to support four distinct algorithms simultaneously. In addition to the counters, we need one cache pollution estimator for each supported algorithm. The pollution estimator requires $2^{13}$ bits of storage for the bit vector. Therefore, the estimator consumes 1KB for each supported algorithm. The cache structure needs a slight modification to support adaptive selection of prefetching algorithm as discussed previously. The modification is one bit for each supported algorithm. For a typical 1MB L2 cache with 64 bytes cache line, we have 16,384 cache lines. To support one prefetching algorithm, the additional hardware cost will be 16,384 bits or 2KB. Therefore, as a normal case, to support adaptation among four algorithms, the overall hardware cost will be around 12KB. This hardware budget is trivial – as only around 1% compared to a regular 1MB L2 cache. However, as the simulation verifies, the adaptive prefetching can substantially reduce cache misses and improve the overall system performance.

As discussed partially in the previous section, the combinatorial logic to realize the proposed adaptive prefetching is not complicated as well. The major required
combinatorial logic resides in maintaining the prefetch bits within the cache line, maintaining statistics counters, filtering through prefetch pollution estimator, and adapting prefetching algorithms via the performance table. Maintaining prefetch bits is straightforward because it simply requires set/unset the corresponding bit according to whether a cache line is brought in due to a specific algorithm. Maintaining counters is a simple logic too. Filtering is slightly complicated, but as we show with the estimator, the hardware state machine can be easily described. Adapting the algorithm mainly needs comparison logic, which can be implemented effortlessly as well.

4.4 Simulation Methodology

We have conducted simulation experiments to study the feasibility of our proposed generic prefetching-dedicated cache, DAHC, for various prefetching strategies. Stride prefetching, Markov prefetching and MLDT aggressive prefetching algorithms were selected for simulation. We have also carried out simulation experiments to validate the design of adaptive hardware prefetcher based on DAHC and verify the performance gain. This section discusses the simulation methodology and experimental setup.

4.4.1 SimpleScalar Simulator and Enhancement. The SimpleScalar simulator [9] was enhanced with data prefetching functionality to demonstrate how different prefetching algorithms can be implemented with the DAHC. The SimpleScalar tool set provides a detailed and high-performance simulation of modern processors. It takes binaries compiled for SimpleScalar architecture as input and simulates their execution on provided processor simulators. It has several different execution-driven processor simulators, ranging from extremely fast functional simulator to a detailed and out-of-order issue simulator, called the sim-outorder simulator.
We chose the sim-outorder simulator for our experiments. Figure 4.10 shows our modified SimpleScalar simulator architecture. We introduced two new modules: DAHC module and Prefetcher module. The DAHC module simulated the functionality of the proposed DAHC. Monitored data accesses were stored in the DAHC. The DAHC cache controller is responsible for updating all three tables. The Prefetcher module implemented the prefetching logic and different prefetching algorithms. In this module, a prefetch queue, similar to the ready queue of the original sim-outorder simulator, was created to store prefetch instructions. Prefetch instructions are similar to load instructions with a few exceptions. The first exception is that the effective address of each prefetch instruction is computed based on a data access pattern and prefetching strategy instead of computing the address using an integer-add functional unit. Another exception is that when prefetch instructions proceed through the pipeline, it is not necessary to walk through writeback and commit stages, and prefetch instructions do not cause any exceptions (prefetch instructions are silent). These similarities and differences provide us the guidelines to handle prefetch instructions. The implementation of prefetching strategies based on the DAHC follows the discussion given in Section 4.2.

In addition to these two new modules, several existing modules were enhanced to incorporate the DAHC and data prefetching functionality. First, the simulator core module was revised to support the DAHC and Prefetcher modules. The pipeline was modified to have prefetching logic. The first improvement is each ready-to-issue load instruction is tracked to DAHC after the memory scheduler checks data dependencies. The prefetcher performs access pattern detection based on prefetching algorithms and makes prediction for future data accesses once a pattern is detected. Prefetch instructions
are thus enqueued to prefetch queue. Another improvement is in instruction issue phase. During this phase, when we have available issue bandwidth, i.e. if there is idle bandwidth after issuing normal instructions, the prefetch queue is walked through and prefetch instructions are allocated with functional units to fetch the predicted data to data cache. Second, the memory module was modified to introduce a prefetch command to the memory component in addition to a load and a store command. The cache module was augmented with prefetch access handlers. Prefetch accesses can be handled similarly to load instructions except prefetch accesses do not cause any exceptions. Some additional statistics counters were added for measuring the effectiveness of prefetching.

Figure 4.10. Enhanced SimpleScalar Simulator

To simulate the adaptive prefetcher, we have modified cache line structure to identify whether a cache line is brought in due to a prefetch, and which algorithm brought it in. We have added in all required counters to simulate the evaluation as well. The pollution estimator was simulated to provide the cache pollution statistics. The combinatorial logic was simulated to compare the evaluation of each algorithm, and
output dynamically chosen algorithms, with best-strategy and multi-strategy selections. The multi-strategy selection was configured as level 0, level 1 and level 2 algorithms.

4.4.2 Experimental Setup. We use the Alpha-ISA and configure the simulator as a 4-way issue and 256-entry RUU processor. The level one instruction cache and data cache are split. We configure L1 data cache as 32KB, 2-way with 64B cache line size. The latency is 2 cycles. L2 unified cache is configured as 1MB, 4-way with 64B cache line size. The latency of L2 cache is 12 CPU cycles. The DAHC is set as 1024 entries, and the replacement algorithm is FIFO. Both index tables are simulated with 4-way associative structures. We assume each DAHC access, such as a lookup within index tables, costs one CPU cycle. This should be a reasonable assumption for a small 4-way cache. We also assume a traversal within DAH table costs one cycle. If a prefetching algorithm needs to traverse multiple locations to make predictions, it consumes multiple cycles. The prefetch queue is set as 512 entries. Table 4.1 shows the configuration of our simulator.

<table>
<thead>
<tr>
<th>Table 4.1. Simulator Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
</tr>
<tr>
<td>Issue width</td>
</tr>
<tr>
<td>Load store queue</td>
</tr>
<tr>
<td>RUU size</td>
</tr>
<tr>
<td>L1 D-cache</td>
</tr>
<tr>
<td>L1 I-cache</td>
</tr>
<tr>
<td>L2 Unified-cache</td>
</tr>
<tr>
<td>Memory latency</td>
</tr>
<tr>
<td>DAHC</td>
</tr>
<tr>
<td>Prefetch queue</td>
</tr>
</tbody>
</table>

4.5 Experimental Results and Performance Analysis
We present the experimental and analytical results in this subsection.

4.5.1 **Matrix Multiplication Simulation.** We first set up experiments to test the enhanced SimpleScalar simulator with DAHC-based data prefetching functionality. The prefetching strategy was set as the MLDT algorithm. Matrix multiplication was selected as the application because it is widely used in scientific computing and the correctness of its output results is easy to verify. The size of matrices was set as $200 \times 200$. We randomly generated the input, conducted simulation and then compared the output result with standard output to verify the correctness of the enhanced simulator. The correctness was also validated through checking the number of instructions (normal instructions) issued by the original and the enhanced version. The simulation results are shown in Table 4.2. The simulation time is the elapsed time for simulation (how much time the simulator spent in simulating). The results confirm that the enhanced SimpleScalar simulator worked correctly, and cache misses were reduced significantly through DAHC-based data prefetching.

<table>
<thead>
<tr>
<th></th>
<th># of instructions</th>
<th>Simulation Time</th>
<th>L1 cache misses</th>
<th>L1 replacements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>622140213</td>
<td>12633</td>
<td>1031047</td>
<td>1030023</td>
</tr>
<tr>
<td>Enhanced</td>
<td>622140213</td>
<td>13469</td>
<td>28772</td>
<td>1084326</td>
</tr>
</tbody>
</table>

4.5.2 **SPEC CPU2000 Benchmark Result of DAHC-based Prefetching.** We conducted several sets of SPEC CPU2000 benchmark [112] simulation for performance evaluation. Twenty-one of the total twenty-six benchmarks were tested successfully in
our experiments. The other five benchmarks (apsi, facerec, fma3d, perlbmk and wupwise) had problems working under the SimpleScalar simulator (even in the original simulator) and did not finish the test.

The target of the first set of experiments was to compare the performance gain of traditional RPT-based stride prefetching approach and enhanced DAHC-based stride prefetching approach. Figure 4.11 shows the experimental results. The first bar in each test represents the level-one cache miss rate of the base case in which no prefetching was performed. The second and the third bar represent the miss rate in the case of RPT-based conventional stride prefetching and enhanced DAHC-based stride prefetching, respectively. As shown in Figure 4.11, the traditional approach reduced miss rates, and the enhanced approach reduced miss rates further. The rationale comes from that, with DAHC support, enhanced stride prefetching is able to detect complex structured patterns, and in addition, the prediction accuracy was improved through observing more histories. In contrast, many important and helpful histories were not considered and not fully utilized in traditional stride prefetching based on RPT.

Figure 4.12 compares L1 cache miss rates of all tested SPEC CPU2000 benchmarks for the base case and three prefetching cases. This set of experiments showed that DAHC-based data prefetching worked well and the cache miss rates were reduced obviously in most cases. Among the three prefetching strategies, both stride and aggressive MLDT algorithms reduced a large ratio of miss rates. The MLDT algorithm was slightly better than stride prefetching because it searches more levels to find patterns among accesses. The Markov prefetching performed worse than the other two in most cases. One possible reason is that Markov prefetching requires a large set of states to
characterize the probability of transition among accesses well. If the state diagram space is limited, it is hard for the Markov prefetching to guarantee the accuracy and coverage.

![Graph showing L1 Cache Miss Rate comparison between Base Case, Strided with RPT, and Strided with DAHC.](image)

**Figure 4.11. Stride Prefetching With RPT vs. Stride Prefetching With DAHC**

Figure 4.13 illustrates L1 cache replacement rate in these tests. Cache pollution is considered a side effect of prefetching. An incorrect prediction brings a useless data block to cache and might replace useful data. With DAHC support, the prefetching accuracy increases by taking advantage of all available history information. As we can see from Figure 4.13, the replacement rate only increased slightly in DAHC-supported data prefetching.
Figure 4.12. L1 Cache Miss Rate of SPEC2000 Benchmarks

Figure 4.13. L1 Cache Replacement Rate of SPEC CPU2000 Benchmarks
Figure 4.14 shows the overall IPC (Instructions Per Cycle) improvement brought by three prefetching strategies: stride, Markov and MLDT prefetching based on DAHC.

The experimental results demonstrated that the IPC value was improved considerably in most cases. The figure also reveals that even though MLDT achieved the best cache miss rate reduction in almost all cases, the IPC improvement was not always best. The stride prefetching outperformed the MLDT in the applu, crafty, gcc, gzip, lucas, mcf, parser, swim, twolf and vpr benchmarks. This is because MLDT involves more prefetching overhead for its aggressiveness due to more DAHC accesses. When we measured the overall system performance gain in IPC value, it paid for its additional overhead compared to stride prefetching. Another interesting fact shown in Figure 4.14 is
that Markov strategy outperformed the other two in the bzip2, eon and vortex benchmarks. These facts confirmed that different strategies are desired for different applications to obtain the best prefetching benefits. It is necessary to support diverse algorithms and adapt to them dynamically based on distinct application features. This observation has also been confirmed by our on-going experimental tests with best-strategy and multi-strategy adaptive prefetching. The DAHC provides the essential structure support for these adaptive strategies.

4.5.3 SPEC CPU2000 Benchmark Result of Adaptive Prefetching. To evaluate the adaptive hardware prefetcher based on DAHC, we first study the cache miss rate reduction with various prefetching algorithms and the adaptive prefetching. We apply the DAHC and adaptive prefetcher at L2 cache level for evaluation in this section to demonstrate that both specialized hardware structure can be used at different levels of cache. We also add in sequential prefetching for comparison.

Figure 4.15 plots the L2 cache miss rate reported by the simulator for the entire twenty-one benchmarks. This series of tests were conducted under eight cases, including the base case (without data prefetching), the cases with individual sequential, strided, Markov and MLDT data prefetching, the cases with best-strategy and multi-strategy adaptation and the case with all supported prefetching algorithms running simultaneously.

As clearly shown from the results, different applications exhibit distinct access patterns, and thus the cache miss rate reduction of various prefetching algorithms have large variations. For instance, sequential prefetching significantly reduced the misses for equake, gap, mgrid and swim benchmarks, while not for others like ammp, art, galgel and
gcc. Instead, the strided prefetching performed extremely well for ammp, applu, lucas, mcf and etc., and Markov prefetching had considerable miss reduction for bzip2, eon and vortex benchmarks. The MLDT prefetching usually achieved a better miss reduction than strided prefetching, but still not for all benchmarks. These observations confirm that a dynamic and smart prefetcher is desired to be able to adapt to different application features at runtime to achieve a better prefetching result. Adopting a fixed prefetching strategy can not be optimal.

![Figure 4.15. Cache Miss Rate of SPEC-CPU2000 Benchmarks](image)

The best-strategy and multi-strategy adaptive prefetching have demonstrated their strength through the simulation. From the reported miss rate results, we can tell that the
best-strategy can almost achieve the best miss rate reduction from all four supported algorithms. This is because that this strategy can effectively identify the suitable prefetching strategy for current application access pattern. The multi-strategy adaptation can sometimes achieve an even better miss rate reduction, such as in the applu, gap, lucas, mesa and sixtrack benchmarks. The investigation shows that this further reduction is due to two or three well-performing prefetching strategies the multi-strategy adaptation identified and selected at different stages for these benchmarks. These optimal strategies are all able to generate effective prefetches at a specific stage. Table 4.3 lists the primary prefetching algorithms identified and selected by the proposed adaptive hardware prefetching for different benchmarks at runtime.

<table>
<thead>
<tr>
<th></th>
<th>ammp</th>
<th>applu</th>
<th>art</th>
<th>bzip2</th>
<th>crafty</th>
<th>eon</th>
<th>equake</th>
<th>galgel</th>
<th>gap</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-S</td>
<td>ST</td>
<td>ST</td>
<td>MT</td>
<td>MK</td>
<td>ST</td>
<td>MK</td>
<td>SQ</td>
<td>MT</td>
<td>SQ</td>
<td>ST</td>
</tr>
<tr>
<td>M-S</td>
<td>ST</td>
<td>ST,</td>
<td>MT</td>
<td>MT,</td>
<td>ST,</td>
<td>MK</td>
<td>SQ</td>
<td>ST,</td>
<td>SQ</td>
<td>ST</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>SQ</td>
<td>SQ</td>
<td>ST</td>
<td>MT</td>
<td>SQ</td>
<td>ST</td>
<td>MK</td>
</tr>
<tr>
<td>lucas</td>
<td>ST,</td>
<td>SQ</td>
<td>ST</td>
<td>SQ</td>
<td>ST</td>
<td>MT</td>
<td>ST</td>
<td>MT</td>
<td>MK</td>
<td>ST</td>
</tr>
<tr>
<td>mcf</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>SQ</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>MK</td>
<td>ST</td>
</tr>
<tr>
<td>mesa</td>
<td>MT</td>
<td>MT</td>
<td>ST</td>
<td>MK</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>MK</td>
<td>ST</td>
</tr>
<tr>
<td>mgrid</td>
<td>MT</td>
<td>MK</td>
<td>SQ</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>MK</td>
<td>ST</td>
</tr>
<tr>
<td>parser</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>ST</td>
</tr>
<tr>
<td>sixtrack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>swim</td>
<td>ST</td>
<td>MK</td>
<td>ST</td>
<td>MK</td>
<td>ST</td>
<td>MK</td>
<td>ST</td>
<td>MK</td>
<td>MT</td>
<td>MT</td>
</tr>
<tr>
<td>twolf</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>ST</td>
</tr>
<tr>
<td>vortex</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
</tr>
<tr>
<td>vpr</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
</tr>
</tbody>
</table>

Note: B-S: Best-strategy, M-S: Multi-strategy
SQ: Sequential, ST: Strided, MK: Markov, MT: MLDT

The last bar within each set of tests represents the miss rate reduction with all four prefetching algorithms working concurrently. The experimental results show that this case has achieved about the same reduction as multi-strategy adaptive prefetching, while
sometimes better than adaptive strategies. However, as we will see from the reported IPC results, the best miss reduction does not translate to best overall performance because this strategy generates extensive replacements to the prefetch destination. In addition, the all-prefetching strategy consumes considerable more resource, like power, than smart adaptive strategies because the latter can identify the optimal ones and shut off low-efficiency prefathers.

Figure 4.16 demonstrates the overall performance improvement in terms of IPC (Instructions Per Cycle) reported by SimpleScalar simulator. The results shown in the figure include twenty-one benchmarks under eight cases, similarly as discussed in the previous sub-section. The IPC improvement also confirms that different prefetching algorithms benefit distinct benchmarks with different patterns. Any specific algorithm did not achieve the best IPC speedup for all benchmarks. Instead, these four supported prefetching algorithms have large variations in terms of the performance gain measured in IPC.

As shown from the reported results, the adaptive hardware prefetching does have the capability to distinguish well-performing algorithms from others and adapt to these selected algorithms to achieve an overall optimal performance gain. For instance, the best-strategy adaptation has successfully identified strided prefetching suitable for ammp, applu, crafty, gcc, gzip, lucas, mcf, parser, twolf and vpr, while sequential prefetching suitable for equake, gap, mesa, mgrid and swim. Both Markov and MLDT prefetching were also identified as optimal strategies at some cases, like Markov for the benchmark bzip2, eon and vortex, and MLDT for the benchmark art, galgel and sixtrack. Notice that a better cache miss rate reduction does not necessarily result in a better IPC
improvement. Take the applu benchmark as an example. The strided prefetching reduced less misses than MLDT did, but it produced better IPC improvement. This is because that MLDT involves more prediction overhead.

Figure 4.16. IPC Improvement of SPEC-CPU2000 Benchmarks with Data Prefetching

It is interesting to notice that multi-strategy adaptation usually generates better performance improvement than best-strategy does. This is because that the multi-strategy adaptation is able to recognize multiple well-performing algorithms, and can benefit and complement each other while avoiding low-effective algorithms. Adopting all supported prefetching algorithms does not produce the best performance speedup. This is because that adopting a low-accurate, low-coverage or high-pollution algorithm can even
substantially worsen the performance. This fact has also been confirmed from most cases in the experiments. In a summary, as verified from the simulation testing, the proposed best-strategy and multi-strategy adaptive prefetching are able to dynamically choose proper algorithms for different applications and to achieve an overall optimal improvement.

4.6  **Application and Impact**

As a summary of the discussion in this chapter, we present a specialized hardware approach to improving cache-memory stage data-access efficiency by exploring comprehensive and adaptive data prefetching. We first introduce a prefetching dedicated hardware structure, Data-Access History Cache, and study the methodology of supporting a variety of data prefetching mechanisms. We then present an adaptive prefetcher design that is able to dynamically identify runtime access patterns and adapt to suitable data prefetching strategies. The simulation experiment with an enhanced SimpleScalar has confirmed considerable data-access latency reduction and overall system performance improvement with various benchmarks. As the simulation verifies, the specialized hardware data prefetcher is beneficial for a variety of applications, including data compression (e.g. gzip and bzip2), compilation (e.g. gcc), engineering simulation (e.g. vpr), combinatorial optimization (e.g. mcf), multi-grid solver (e.g. mgrid), water modeling (e.g. swim), computational fluid dynamics (e.g. galgel), image recognition (e.g. art), computational chemistry (e.g. ammp), etc.

The impact of this work has several folds. Firstly, our study has verified that application features dominate data referencing patterns, and a large variety of
applications desire distinct prefetching strategies. Secondly, considering diverse application features, a dynamic prefetching strategy selection based on runtime access pattern is necessary to provide a cost-effective and energy-saving prefetcher design instead of always supporting all prefetching strategies. This study has presented such an adaptive prefetcher and provided guideline in hardware implementation. Lastly, rapid semiconductor technology evolution results in large amount of transistors available on a chip. This study suggests an alternative to the processor design approach by utilizing these available transistors to build a dedicated prefetcher for improving data-access efficiency. This specialized prefetcher is essentially a data access accelerator and targets to improve sustained performance considerably instead of the peak performance of a single chip. We believe this suggestion and design alternative will provide a new direction for the vendors in manufacturing future high-performance processor chips.
CHAPTER 5
IMPROVING MEMORY-DISK STAGE DATA-ACCESS EFFICIENCY

This chapter discusses the memory-disk stage latency reduction solution of the Hybrid Adaptive Prefetching architecture in detail. We study specialized software approach of data prefetching for improving data-access efficiency, and discuss related issues and solutions. These studies and results have been published in refereed conference proceedings in [11][19][20][21].

To reduce memory-disk stage data-access latency by exploiting the benefits of data prefetching, we introduce both pre-execution analysis based and post-execution analysis based prefetching. We present the system design, technical challenges and our solutions to these challenges in detail in the following sections. The proposed pre-execution and post-execution based prefetching ideas themselves are general and applicable to both sequential POSIX I/O and parallel I/O, but we investigate these approaches specifically for parallel I/O because parallel applications are of more interest in terms of high performance and high throughput I/O. We have carried out prototype implementation of these approaches for parallel applications (we assume they are MPI applications) within MPI-IO middleware layer, and performed extensive testing with various benchmarks and applications. The experimental testing confirms these software approaches of data prefetching are capable of substantially improving data-access efficiency.

5.1 MPI, MPI-IO and Parallel I/O
In this section, we briefly review the background of MPI (Message Passing Interface), MPI-IO and parallel I/O for the further discussion of our software data prefetching approach to improving data-access efficiency.

5.1.1 MPI. MPI (Message Passing Interface) is the de facto standard of parallel programming model [106][107]. It defines a specification of API (Application Programming Interface) that allows multiple concurrent processes communicate with each other to accomplish a task. It is widely used for programming scientific computing, data mining, information retrieval applications, etc.

MPI is usually implemented as a library or a language extension. Common MPI implementations include MPICH from Argonne National Laboratory, LAM/MPI from Indiana University, FT-MPI from University of Tennessee, LA-MPI from Los Alamos National Laboratory and Open MPI from a joint effort of multiple institutes, national laboratories and companies. Most MPI implementations consist of a specific set of routines (i.e., an API) callable from Fortran, C, or C++ and from any language capable of interfacing with such routine libraries. The programs that users write in Fortran, C and C++ are compiled with ordinary compilers and linked with the MPI library. Although other parallel programming models, such as PVM (Parallel Virtual Machine), exist, MPI remain the most popular and dominant parallel programming model in high-performance/high-end computing community. Therefore, we assume the parallel application under study is an MPI application in this dissertation.

5.1.2 MPI-IO. MPI-IO is a subset of MPI specification [59][107]. It defines I/O access interface for parallel applications. It is part of the MPI middleware, sitting between
applications and underlying parallel file systems, and hides the details of underlying systems.

MPI-IO began as a research project at IBM in 1994 and many researchers from other sites contributed to it subsequently. The primary motivation for MPI-IO specification came from the observation that parallel I/O optimizations require two basic abstractions: the ability to define a set of processes (MPI communicators), and the ability to define complex data access patterns (MPI datatypes). MPI interface has already equipped the ability for these two abstractions. Therefore, built upon communicators and datatypes from MPI, the MPI-IO designers created an interface that supports many parallel I/O operations and optimizations.

![Figure 5.1. ROMIO: A High-Performance, Portable MPI-IO Implementation](image)

The implementation of MPI-IO usually uses many features of MPI, thus, any program that uses MPI-IO must have an MPI library available. The most popular MPI-IO implementation is called ROMIO [93] and is developed by Argonne National Laboratory. Figure 5.1 illustrates a high-level view of ROMIO implementation. As can be seen from the figure, ROMIO hides the details of underlying files systems and provides a consistent parallel I/O API, MPI-IO, to application developers. ROMIO also performs various
optimizations for noncontiguous access patterns, which are common in parallel applications.

In this dissertation, we adopt MPI-IO and ROMIO as the foundation of our proposed data-access latency reduction approach. The prototype implementation integrates our approach into ROMIO, and thus enhances the data-access performance for all parallel applications that use MPI-IO access interface.

5.1.3 Parallel I/O. Parallel I/O is a general term and primarily refers to optimization techniques that improve data-access performance at device and file system level. It sometimes also refers to techniques at middleware (such as MPI-IO) or application level (such as Hierarchical Data Format library).

The primary parallel I/O solution at device level includes JBOD and RAID. JBOD stands for Just a Bunch Of Disks. This approach combines multiple physical drives together to form a much larger logical drive. The drive controller can stripe data and store into each individual disk drive. RAID stands for Redundant Array of Inexpensive (Independent) Disks [59][71]. It is a more complicated management of a bunch of disks to increase data throughput and reliability. The central idea in RAID is to replicate data over several disks so that the throughput can be improved and no data will be lost if one disk fails. The original RAID paper described five strategies, called RAID levels, ranging from RAID-1 to RAID-5. Each level has a different mechanism of replicating data and thus different performance characteristics. Several new RAID models have been developed since the original paper was published, such as RAID-6, RAID-10 and RAID-53. In addition, disk striping without redundant storage is usually referred as RAID-0, which is exactly same with JBOD.
The other major direction of parallel I/O solution is providing high-performance parallel file systems. A parallel file system manages multiple distributed disks and provides a single system image for notions of files and paths. It also manages the metadata (the information about files, such as creation time, access privilege, file size, etc.) efficiently, usually in the form of a distributed management, for extensive concurrent accesses. In addition, a parallel file system has to enforce correct consistency model when multiple processes access the same file simultaneously. The primary distinction between a parallel file system and a distributed file system (such as Network File System, NFS, from Sun Microsystem) is that the distributed file system generally treats concurrent access as an unusual event, and thus is not focused on consistency management. In addition, the distributed file system primarily provides a shared storage and single system image notion for users, while the parallel file system provides high throughput and performance in addition to that.

Common parallel file systems include Parallel Virtual File System (PVFS) from Argonne National Laboratory and Clemson University, General Parallel File System (GPFS) from IBM, Lustre parallel file system from Sun Microsystem, Panasas File System (PanFS) from Panasas, etc. In this dissertation, we employ both PVFS and the Network File System (NFS) as our testing bed. We test the performance improvement of our approach on these file systems with different benchmarks and applications.

5.2 Pre-execution Based I/O Prefetching

As discussed in the previous section, current parallel I/O solutions can greatly improve I/O throughput for large and well-formed I/O. However, a critical limitation of these solutions is that they are not capable of reducing the I/O access latency effectively,
especially in the case of many isolated, irregular or small accesses. In the meantime, numerous prior studies have revealed that many data-intensive applications often exhibit non-contiguous and isolated accesses, as well as complex and irregular accesses [45][52][74][75][79][96]. Many requests are also periodic and small accesses. These diverse application features should be well considered in order to deal with data-access bottleneck. In the following section, we will present a novel technique, called pre-execution I/O prefetching, to address the limitation of existing parallel I/O solutions and improve the data-access efficiency. We first present the design rationale of the pre-execution prefetching, then we introduce pre-execution prefetching construction methodology and automating the construction.

5.2.1 Pre-execution Prefetching Methodology. The essential idea of the proposed approach is to overlap the computation and I/O accesses via speculative prefetching. This approach speculatively pre-executes a fragment of code on each process to identify future I/O references, called I/O hints. The speculative execution deals only with I/O related operations and the computations that are critical to the I/O access address. Since we assume that the computational capability is enormous and I/O is the performance bottleneck, the computing power spent on pre-executing I/O related operations is negligible and the overall performance is improved. An underlying library collects and processes I/O hints and proactively fetches data into a buffer cache near client nodes. The cached data can be retrieved by the MPI-IO library to serve requests from regular computation processes instead of stalling the process and fetching data from the low-level storage. Therefore, the process stall time on I/O accesses can be effectively masked. Figure 5.2 illustrates an ideal case, where the latency of three periodic reads (R2, R3 and
R4) is masked completely via pre-execution prefetching and the total execution time is reduced noticeably.

Figure 5.2. Hiding Latency with Pre-execution Prefetching

Figure 5.3 illustrates a high-level view of the pre-execution I/O prefetching. The pre-execution is conducted via a helper thread or prefetching thread/pre-execution thread for each parallel process. Each original process forms a main thread or computation thread. The prefetching thread is composed of only I/O related operations of the original process and is attached to each main thread to prefetch data in advance. The original parallel application source code is transformed either with the programmer’s intervention or with a source-to-source pre-compiler to obtain the prefetching thread. The prefetching thread shares certain resources with the main thread, such as MPI file handles and process rank. It runs ahead of the main thread because it only contains the essential computation for data address calculation, and thus is able to produce effective prefetches for the main thread. The prefetching thread is supported by an underlying prefetch function call library that provides the prefetch counterparts of normal I/O function calls. It collects hints, generates prefetch requests, and schedules prefetches. The prefetch
library can also track function-call identifiers to synchronize the prefetching thread and the computation thread I/O calls, and to force the prefetching thread to run properly. The cache buffer resides on the client side (in contrast, the source data resides on the server side) and serves as the prefetch destination. A caching library manages the actual fetching of data to the buffer cache. The regular MPI-IO library is enhanced to take advantages of the prefetched data residing in the buffer cache. As the figure demonstrates, the logical flow is that the prefetching thread communicates with the prefetching library, generates hints, and fetches data into buffer cache through the caching library. The computation thread is thus able to access the cached data via the enhanced MPI-IO library and mask process stall time. The caching library and the regular MPI-IO library talk to the underlying file system and perform actual data transfer.

Figure 5.3. Pre-execution Parallel I/O Prefetching
The pre-execution based prefetching approach has many technical challenges that include generating accurate I/O hints, guaranteeing expected program behavior, constructing the pre-execution thread efficiently, synchronizing the pre-execution thread with the main thread as necessary, and performing the prefetching and caching with the library support. We address these challenges in the following subsections. The prefetching and caching library support is discussed in Section 5.4.

5.2.2 Pre-execution Prefetching Thread Construction. In this section, we analyze the pre-execution thread construction problem in detail and present design considerations of various aspects. An efficient method of extracting the I/O related code to construct the pre-execution thread is discussed in the following section. This section addresses the challenges in generating accurate I/O hints, preserving correct program behavior, and handling necessary synchronizations.

5.2.2.1 Design Considerations. The pre-execution thread runs at the same time with the main thread, but usually ahead of the main thread to trigger I/O operations earlier and warms up the underlying buffer cache with prefetched data to reduce the access latency for the main thread. This approach essentially tries to overlap the expensive I/O access with the computation in the main thread as much as possible.

The main design considerations include two aspects: correctness and effectiveness. Correctness means that the prefetching must not compromise the correct behavior of the main computation thread. Since the prefetching thread shares certain resources with the main thread, such as memory address space, process identification, and opened file handles, an inconsiderate design of the pre-execution prefetching might result in unexpected results. We discuss in detail our design to guarantee that the
prefetching does not disturb the main thread with regards to memory, communication, and I/O behavior. The design provides a systematic way to perform pre-execution prefetching effectively and generate accurate I/O hints.

5.2.2.2 Dealing with Memory Behavior. A straightforward design to guarantee the correct behavior of the main thread is to perform store removal within the pre-execution thread. After removing the potential writes to shared variables between the main thread and the prefetching thread, we prevent the possibility that the prefetching thread can change the memory state of the main thread. Note that store removal does not need to apply to automatic variables (stack variables) because these variables are on the stack and are private to each thread. The limitation of this approach, however, is that it affects the accuracy of the pre-execution thread. This inaccurate pre-execution behavior will not affect the correctness of the program though. It merely decreases the accuracy of the prefetching, and thus affects the effectiveness.

We propose a code cloning or variable renaming technique to increase the pre-execution accuracy while guaranteeing the correctness in the meantime. This technique creates another separate variable (for the purpose of speculative prefetching) whenever a variable is potentially shared among the main thread and prefetching thread. It can guarantee that the main thread’s memory state is untouched while allowing the prefetching thread to run accurately. We perform a source-level code cloning to realize the variable renaming technique. The variable renaming, however, is not free of cost. It consumes additional memory at runtime for the prefetching thread even though it is safe to share the memory region with the main thread. We assume that memory space is not a factor in limiting performance considering the trend of much larger memory at low cost.
An advanced technique, *copy-on-write*, can be used to reduce the memory overhead. This technique tries to share the memory space as much as possible and make extra copies only when necessary. The copy-on-write technique is widely used in efficiently constructing new processes (such as with the fork() system call) by the operating-system kernel.

**5.2.2.3 Dealing with Communication Behavior.** In general, I/O related operations that constitute the pre-execution thread of a specific process do not involve communication with other processes. If they do involve communication, our design will preserve the correct communication behavior for the main thread. The communication is in essence an exchange of memory state among multiple processes; therefore, we can follow the memory-behavior handling to deal with communication. It is possible to make the communication among prefetching threads speculative (ignore certain sends and receives) to accelerate the pre-execution. The drawback, however, is similar to the store removal approach in the memory behavior handling, and can result in inaccurate prefetching results. The approach we choose allows prefetching threads to communicate with each other as normal, and uses special message tags to isolate this communication from the communication in the main thread. We believe that a small communication overhead is justified for obtaining more accurate and effective pre-execution results. This approach can be extended to handle collective communication as well.

**5.2.2.4 Dealing with I/O Behavior.** To simplify the discussion and focus on the methodology itself, we only deal with MPI-IO operations with individual file pointers or with explicit offsets. The methodology, however, is general and extensible for collective operations and operations with shared file pointers.
**MPI-IO Thread-safety.** The underlying prefetching library provides prefetch counterparts of I/O functions to support the proposed approach. MPI-IO function calls (reads/writes) can be roughly classified into two categories, one with hidden file pointer as the file offset and one with explicit file offset. The one with explicit offset is thread-safe because these functions use a specified offset to access the file and do not rely on a hidden and shared file pointer among multiple threads. The proposed approach employs a separate thread to run ahead and prefetch data, and thus it involves the thread-safety consideration. To solve this issue, we introduce one more hidden file offset pointer, named *prefetch file pointer*, within the opaque MPI file handle object to track the prefetching thread file offset. The prefetch file pointer is generally different from the normal file pointer, and does not match with the system-level file pointer position usually maintained in a MPI-IO library implementation. Note that the prefetch version of the thread-safe functions does not use the prefetch file pointer and they guarantee the thread-safety naturally.

**Dependence Considerations.** The proposed pre-execution I/O prefetching runs a fragment of code ahead of the main thread to page in data into the buffer cache in advance. It is possible that the pre-executed I/O operations rely on previous reads/writes from the main thread. If we do not resolve this issue carefully, we might break the sequential semantics guaranteed by MPI-IO. This subsection discusses the dependence considerations within a single process, and Section 3.4.4 discusses preserving consistency semantics among multiple processes.

Concurrent reads do not interfere with each other, but writes can potentially conflict with other reads/writes. Therefore, to preserve the correct dependence and
consistency among I/O calls and not disturb the main thread I/O behavior, the simplest solution is converting write operations as synchronization points when generating the pre-execution thread. To preserve data integrity, only the main thread performs writes and not the pre-execution thread. This approach is analogous to partitioning a program into many segments delimited by write operations. The pre-execution prefetching is available and safe within each segment, but not across segments. Obviously, the downside of this approach is that it limits the degree of prefetching to explore the computation and I/O concurrency because not all writes need to be immediately visible to the process. Therefore, it is possible to speculatively perform prefetching for the future reads if they are not conflicting with prior writes.

We propose a delayed synchronization approach to tackle this issue. The rationale of the approach comes from the fact that only the RAW (Read After Write) dependency is a true dependency, and only its corresponding writes need to be visible to the reads. This approach allows the prefetching thread to record the write byte ranges when encountering a write and to continue to run ahead without synchronizing with the main thread. This byte range is termed dirty range and indicates the region of data that is supposed to be written with new data from the main thread. However, as long as the future reads do not need this data region, it is safe to allow the prefetching thread to run ahead and page in required data. When the prefetching thread encounters a read, it always performs a boundary check with the current dirty range. If the read region falls into or overlaps with the dirty range, we perform a delayed synchronization to wait for the data from the main thread to be written into the disk. The synchronization is implemented by forcing the prefetching thread to wait until the specified function is performed from the
main thread. A *dependency analysis table* is maintained to map the byte range and the function identifier of the writes that contribute the dirty range. This mapping is used to look up the function call that needs to be synchronized for a certain dirty range. The dirty ranges can be combined or split as the I/O reads, writes and synchronizations go on.

**Prefetch Conversions.** Prefetch conversions are required for the proposed pre-execution prefetching, either with the programmer’s intervention or with an automatic tool such as the pre-compiler discussed in the next section. The general rules are to convert reads, writes, and seeks to prefetch counterparts as supported by the prefetching library (reads/writes are handled with the previous dependence analysis, and seeks simply change the prefetch file pointer), and add in necessary synchronization handling. This handling includes converting file open/close operations and MPI_File_sync() and file attribute modification operations (such as setting file size or deleting a file) as synchronization points. The MPI file handles are transformed to global variables to make them shareable between the main thread and prefetching thread (different from the memory behavior handling). The MPI initialization is converted to MPI_Init_thread() for thread support if that is not the case.

**Preserving MPI-IO Consistency Semantics.** Pre-execution prefetching also preserves MPI-IO consistency semantics among multiple processes. As the MPI-2 standard [107] indicates, MPI-IO provides weak consistency by default, and for stronger semantics, users need to take explicit actions, such as setting the atomic mode, closing and reopening the file, or using MPI_File_sync() and MPI_Barrier() to prevent two concurrent overlapping writes. In all these cases, the MPI-IO consistency semantics are preserved with the prefetching methodology because the required locking for the
atomicity mode is performed for the prefetching thread, MPI_BARRIER() semantic is also preserved, and the file closing and opening, and MPI_File_sync() are turned into synchronization points as required to preserve the consistency semantics.

5.2.3 Automating Pre-execution Thread Construction with Program Slicing. It is possible to follow the construction methodology and utilize the caching library, prefetching library and enhanced MPI-IO library to construct the prefetching thread manually to benefit from pre-execution prefetching. The manual construction, however, is tedious and error-prone. In this section, we present the design of a source-to-source pre-compiler to address the challenges of constructing the pre-execution thread automatically and efficiently.

5.2.3.1 Mapping Pre-execution Thread Construction to Program Slicing. We use the program slicing technique [95] to automatically construct a pre-execution prefetching thread. The program slicing technique was originally proposed for debugging and studying program behavior. It is a family of program decomposition techniques based on extracting statements relevant to computation within a program. Program slicing relies on Program Dependence Graph (PDG) analysis [32], a combination of control dependence and data dependence analysis of programs. It takes the source code as input and computes a slice (subset of the original program) based on the slice criteria, the variables or statements of interest. The construction of the pre-execution thread can be mapped to the program slicing problem because the pre-execution thread is essentially a subset of the original program, where I/O variables and statements are of interest. If we slice the original program with all I/O function calls and their arguments as slice criteria, we
obtain all I/O related operations, that is, I/O operations and the critical computations that might affect those I/O operations.

5.2.3.2 Program Slicing with Unravel. We employ a well-implemented open-source program slicing toolkit, Unravel [54][114], for our prototype pre-compiler development. To compute program slices, Unravel parses the source program and represents it as a flow graph of nodes annotated with lists of variables and based edges indicating control flow. For each node, the annotation maintains a defined variable set, a referenced variable set, and an active variable set – the set of variables that the slicing criteria depend on just before program execution reaches that node. The slicing computation starts with all the active sets initialized to be empty, except that the active set for the slicing criterion statement is initialized to the criterion variable. The slice is computed by propagating the active sets across the entire flow graph until no changes occur to the active sets. The computation of the active set for an arbitrary node is controlled by comparing variables defined at that node with the active sets of immediate successor nodes by slicing rules [43][54].

Figure 5.4. Unravel Structure Overview

Figure 5.4 illustrates the structure of the Unravel toolkit. Unravel is composed of three main components: a source code analysis component, a link component, and a
slicing component. Source files are transformed to a representation independent of source language called *language independent format* (LIF) by the analyzer. The analyzer is similar to a compiler with a scanner to break the source code into tokens that are recognized by a parser, but instead of generating object code, it produces LIF code. The LIF files for a given program are bound together by the linker into a single link file. The link file is fed into the slicer, and the slicer outputs sliced code for different slicing criteria.

### 5.2.3.3 Slicing for Pre-execution I/O Prefetching

The overview structure of our prototype pre-execution code generation pre-compiler is shown in Figure 5.5. The pre-compiler is built upon Unravel and uses the Unravel analyzer and slicer components to compute slices of I/O related codes based on each individual I/O function call statements. The complete pre-execution code is built via merging these slices and performing necessary prefetch conversions with the LIF files and link file support. The output of the pre-compiler is an optimized code with pre-execution prefetching enabled, and the optimized code uses the underlying library support to accomplish the prefetching work.

The basic slicing algorithm for pre-execution is shown in Figure 5.6, where $S_{m, v}$ denotes the slice computed for the slice criterion, variable $v$ at statement $m$. The algorithm considers all predecessor statements $n$ in the PDG. If statement $n$ does not assign a value to the variable $v$, it is omitted from the slice, and we recursively evaluate $S_{n, v}$. Otherwise, if statement $n$ assigns a value to the variable $v$, it is included in the slice for criterion $<m, v>$, and we recursively evaluate the program slice for all referenced variables $x$ used to compute $v$ at statement $n$ (the second term), as well as the program slice for all referenced variables $y$ at all statements $k$ that *control* the execution.
of statement \( n \), denoted by \( \text{req}(n) \) set (the third term). The second term within the algorithm deals with the data dependence among statements, while the third term deals with the control dependence and includes necessary statements into the slice.

![Figure 5.5. Pre-execution Code Generation](image)

In addition to the basic slicing algorithm, we also use Unravel features to support advanced analyses, such as arrays and structures analysis, pointer analysis and procedure analysis to provide more fine-grain dependence information and improve the quality of the slice [54]. For instance, Unravel keeps track of pointer assignments and references, and analyzes each level of indirection when generating slices. It also supports inter-procedural analysis to construct slices across procedure boundaries. The basic algorithm and these advanced features are sufficient for our purpose in building the pre-execution thread construction pre-compiler. Some existing studies of data flow analysis specifically for MPI programs [83] also provide useful experiences for our study.
The prefetching thread is able to run ahead of the main thread and is effective in fetching data in advance to overlap the computation and I/O accesses for the following reasons. As the previous discussion illustrates, the code not relevant with I/O operations is sliced away, which makes the prefetching thread contain only the essential I/O operations and the code on the critical path to these operations. Therefore, the prefetching I/O thread is not involved in enormous computations and runs much faster than the main thread. Secondly, the prefetch version of I/O calls are used within the pre-execution thread to replace normal I/O calls. These prefetch calls avoid the cost of making an extra memory copy to the user buffer. They can also be implemented with non-blocking accesses to accelerate the prefetching thread. Other techniques, such as delayed synchronization, also contribute to the fast execution of the prefetching thread, and allow the prefetching thread to speculate as far as allowed and generate accurate I/O hints. When the prefetching thread happens to lag behind the main thread, the underlying library implementation makes it able to detect that to skip prefetch calls and catch up with the main thread.

### 5.3 Post-execution Based I/O Prefetching

In addition to the pre-execution I/O prefetching, we have also proposed a prefetching method with a combination of post-analysis and runtime analysis of I/O accesses [11]. The idea behind this method is to detect the pattern of I/O accesses of an
application, store the pattern information as a *signature* representation, and use that signature in the future runs of the application. To develop the signature notation of I/O accesses, we have classified I/O access patterns based on a study of a collection of widely used parallel benchmarks [11]. We collect information of an application’s MPI-IO calls and analyze these traces to generate a representation of an I/O access pattern, called I/O signature [11]. At runtime, a prefetching thread runs alongside the main computing thread during I/O operations to predict data requirements of the main thread and to bring that data into client-side cache that is closer to the application. In the following subsections, we briefly introduce the I/O access pattern classification, I/O signature notation and signature based post-execution prefetching method. The detailed study can be found in [11].

### 5.3.1 Access Pattern Classification

Based on our study of various I/O benchmarks that represent real parallel applications, we have introduced a five-dimensional classification for I/O access patterns [11]. The five dimensions are *spatiality*, *request size*, *repetitive behavior*, *temporal intervals*, and *type of I/O operation*. The sequence of file locations accessed represents the *spatial pattern* of an application. They can be contiguous or non-contiguous or a combination of both. Non-contiguous accesses refer to gaps (or strides between successive file offsets) in accessing a file. These gaps can be of fixed size or variable size. Variable size gaps can follow a pattern of two or more dimensions (2-d or k-d). Another possible pattern is one with decreasing (negative) strides. Some I/O accesses have no regular pattern, where the strides are random.

Applications exhibit *repetitive behavior* when a loop or a function with loops issues I/O requests. We classify I/O access patterns either with repetitive behavior or
without (i.e., pattern occurs only once). When I/O access patterns are repetitive, caching and prefetching can effectively mask their access latency. By capturing repetitive behavior, cached data can be kept longer or accesses can be reordered in a way that the fetched data is completely used before replacing it. Prefetching can utilize this repetitive behavior by storing previous pattern information and reuse that information to calculate future I/O access offsets without searching for the same pattern multiple times.

Request sizes can be small, medium, or large. The sizes of requests can be either fixed or varying. We characterize a request as a small request when it is only a fraction of a page size, and as a large request when it is multiple times larger than a page size. Because of the high I/O latency, small I/O requests commonly cause performance bottlenecks if disks must be accessed multiple times for a small number of bytes. If possible, multiple small accesses can be combined into a larger contiguous request to reduce the number of disk seeks. Temporal patterns capture regularity in I/O bursts of an application. They can occur either periodically (at fixed intervals) or irregularly. Capturing temporal regularity can be used in prefetching strategies to initiate prefetch requests in time, so that prefetched data reaches its destination cache neither too early nor too late. The type of I/O operation is the last criteria for pattern classification. We classify the operations as read, write, or read/write.

Classifying I/O accesses into a set of patterns gives us an opportunity to tune performance. Prefetching strategies can utilize these patterns to predict future accesses. I/O accesses can be reordered to improve cache reuse by having the information of access patterns. For instance, in out-of-core applications, data processing operations can be performed on all the data that has been fetched into memory before it is swapped to disk.
5.3.2 I/O Access Signature Notation. We have developed a set of notations to describe I/O access patterns, which we call the I/O signature of access patterns for an application [11]. The I/O signature can be given in two forms; the first describing the sequence of I/O accesses in a pattern and the second identifying I/O patterns. We call the description of a sequence of I/O accesses in a pattern a trace signature, and the abstraction of a pattern a pattern signature. Using the five dimensions mentioned above, trace signature takes the form as follows:

\{I/O \text{ operation}, \text{ initial position}, \text{ dimension}, ([\{\text{offset pattern}\}, \{\text{request size pattern}\}, \{\text{pattern of number of repetitions}\}, \{\text{temporal pattern}\}], [...]), \# \text{ of repetitions}\}

It stores information of an I/O operation, starting offset, depth of a spatial pattern, temporal pattern, request sizes, and repetitive behavior. In some instances, offsets, request sizes, timing, and number of repetitions also contain a pattern. Random temporal patterns are not captured in the trace signature, as the usage of randomness is limited. While a trace signature provides a way to reconstruct the sequence of I/O accesses, a pattern signature provides an abstract description that explains the nature of a pattern. A pattern signature takes the following form:

\{I/O \text{ operation}, <\text{Spatial pattern, Dimension}>, <\text{Repetitive behavior}>, <\text{Request size}>, <\text{Temporal Intervals}>\}

A significant advantage of an I/O signature, which has been the motivation behind this method, is its usage in predicting future I/O accesses and prefetching that data. Post-execution analysis of an application’s I/O accesses can be stored as a signature, which can be used by a prefetching strategy as hints for generating prefetching requests. Let us take an example where a process is accessing I/O in a strided pattern. Assume a fixed
stride of 8192 bytes and each I/O read request is of 2048 bytes. The trace signature of that pattern is \{READ, initial position, 1, ([8192, 4096, 100]), 1\}. Prefetching strategies can use this trace signature to calculate future file offsets from initial position, i.e. (0 * 8192, 1 * 8192, 2 * 8192, … , 99 * 8192).

A limitation of the signature is its usage for representing random accesses. While usage of trace signature for regular patterns reduces the size of a trace file, there is no such benefit for representing random I/O accesses. For random accesses, the storage space required for a trace signature representation and a trace file are of the same order. In such cases, instead of having a large trace signature, providing a pattern signature can itself express the information that these I/O accesses are random. This can guide prefetching strategies to avoid trying to predict future offsets, as it is difficult to predict random I/O accesses from the history of accesses.

5.3.3 I/O Signature Based Prefetching. I/O signature-based prefetching is a two-step process [11]. In the first step, traces of a running application are collected and analyzed to detect any patterns among them. The detected patterns are stored as an I/O signature. An I/O signature contains information regarding the strides between successive I/O accesses (spatial pattern), how many times the pattern is repeated, its temporal pattern, the size of requested data, etc. The second step involves prefetching at runtime, where signatures of an application are read, verified, and used to prefetch data when a stable pattern is found.

In order to collect application traces, we have developed a tracing tool that traces all MPI-IO read and write related calls [11]. To capture MPI-IO calls, we use the profiling interface of MPI [106], which provides a convenient way for us to insert our code in an implementation-independent fashion. In MPI implementations, every function
is available under two names, MPI_ and PMPI_. User programs use the MPI_ version of the function, for example, MPI_File_read. We intercept the user’s call to MPI_File_read by implementing our own MPI_File_read function in which we retrieve information required for tracing and then call PMPI_File_read to do actual file read. As a result, application programs do not need to be recompiled; they need only to be re-linked with our version of the MPI functions appearing before the MPI library in the link command line. Traces are stored in a text file, with a header describing the records of each trace. We have also developed an analysis tool that reads through traces and gives the I/O signatures of an application as output. It has five pattern detectors for finding patterns among initial positions, offsets, request sizes, temporality, and repetitions. For successive trace records, we search for various patterns including fixed strided, \( k\)-d \((k = 2 \text{ and } 3)\), and negative strided patterns. While searching for patterns, we keep three states: initial state, learning state, and stable state. The search is in the initial state at the beginning and goes into learning state when a pattern is first detected. If the same pattern is detected in the following traces, the search state becomes stable. Otherwise, it returns to the learning state and searches for next type of pattern. If no pattern is found, the search will be in initial state. Once a pattern is in stable state, it is written into a signature and the number of repetitions is updated in the signature. All the sub-signatures are combined to form trace signature and pattern signature.

For the second step, the prefetching thread reads the corresponding signature of an application at runtime \([11]\). In order to start prefetching, it is necessary to verify whether the signature is following the current I/O requests of the main thread. It is possible that a trace signature values depend on the rank of an MPI process and according
to the current I/O accesses of an MPI process, the signature has to be adjusted to start prefetching data efficiently. We implement this signature adjustment process using shared variables. The main thread uses these shared variables to communicate with the prefetching thread. These variables, which are protected with a POSIX mutex, include file handle of the file that is being read, file location (i.e. where in the file the I/O read is occurring), and request size of an I/O read. When the shared variables are available, the prefetching thread reads them and compares them with the current signature. Take stride prefetching as an example, if the stride and request size of the current signature are same as the stride and request size of the main thread I/O reads, respectively, then the prefetching thread assumes that the signature is in a stable state and issues prefetching requests. The initial file location in the trace signature is set from the current file location from shared variables. If either the strides or request sizes of the signature are different with current I/O read parameters (from shared variables), the prefetching thread stays in the learning state and retrieves current strides or request size values to adjust the signature. If both strides and request sizes are different from current I/O read values, prefetching thread does not issue any prefetching requests assuming that prefetching thread does not have the correct signature.

Figure 5.7 illustrates the prefetching thread operations of one client node. A prefetching thread starts for each MPI process when the first file is opened and ends when the last file is closed. The prefetching thread initiates a user level prefetching cache and reads I/O signatures from a file (I/O signature DB). The parameters of the I/O signatures are adjusted dynamically by observing the I/O accesses of the MPI process to which the prefetching thread is attached. The main thread communicates I/O access
information to its prefetching thread using shared variables. When the prefetching thread finds a stable pattern, it starts prefetching data into the prefetch cache, which the main thread looks up before sending a request to its underlying file system. If data is found in the prefetch cache, the main thread accesses that data. Otherwise, its normal operation of sending an I/O request to file system is performed.

Figure 5.7. Prefetching Thread Operations from a Single MPI Process View

5.4 I/O Prefetching and Caching Library Support

This section discusses the design and algorithm of the underlying library support for the pre-execution and post-execution analysis based parallel I/O prefetching strategy, as well as the prototype implementation within ROMIO [111] and MPICH2 [108].

5.4.1 MPI-IO Prefetch Cache Library. To implement I/O prefetching, a cache closer to the computing node is needed. Several research projects have been working on MPI-IO caching libraries. Ma et al. proposed active buffering [55][56] and Liao et al. proposed collective caching [47][48]. Instead of reinventing a brand new caching library, we
choose the collective caching code [47][48] and customize for our purpose to build a client-side prefetch cache.

This client-side prefetch cache is implemented within ROMIO [111]. It maintains a global buffer cache among multiple processes at the client side. Figure 5.8 demonstrates the high-level view of the collective prefetch cache system. Each client contributes part of its memory to construct the global cache pool, and the high-speed interconnect network enables the rapid transfer of cached data among clients. A specialized cache-coherency protocol that allows at most one copy of data block is cached among all processes is used to maintain consistency among cache copies in the cache pool. We have disabled write caching of the original collective caching code and enabled read caching only. In addition, we utilize prediction results to direct caching policy. For instance, if the speculated future I/O references are already cached, these data blocks are given a higher priority to stay in the cache buffer instead of being replaced.

![Figure 5.8. Collective Client-Side Prefetch Cache Overview](image)

5.4.2 MPI-IO Prefetching Library. The prefetching library provides the implementation of prefetch counterparts of MPI-IO read/write function calls. Figure 5.9 shows the general algorithm of the prefetching library design and implementation.
The syntax and semantic of the prefetching reads are quite similar to the existing MPI-IO library design, but there are several key differences. First, the prefetching library calls do not have a user-specified buffer parameter. This distinction is straightforward because the data fetched by prefetching threads are stored in client-side buffer cache and are not supposed to return data to the user’s buffer. The second difference is that the prefetching library does not update the normal file pointers. It maintains a prefetch file pointer for the prefetching thread and always uses this file pointer to access data blocks. Another difference is that the prefetching reads perform a boundary check over the current dirty range and performs necessary delayed synchronization as discussed.

Figure 5.9. Algorithm of MPI-IO Prefetching Library Functions

```c
Algorithm mpiioplfnMPI-IO Prefetching Library Functions*
Input: MPI file handle, hints(offset, count and data type)
Output: none
{
    performs boundary check and necessary synchronizations
    if (pfid++ < fid)
        return;
    split prefetch requests into blocks
    for each requested block
    {
        if this block is already cached due to previous prefetches
            utilize requests to migrate remote cached copy to local nodes
        else
        {
            allocate buffer for this block
            if succeed
            {
                if I/O alignment is required
                    perform aligned read to allocated buffer
                else
                    perform direct disk read to allocated buffer
                add metadata to buffer caches list
                update prefetch file pointer
                /*do not return to users’ buffer*/
            }
            else
                return /*ignore prefetch requests*/
        }
    }
}
```
previously. The last difference is that, unlike ordinary MPI-IO library calls, prefetching function calls are silent: they do not return errors in general. The errors or exceptions caused by prefetching are generally discarded, and previous states are restored.

5.4.3 MPI-IO Regular Library. To benefit from prefetching, the regular MPI-IO library implementation is modified to be able to access the buffer cache for requested data in addition to satisfying the requests directly from the file system when the data is not found in the cache. The algorithm shown in Figure 5.10 describes the general modifications to the existing implementation. The algorithm divides the I/O request into blocks and checks whether each block already resides in the buffer cache or not. If the block is cached, we copy the block from buffer cache to user’s buffer via `memcpy()`. If the block does not appear in the buffer cache, we perform direct I/O reads from underlying file system, which is exactly the same as what the existing ROMIO does.

```c
Algorithm mpiiorlf /*MPI-IO Regular Library Functions*/
Input: MPI file handle, demand request(offset, data type, count)
Output: user’s buffer buf
{
    fid++
    split demand request into blocks
    for each block
    {
        if the block is already cached
        {
            hits++
            copy buffer cache to user specified buffer by using memcpy()
        } else
        {
            perform direct reads from file system
            /*do not cache requested data*/
        }
    }
}
```

Figure 5.10. Algorithm of MPI-IO Regular Library

5.5 Experimental Results and Performance Analysis
We have carried out experiments to verify the benefits of the pre-execution and post-execution analysis based prefetching for parallel I/O applications. This section discusses the experimental setup and experimental results.

5.5.1 Experimental Setup. Our experiments were conducted on a 17-node Dell PowerEdge Linux-based cluster. This cluster is composed of one Dell PowerEdge 2850 head node, with dual 2.8 GHz Xeon processors and 2 GB memory, and 16 Dell PowerEdge 1425 compute nodes with dual 3.4 GHz Xeon processors and 1 GB memory. The head node has two 73 GB U320 10K-RPM SCSI drives. Each compute node has a 40 GB 7.2K-RPM SATA hard drive. The experiments were tested on both NFS and PVFS file systems. PVFS [13][50] was configured with one metadata server node, the head node, and 8 I/O server nodes. All compute nodes were used as client nodes. The cache page size of the collective caching was set as 64 KB and the buffer cache size at each client was set as 32 MB.

5.5.2 Performance Results with Pre-execution Based Prefetching. In this subsection, we present the performance results with the pre-execution prefetching.

5.5.2.1 PBench Experimental Results. We have followed the PIO-Bench framework [78] and developed a parallel I/O benchmark, called PBench. PBench emulates a regular parallel application’s computation and I/O access behavior of many small and non-contiguous accesses. The computation is emulated with floating-point calculation, and the I/O accesses are emulated with accessing huge two-dimensional double-precision matrices. The difference between the PBench and PIO-Bench is that PBench characterizes both computation and I/O accesses, whereas PIO-Bench characterizes I/O behavior only. PIO-Bench is usually used for measuring the peak I/O performance with
different access patterns, while PBench is suitable for studying the sustained performance and the impact of different optimization techniques, MPI-IO implementations, and file systems.

We have conducted two sets of experiments with the PBench on NFS and PVFS respectively. In each set, we tested PBench with three settings: accessing a 4K by 4K, 8K by 8K, and 16K by 16K matrices. In each test, every I/O access is random, but the average request size is the row size. We flush the buffer cache before every run. The total accessed data was 128 MB, 512 MB, and 2 GB, respectively. The computation was configured as 1M iterations calculation of the accessed data.

Figure 5.11 shows the experimental results with 1, 2, 4, 8, and 16 processes on NFS and PVFS respectively. Each reported result is the average of at least three runs. In each figure, the first bar of every column represents the original execution time, and the second bar represents the execution time with pre-execution prefetching. The execution time was significantly reduced in almost all cases. The execution time reduction was up to 37.92%, and the average reduction was 29%, 33%, and 26% respectively in three cases when tested on NFS. When tested on PVFS, the execution time reduction was up to 32% and the average reduction was 23%, 24%, and 26%.

Figure 5.12 and Table 5.1 show another view of these results. It illustrates the aggregate sustained bandwidth when testing PBench with a 16K by 16K matrix on NFS and PVFS. The sustained bandwidth improved considerably with the pre-execution prefetching, and the bandwidth was much higher on PVFS than NFS. Since the proposed approach is on top of existing optimization techniques in MPI-IO or the file system, it
complements the existing approaches and can reduce I/O access latency further when combined with them.

Figure 5.11. PBench Results on NFS and PVFS
Figure 5.12. Aggregate Sustained Bandwidth on NFS and PVFS
Table 5.1. Aggregate Sustained Bandwidth on NFS and PVFS

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O</td>
<td>P</td>
<td>O</td>
<td>P</td>
<td>O</td>
</tr>
<tr>
<td>NFS</td>
<td>37.3</td>
<td>54.1</td>
<td>51.6</td>
<td>65.9</td>
<td>152.0</td>
</tr>
<tr>
<td>PVFS</td>
<td>103.5</td>
<td>147.6</td>
<td>208.2</td>
<td>326.2</td>
<td>402.9</td>
</tr>
</tbody>
</table>

Note: O: Original, P: Pre-execution prefetching (unit: MB/s)

5.5.2.2 Tile 2D-convolution Experimental Results. Tile 2D-convolution is a real application to conduct two-dimensional convolution on paired tile images. Each process is responsible for the 2D-convolution of two tiles. Each tile is composed of \( N \) elements in both \( X \) and \( Y \) dimension. The size of each element varies (e.g., 1 KB or 2 KB). The 2D-convolution uses Fast Fourier Transform (FFT) as its kernel. It first takes a 2D-FFT of each tile, then performs a point-wise multiplication of the intermediate results from the 2D-FFT, followed by an inverse 2D-FFT. A 2D-FFT can be performed by using a 1D-FFT routine and performing the 1D-FFT \( N \) times along rows followed by \( N \) times along columns. The procedure of 2D-convolution can be described as following:

\[
A = 2D-FFT(tile1)
\]

\[
B = 2D-FFT(tile2)
\]

\[
C = MM\_Point(A,B)
\]

\[
D = Inverse-2DFFT(C)
\]

Table 5.2 and Table 5.3 illustrate the experimental results of the tile 2D-convolution application on PVFS, and Figure 5.13 is the plotted figure.
Table 5.2. Aggregate Sustained Bandwidth of Two 5 by 5 Tiles 2D-Convolution on PVFS

<table>
<thead>
<tr>
<th></th>
<th>100×100</th>
<th>200×200</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1K×1K</td>
<td>2K×2K</td>
</tr>
<tr>
<td>O</td>
<td>56.05</td>
<td>102</td>
</tr>
<tr>
<td>P</td>
<td>67.25</td>
<td>123</td>
</tr>
</tbody>
</table>

Note: O: Original, P: Pre-execution prefetching (unit: MB/s)

Table 5.3. Aggregate Sustained Bandwidth of Two 10 by 10 Tiles 2D-Convolution on PVFS

<table>
<thead>
<tr>
<th></th>
<th>50×50</th>
<th>100×100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1K×1K</td>
<td>2K×2K</td>
</tr>
<tr>
<td>O</td>
<td>62</td>
<td>118.5</td>
</tr>
<tr>
<td>P</td>
<td>74.6</td>
<td>131.1</td>
</tr>
</tbody>
</table>

Note: O: Original, P: Pre-execution prefetching (unit: MB/s)

The first set of experiments were conducted with 25 processes, where each process performs the 2D-convolution of two tiles. The number of elements was set as 100 and 200, and the element size was set as 1KB and 2KB, respectively. The total accessed data was 256 MB, 512 MB, 1 GB and 2 GB, respectively. With pre-execution prefetching, the sustained bandwidth improved by up to 20.58% and the average improvement was 18.37%. The second set of experiments used 100 processes; the number of elements was set as 50 and 100, and the element size was set as 1 KB and 2
KB respectively. The total accessed data was the same as in the previous set of experiments. The sustained bandwidth increased by up to 20.32%, and the average improvement was 14.71%. Both sets of experiments verified that the pre-execution prefetching achieved considerable execution time reduction and sustained bandwidth improvement.
5.5.3 Performance Results with Post-execution Based Prefetching. We have conducted three sets of experiments: two with strided and nested strided patterns using PIO-Bench [78] and another with BTIO benchmark [109]. PIO-Bench is a synthetic parallel file system benchmark suite that is designed to reflect I/O access patterns appearing in typical workloads of real applications. This benchmark suite tests several I/O access patterns including sequential, simple strided, nested strided and random strided. The BT benchmark [109] is based on a CFD code that uses an implicit algorithm to solve the 3D compressible Navier-Stokes equations. A finite-difference grid is assumed, and systems of 5 x 5 blocks at each node are solved using a block-tridiagonal solver. This benchmark uses a matrix with a size of (102 x 102 x 102). Each PIO-Bench test is run with multiple strides by varying the number of processors 2, 4, 8, and 16. For BTIO we evaluated Class B I/O size with 4, 9, and 16 processors.

5.5.3.1 PIO-Bench Experimental Results. Figure 5.14 compares the I/O read bandwidth results of PIO-Bench with 1-d strides (from 512 KB to 4MB) using NFS with collective prefetch cache and native approach (without prefetch cache). Figure 5.15 shows the same bandwidth results with PVFS. In order to focus on the performance benefits of prefetching, we measured only the time for I/O read requests. Each reported result is the least value of multiple runs; we observed that the variation among performance numbers in multiple runs was negligible. With PIO-Bench, the amount of data transferred between disks and MPI processes is higher as the size of stride increases for the same number of I/O reads. Hence, there is an increase in read bandwidth as the stride increases. From the figures we can see that the I/O read bandwidth with prefetching is better than that with no
prefetching for all stride sizes with various numbers of processors. On average, the performance gain with prefetching for multiple stride sizes on NFS is ~23%. On PVFS, the performance gain is lower than that on NFS, probably because PVFS has optimizations for strided patterns to combine multiple accesses. Prefetching benefits are low for small number of processors as well as for small stride between successive I/O reads. As the number of processors increases, the performance gains increase to ~14%.

Figure 5.14. Bandwidth of PIO-Bench, with Simple Strided pattern on NFS

Figure 5.15. Bandwidth of PIO-Bench, with Simple Strided pattern on PVFS
Figure 5.16. Bandwidth of PIO-Bench, with Nested Strided pattern on NFS

Figure 5.17. Bandwidth of PIO-Bench, with Nested Strided pattern on PVFS

Figure 5.16 compares the I/O bandwidth results of PIO-Bench Nested strided pattern using NFS with and without prefetching, and Figure 5.17 compares them with PVFS. The difference in offsets between successive I/O reads is 2-d strided in these tests. It is noticeable that bandwidth reduces as the number of processes increase for smaller strides. The I/O read performance with prefetching is better than that with no prefetching in all cases here. The performance gain with prefetching is low for small number of
processors and it increases as the number of processors grows in both NFS and PVFS tests. In the 2-d strided I/O reads, there is a lack of locality, which requires loading a different page for each access. As the number of processors grows, serving each processor increases the load on I/O servers and results in poor performance. Prefetching methods benefits highly in these cases by sending requests early. The average performance gain on NFS is ~36% and that on PVFS for larger strides is ~20%.

5.5.3.2 BT-IO Experimental Results. Figure 5.18 and Figure 5.19 compare the I/O bandwidth performance of BTIO (Class B) with and without prefetching on NFS and on PVFS, respectively. We can see that on both NFS and PVFS, we can see I/O read bandwidth improvement with different number of processors. The read accesses have fixed 1-d stride pattern, and the prefetching benefits are similar to the case of 1-d stride in the PIO-Bench test (Figure 5.14 and Figure 5.15). The performance gain for all processors is same on NFS (~25%). On PVFS, with 4 processors, the gain is ~8%, and as the number of processors increase, the performance gain increases to 15% (9 and 16 processor tests).

![Figure 5.18. Bandwidth of BTIO Reads with Prefetching on NFS](image)
5.6 Application and Impact

As a summary of the discussion in this chapter, we have presented a specialized software approach to improving memory-disk stage data-access efficiency by exploring both pre-execution and post-execution analysis based data prefetching. We have presented the pre-execution prefetching system design, the pre-execution thread construction methodology, and the automation of the construction with a source-to-source pre-compiler that takes advantage of program slicing technique. The pre-execution prefetching is essentially an approach to trading computing capability for more effective I/O accesses. It explores the concurrency of computation and I/O operations well and hides the data-access delay effectively. We have also introduced a comprehensive pattern classification and I/O signature notation to study the post-execution analysis based data prefetching. We have presented the mechanism of trace collection, trace analysis and dynamic signature adjusting at runtime as well. We have carried out the prototype implementation within the most popular MPI-IO implementation, ROMIO system, and have presented the implementation related issues and solutions. The experiments have observed considerable performance improvement over existing approaches.
The software approach of latency tolerance technique presented in this chapter is promising and can benefit a variety of data-intensive applications, ranging from visualization applications, gaming applications, engineering system simulations, audio/video processing, parallel data mining; and a variety of scientific applications, such as molecular structure simulation, computational fluid dynamics simulation, geographical information systems, etc. The impact of this work has several folds. Firstly, our study has confirmed that data prefetching can significantly reduce memory-disk stage access latency and thus improve the overall system performance considerably. Secondly, the study of the design and prototype implementation of the pre-execution and post-execution analysis based prefetching provides valuable experience and resource to the community. It advances the state-of-the-art of the techniques of improving memory-disk stage access efficiency. Thirdly, we have developed the prefetching library and client-side prefetch cache, which can be further explored by interested researchers to develop other prefetching strategies. Lastly, the I/O access pattern classification and analysis presented in this chapter enhance the current understanding of the behavior of data accesses, and can help researchers in identifying further optimization techniques.
CHAPTER 6
RELATED WORK

Data prefetching is a promising technique for masking data access delay. Many efforts have been devoted to this research area. This chapter discusses the state-of-the-art of data prefetching techniques and compares them with the solution we introduce in this dissertation. We discuss and compare the existing studies in two categories, data prefetching techniques at cache-memory level and memory-disk level respectively, corresponding to the two-stage latency reduction solution of the Hybrid Adaptive Prefetching architecture. Part of the discussion has been published as survey articles in refereed journal and conference proceeding in [10][12]. In this chapter, we briefly review the existing work and analyze them.

6.1 Data Prefetching at Cache-Memory Level

Data prefetching techniques at the cache-memory level fetch data from main memory component into a cache level closer to the processor, usually the processor’s primary cache (level one cache) or secondary cache (level two cache). Data prefetching at this level is primarily in the form of hardware prefetching, while there also exist a certain number of software prefetching techniques that usually need the programmer or compiler’s intervention. We discuss the software prefetching and hardware prefetching techniques respectively in the following.

Software prefetching [2][53][62][94] instruments prefetch instructions to the source code either hand-coded by a programmer or inserted by a compiler during optimization phase. In either case, the software prefetching is often used for large amount of loops, which are very common in scientific computation. Such loops usually exhibit
poor cache utilization but have predictable memory referencing patterns, thus provide excellent prefetching opportunities. The major software prefetching techniques [2][62][94] are simple prefetching, unrolling the loop and software pipelining. The advantage of software prefetching is that it can take benefit of compile-time information to schedule prefetches more accurately than hardware prefetching, which is less likely to cause useless prefetching due to late prefetch or cache pollution due to early prefetch. Nevertheless, using explicit extra prefetch instructions for software prefetching involves extra program execution time and suffers certain performance penalty. In addition, software prefetching usually results in significant code expansion. Since the memory-disk level prefetching is generally software based, we discuss more about software prefetching in the following section.

Hardware prefetching does not require any modification to the binary or source code, and can benefit existing binary code directly. There is no need for programmer or compiler’s intervention. Commonly used hardware prefetching techniques include sequential prefetching, stride prefetching and Markov prefetching. Sequential prefetching [27][28] fetches consecutive cache blocks by taking advantage of locality. The One-Block-Lookahead (OBL) approach automatically prefetches the next block when an access of a block is initiated. However, the limitation of this approach is that the prefetch may not be initiated early enough prior to processor’s demand for the data to avoid a processor stall. To solve this issue, a variation of OBL prefetching, which fetches \( k \) blocks (called prefetching degree) instead of one block, is proposed. Another variation, called adaptive sequential prefetching, varies prefetching degree \( k \) based on the
prefetching efficiency. The prefetching efficiency is a metric defined to characterize a program’s spatial locality at runtime.

The stride prefetching approach [16][94] observes the pattern among strides of past accesses and thus predicts future accesses. Numerous variations have been proposed based on stride prefetching, and these strategies maintain a Reference Prediction Table (RPT) to keep track of recent data accesses. RPT provides a practical approach to implement stride prefetching, but the limitation is that only constant strides are recognizable.

To capture repetitiveness in data reference addresses, Markov prefetching [40] was proposed. This strategy assumes the history might repeat itself among data accesses and build a state transition diagram with states denoting an accessed data block. The probability of each state transition is maintained so that the most probable predicted data are prefetched in advance and the least probable predicted data references can be dropped from prefetching.

Other recent representative efforts in hardware prefetching include Kandiraju et al’s distance prefetching [41], Sun et al’s multi-level difference table (MLDT) prefetching [87], Nesbit et al.’s global history buffer prefetching [64][65], Zhou’s dual-core execution (DCE) approach [101] and Solihin et al.’s memory-side prefetching [81]. Distance prefetching uses Markov chains to build and maintain probability transition diagram of strides (or distances) among data accesses. MLDT prefetching uses time-series analysis method to predict future accesses in a sequence, by finding the differences in a sequence to multiple levels. GHB prefetching provides more access histories to improve prefetching accuracy. DCE prefetching was proposed specifically for multicore
architecture. It employs idle cores to execute future loop iterations to bring data into cache in advance. The memory-side prefetching approach uses a memory processor residing within main memory to observe data access histories and prefetch data proactively upon prediction. It is usually distinguished as push based prefetching from traditional pull based prefetching.

Without the benefit of programmer or compiler hints, the effectiveness of hardware prefetching largely relies on the accuracy of prediction strategies. Incorrect prediction brings useless blocks into cache, consumes memory bandwidth and might cause cache pollution. To increase prefetching accuracy and coverage, hardware prefetching strategies should be more comprehensive. On the other hand, it is desired that data prefetching should support various algorithms and make dynamic selections. This is because that access pattern is decided by application features, and different prefetching algorithms are required for assorted applications. All these observations motivate the Hybrid Adaptive Prefetching architecture discussed in this dissertation. Our solution provides a generic and prefetching-dedicated Data-Access History Cache structure, and an algorithm-level hardware adaptive data prefetching based on it. Some existing literature [27][82] provide certain forms of adaptation, however, most of them targets at adapting the prefetch degree and prefetch distance only. Our work is motivated from the fact that no single prediction algorithm can work universally well for all applications. The adaptation at an algorithm-level is a necessity. The specialized hardware approach of Hybrid Adaptive Prefetching architecture provides such a solution. It has a real potential in bridging the gap between processor and memory and improving data-access efficiency.
6.2 Data Prefetching at Memory-Disk Level

Several previous studies observed that certain data accesses at memory-disk level follow regular patterns. Miller and Katz [60] studied several applications running on a Cray Y-MP vector computer and detected that these applications access data in chunks ranging from 32 KB to 512 KB. They concluded that I/O access sizes of the tested applications were relatively constant, cyclic, bursty, and predictable. Keeton et al. [42] observed small and large jumps (stride or difference between file offsets) among sequential accesses and interferences between concurrent accesses. Pasquale et al. [67] observed similar regularity among I/O patterns on a Cray C90. Crandall et al. [25], Madhyastha et al. [57], and Smirni et al. [79][80] studied scalable I/O applications and provided a classification of patterns based on three dimensions of file access features: type of I/O operations (read/write), sequentiality, and size of I/O requests. In this dissertation, we classify access patterns further in the dimensions of repetitiveness and temporal behavior. Using these dimensions, we provide a representative notation for I/O accesses. Marathe et al. [58] used a notation to represent memory access traces that represents the length of a memory access and sequentiality. Our pattern representation notation is for I/O accesses and covers more information in multiple dimensions. Each of these dimensions can contain complex nested pattern notation. This notation not only reduces the size of traces, but is also useful in selecting prefetching strategies.

These studies in I/O access pattern analysis revealed that data prefetching can work well for memory-disk stage accesses. Several existing work [30][44][59][66] have studied data prefetching via predicting future accesses based on observed patterns among past access history. Kotz and Ellis [44] studied prefetching for reducing disk access
latency in early 1990s, and were among the initial investigations of heuristic prediction based approaches. Ding et. al. recently proposed a DiskSeen approach in [30] to reveal disk layout to the file system and aid data prefetching in the file system with prediction-based approach. Their approach in essence exploits data locality and fetches data in advance with history-based predictions. Papathanasiou and Scott [66] suggest a form of aggressive prefetching with higher prefetching degree and appropriate prefetching distance for memory-disk level prefetching. In the Hybrid Adaptive Prefetching architecture, we have introduced I/O signature based prefetching for I/O accesses with well-formed perceivable patterns. This method also reduces runtime processing cost and makes dynamic prefetching more feasible.

When accesses lack regularity or even totally random, the pre-execution analysis based prefetching shows its special strength, though it also works well for accesses with regularity. The fact that considerable amount of I/O accesses are complex and irregular have been confirmed by many existing studies [45][52][74][75][79][96]. Several studies of I/O accesses on distributed memory systems such as CM-5, iPSC/860, and the Intel Paragon XP/S [25][45][74] show that certain I/O requests are small and have irregular patterns. Madhyastha et al. [75] and Smirni et al. [79] studied scalable I/O applications and also concluded that many I/O accesses are small, non-contiguous and irregular. Although numerous studies have been conducted and several well-known strategies, such as collective I/O and data sieving [75][92], have been proposed and used to combine small I/O requests into large ones, many small I/O requests cannot be eliminated due to the inherent nature of the applications. File-system level parallelism (i.e., parallel file systems such as Lustre [24], PVFS [13][50] and GPFS [77]) and disk-level parallelism
(usually in the form of RAID) can greatly increase the I/O throughput, they are not capable of reducing the I/O latency effectively, especially in the case of a large number of isolated or small accesses. In the Hybrid Adaptive Prefetching architecture, we introduce a pre-execution analysis based prefetching that works for any application, especially for those with irregular and complex accesses, and has a high accuracy in discovering future references. The Hybrid Adaptive Prefetching architecture essentially combines the heuristic prediction approach for regular accesses and pre-execution approach for irregular or even random accesses.

Some other representative prefetching approaches include Chang and Gibson’s SpecHint [14][15], Patterson and Gibson’s informed prefetching TIP [69][70], and Yang’s AASFP approach [99]. All of these approaches demonstrate that it is fully feasible to speculate future I/O accesses in time and reveal this information to the underlying file system to fetch data in advance. However, their approaches are conservative and only utilize idle cycles to perform speculation. Our proposed approach combines the merits of existing approaches, and provides a more comprehensive speculative prefetching.

There are several efforts in hiding data access latency in other directions, such as compilation optimization and providing a caching layer at a library level. Many researchers introduced advanced compilation techniques for I/O optimization. Brezany et al. [8] have developed VIPIOS that can be used by an optimizing compiler. Targeting out-of-core datasets, Bordawekar et al. [4][5] have presented several algorithms to optimize communication and to reorder stencil computations. Mowry et al. [63] have developed compiler-inserted I/O prefetching. A common problem with compiler
optimizations is that they are not effective with dynamic nature of I/O accesses. Moreover, making compilers perform complex analysis in search of optimizations increases compilation time severely. Collective caching [47][48] and active buffering [55][56] are caching optimization examples. Collective caching is an effective solution and can benefit both read and write accesses. We have customized the global buffer cache maintained by collective caching as the prefetching destination in the memory-disk level prefetching. Our prefetching approach is a complement to existing caching approaches and can improve data access performance further.
CHAPTER 7
CONCLUSION AND FUTURE WORK

As memory and disk speed lag far behind processor speed, data-access delay has a severe impact on overall system performance. The preliminary investigation has shown that data-access performance has become a bottleneck and a dominant factor that decides the actual sustained performance for high-performance computing and high-end computing system. The research study on the scalability analysis of emerging multicore architecture and conventional parallel and distributed computing systems confirms that the sustained performance tends to be limited by data accesses.

In this dissertation, we present a Hybrid Adaptive Prefetching architecture to improve data-access performance via a two-stage latency reduction solution, cache-memory stage and memory-disk stage. The Hybrid Adaptive Prefetching architecture combines heuristic prediction, pre-execution and post-execution analysis based prefetching to work for both regular and irregular accesses, and to achieve an effective overall data-access performance improvement. In this chapter, we summarize our contributions of this research study and discuss the potential future work.

7.1 Research Contributions

Our contributions in this dissertation study have several folds:

- Firstly, we have shown the great need of the research in improving data-access performance. We have studied the achievable scalability and potential performance limitation of emerging multicore processor architecture, and traditional parallel and distributed processing architecture. We analyzed the scalability of multicore architecture with scalable computing concept and from
both sequential processing and data-access constraints. These analyses reveal that data-access performance is the dominant limiting factor for the sustained performance of multicore architectures. The multicore architecture with constant data-access latency can become perfectly scalable. Additionally, we studied the scalability and performance constraints of parallel and distributed system with a novel model, isospeed-efficiency model. This performance evaluation model provides insightful guidance to the design and development of scalable computing systems and algorithms. The scalability study of the multicore and the parallel and distributed architecture has revealed that innovative techniques of reducing data-access latency are necessities to make contemporary high-performance/high-end computing systems scalable.

- Secondly, motivated from the scalability study and the identified limitations of existing work on reducing data-access latency, we propose a Hybrid Adaptive Prefetching architecture to enhance data-access efficiency by exploring the benefits of comprehensive prefetching strategies. The Hybrid Adaptive Prefetching architecture employs a two-stage latency reduction mechanism, with a specialized hardware approach of latency reduction at cache-memory level, and a specialized software approach at memory-disk level. The proposed solution addresses the limitations of existing data prefetching studies well with a general feedback-controlled adaptation mechanism, a novel pre-execution strategy to increase prefetching accuracy and coverage, and a systematic prefetching system that boosts data-access performance.
Thirdly, we have introduced a novel concept of a prefetching-dedicated cache considering the evolvement trends of both hardware technologies and application features. We provided the design of such a generic prefetch cache structure, named Data Access History Cache (DAHC), and performed extensive simulation with an enhanced SimpleScalar simulator to validate its design and verify its functionality. Based on the newly introduced prefetch cache structure, we presented the data prefetching methodology and demonstrated DAHC’s support for various prefetching algorithms with representative exemplar algorithms. In addition, we introduced a feedback-controlled dynamic adaptive prefetching. This adaptive prefetching is capable of identifying the diverse nature of applications and adapting to suitable algorithms. The simulation experiments have shown that DAHC-based prefetching and feedback-controlled adaptive prefetching can achieve considerable cache miss rate reduction and IPC (Instruction Per Cycle) improvement. These studies complete the specialized hardware approach of improving cache-memory stage data-access efficiency.

Fourthly, we have introduced an innovative pre-execution analysis based prefetching approach to reducing memory-disk stage data-access latency, as well as a post-execution analysis based prefetching. The pre-execution approach essentially explores the concurrency of computation and I/O accesses, and utilizes available computing capability to speed up data accesses. The post-execution approach identifies application-specific access characteristics and can enhance the effectiveness of prefetching at runtime. We have presented the design, methodology, challenges and solutions of these approaches and developed a
prototype implementation with collective client-side prefetch cache and existing ROMIO library. The experimental results have confirmed considerable execution time reduction and data access bandwidth improvement. These approaches complete the specialized software approach of improving memory-disk stage data-access efficiency.

As applications are becoming be more and more data intensive, data-centric computing and novel techniques that can effectively reduce data-access latency will have a fundamental impact to high-performance and high-end computing. Data-access performance is predicted to be the most critical factor that decides the sustained performance of computing systems and applications. The recent advances in Grid computing [33] and Cloud computing [7] have shown the great necessity of migrating to data-centric computing and the great need of improving data-access efficiency as well. The Hybrid Prefetching Architecture and associated innovative data prefetching strategies presented in this dissertation are critical techniques in data-centric computing era. They can benefit many applications, including scientific simulation, visualization and multimedia applications, engineering system simulation, parallel data mining; information retrieval, etc. The impact of the research study in this dissertation is profound.

7.2 Future Work

In terms of the future research, we plan to extend the current work and explore two major directions: exploiting global-aware collective prefetching and exploiting global-aware caching bypassing optimization.
The investigation of collective prefetching optimization is motivated from the deficiency that current parallel I/O prefetching strategies are in an uncoordinated manner and usually serve each individual process independently. Multiple processes working on the same application, however, have strong correlations in I/O accesses. These processes can concurrently access a sequence of blocks, or access a locally non-contiguous but globally contiguous chunk of data (e.g. strided data accesses with a block-cyclic data distribution). An appropriate parallel I/O prefetching strategy should be designed to take advantage of these correlations and initiate a coordinated and global-aware collective prefetching.

The other direction we plan to conduct further research is exploiting cache bypassing optimization to improve cache utilization. Since there are multiple cache levels in the data-access hierarchy, e.g. disk level, file system level, and user library level, caching data at multiple levels reduces the efficiency. We plan to investigate a cache bypassing strategy that allows data cached in fewer levels based on its reuse characteristics. The investigation on global-aware cache bypassing is expected to lead to a deeper understanding of caching mechanism and an approach to improving caching performance effectively.
BIBLIOGRAPHY


