2015

Dynamic data shapers optimize performance in Dynamic Binary Optimization (DBO) environment

Varun Kumhar Venkatesan
Iowa State University

Dynamic data shapers optimize performance in Dynamic Binary Optimization (DBO) environment

by

Varun Venkatesan

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Akhilesh Tyagi, Major Professor
Pavankumar R Aduri
Zhao Zhang

Iowa State University
Ames, Iowa
2015

DEDICATION

I dedicate this thesis to my family, whose unconditional care has been at the heart of all my accomplishments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to take this opportunity to thank all those who helped me with the research and writing of this thesis. I would like to express the deepest appreciation to my advisor, Dr. Akhilesh Tyagi for his inspiration, teaching and support throughout this work. His insights and words of encouragement have often inspired me and were instrumental in helping me succeed in my research. Without his guidance and persistent help, this work would not have been possible.

I am also greatly thankful to my committee members, Dr. Pavankumar D. Aduri and Dr. Zhao Zhang for their valuable suggestions and contributions to this work. I would also like to sincerely thank Swamy D. Ponpandi, my research group member for his immense contribution and insight throughout this work.

I also appreciate my family and friends who helped me tremendously over the years and have inspired me to reach greater heights. I would like to thank them for their guidance and support throughout my career.

ABSTRACT

Processor hardware has been architected with the assumption that most data access patterns would be linearly spatial in nature. But, most applications involve algorithms that are designed with optimal efficiency in mind, which results in non-spatial, multi-dimensional data access. Moreover, this data view or access pattern changes dynamically in different program phases. This results in a mismatch between the processor hardware's view of data and the algorithmic view of data, leading to significant memory access bottlenecks. This variation in data views is especially more pronounced in applications involving large datasets, leading to significantly increased latency and user response times. Previous attempts to tackle this problem were primarily targeted at execution time optimization. We present a dynamic technique piggybacked on the classical dynamic binary optimization (DBO) to shape the data view for each program phase differently resulting in program execution time reduction along with reductions in access energy. Our implementation rearranges non-adjacent data into a contiguous dataview. It uses wrappers to replace irregular data access patterns with spatially local dataview. HDTrans, a runtime dynamic binary optimization framework has been used to perform runtime instrumentation and dynamic data optimization to achieve this goal. This scheme not only ensures a reduced program execution time, but also results in lower energy use. Some of the commonly used benchmarks from the SPEC 2006 suite were profiled to determine irregular data accesses from procedures which contributed heavily to the overall execution time. Wrappers built to replace these accesses with spatially adjacent data led to a significant improvement in the total execution time. On average, 20% reduction in time was achieved along with a 5% reduction in energy.

CHAPTER 1. OVERVIEW

This chapter introduces the increased memory latency and execution time problems caused by non-spatially adjacent data access, some of the applications where this problem is prevalent, Data Shaping as a solution to deal with this problem and some background on the intricacies of this issue.

1.1 Introduction

Non-spatial data access has been a leading contributor to memory access latency in most applications, resulting in increased program execution times and lowered response times. The primary contribution of this work is the development of an effective approach to reduce the data access time in most commonly used applications. The negative impacts of non-spatial data access have been lowered by creating a *Dynamic Dataview* of spatially adjacent data, which replaces these irregular data access patterns at runtime. The dynamic binary optimization capabilities of HDTrans[27][28] have been leveraged to help achieve this goal. This study also evaluates the performance of three data stores that host the dynamically shaped data - the ***Dynamic Data View Array(DDVA)***, Tagless D-Cache and Scratchpad Memory. This implementation also helps reduce the energy consumption for data access. The applicability of this scheme to various common applications in the SPEC2006 benchmark suite[12] were studied.

The code block in Figure 1.1 is a snippet of a linked list access function, where data of all nodes are being accessed in a repetitive manner.

Note that the accessed data in the DS0 linked list above has no spatial locality. The high level meta-wrapper or data-shaper specification proposed in this work could take on a

```
/* Iterative structure for
                temporal locality */
for(i=0; i < k; i = i+1){

  /* Linked List data access loop */
  for(j=0; j < N; j = j+1){
    data = DS0.data;
    DS0 = DS0.next;
  }

}
```

Figure 1.1 Original Code Block with non-spatially adjacent data access

form as shown in Figure 1.2. The wrapper is a small piece of code which transforms the architecture dependent data view (irregular accesses of Figure 1) into an algorithm amenable data view (potentially performance and energy optimized accesses) through temporary storage structures, referred to as the Dynamic Data View Array. In Figure 1.2, a simple linear array (buf[j]) is used to coalesce data from the structure DS0 and any further reference to DS0 is piped to the DDVA structure, buf[j]. Accesses to more than one field of complex data structures could be transformed to one or more DDVA structures.

The following sections introduce the targeted applications and the need for performance optimization in these scenarios. Also, some background information on significance of properly exploiting Spatial and Temporal Locality to deliver benefits has been provided. The principles behind Dynamic Binary Optimization and the predominant frameworks which deliver this capability have been detailed. A special focus has been placed on detailing the internals of HDTrans, the Dynamic Binary Optimization framework used in our implementation.

1.2 Targeted Applications

Recent applications are best placed to leverage highly advanced hardware processing capabilities and memory building blocks. They are optimized for multitasking, parallel processing and extensive computation needs. These applications often have to deal with processing huge datasets and may also need to generate and store results which tend to be even larger. The

```
/* Wrapped scope with the Data Shaper */
for(i=0; i < k; i = i+1){

  //DATA WRAPPER
  /* Temporal Locality - First Epoch */
  if(i == 0) {
  /* Linearize or Shape the DS0 data */
    for(j=0; j < N; j = j+1){
      buf[j] = DS0.data;
      DS0 = DS0.next;
    }
  }
  //END DATA WRAPPER

  /* Data fed from Shaped buf */
  for(j=0; j < N; j = j+1){
    data = buf[j];
  }

}
```

Figure 1.2 Optimized Code Block with Data Shaping and spatially adjacent data access

trend of algorithmic implementation of these applications has been rapidly shifting towards that of achieving a higher efficiency. In this process, data tends to stored in memory in a largely non-spatial manner. The performance benefits delivered by this work are demonstrated in the SPEC2006 suite of applications, including, mcf and h264_ref. This work might also deliver huge benefits when applied to other common applications with huge data processing needs, such as those detailed below:

1. IMDB generates and maintains moderately large, real database of movies.

2. Wikimedia maintains Page View statistics with hourly updates for its suite of projects including Wikibooks, Wiktionary, Wikimedia, Wikipedia mobile, Wikinews, Wikiquote, Wikisource, Wikiversity and Mediawiki. This involves maintaining and processing huge datasets with up-to-date information. The stream is available unsampled as gzipped hourly files from their website.

3. The FlightStats database maintains flight on-time arrival data and a host of other related flight information.

4. Amazon AWS provides Public Data Sets with a large variety of data such as the mapping of the Human Genome, NASA NEX Earth science datasets, Landsat satellite imagery of all land on Earth, Common Crawl Corpus composed of over 5 billion web pages and the US Census data which require hours or days to locate, download, customize and analyze.

5. Weather Underground provides extensive Weather History statistics for numerous cities worldwide over different times in the past.

6. Wireless sensors generate massive data due to the high resolution sensing requirements in numerous applications.

7. Scientific simulations too generate large volumes of data due to increased scales and resolutions of simulated domains.

8. The North American electric power grid operations generate 15 Tera Bytes data per year.

9. Social networking sites such as Facebook capture and store Peta Bytes of heterogeneous information.

10. Google sorts through 20 Peta Bytes everyday.

11. The Large Hadron Collider(LHC) at the Center for European Nuclear Research(CERN) generates raw data at a rate of 2 Peta Bytes per second starting from 2008.

12. The Large Synoptic Survey Telescope will generate several Peta Bytes of new image and catalog data every year. The Square Kilometer Array will generate about 200 Giga Bytes of raw data per second that will require Peta Flops (or possibly Exa Flops) of processing to produce detailed radio maps of the sky.

1.3 Problem Statement

Hardware implementation tends to be much simpler for a linear layout of address spaces. It is for this reason that most commercial processors have a spatially linear view of data. Applications, on the other hand, involve extensive use of optimal algorithms, tailor-made for program efficiency. This often results in a spatially non-adjacent view of data from the application. This mismatch between the processor's view of data and the algorithm's view of data results in several performance bottlenecks such as an increased memory access latency, increased program execution times, increased memory bandwidth between various levels of the memory hierarchy and greater power consumed for each data access. This performance loss is much more obvious in emerging applications, due to a magnified mismatch resulting from the highly non-spatial algorithmic data views.

There have been several attempts in the research community to address the pressing needs of this problem. Even though some of these approaches claim to be successful in reducing memory access latency and execution time, they introduce unintended increases in memory bandwidth and energy use. They also add additional overheads since these techniques are not dynamic.

1.4 Background and Significance

The following subsections discuss in detail the significance of Spatial and Temporal Locality in helping reduce access latencies, Dynamic Binary Optimization as an approach to address these issues, some of the most commonly used Dynamic Binary Optimization frameworks (DynamoRIO, QEMU, Pin, Valgrind and HDTrans) and the internals of the HDTrans framework as used in our implementation.

1.4.1 Spatial and Temporal Locality

The two classical attributes of data represented as linear memory mapped data views are spatial locality and temporal locality. Spatial locality refers to the program property that if the data at address A is accessed now (at time T), data in its spatial neighborhood (addresses in the range A - e to A + e) is likely to be accessed in the near future (within time range T to T + t). Temporal locality states that if data at address A is accessed at time T (now), it is likely to be accessed again in near future (between time T and time T + t). Such locality allows a processor to pay 40-100 cycle cost for a data item to fetch it from memory the first time it is seen. However, all future accesses resulting from temporal locality end up costing just one to two cycles if the data item is cached on the on-chip L1 cache on the first access. Data is fetched into L1 cache in a chunk of multiple data items, called a cache block in order to amortize the costs of multiple accesses. Hence, when we bring in the data item at address A, we also bring in data items at address A+1, A+2, ..., A+k for block size k+1. If a program exhibits spatial locality then the one-time cost of 40-100 cycles for access to the data item at address A is amortized over k+1 items.

Assume that memory costs 100 cycles, L1-cache access takes 1 cycle, and block size equal to 8 data items. Fetching 8 data items without any locality costs 800 cycles at 100 cycles per data item. However, spatial locality reduces the cost to 107 cycles at 13 cycles per data item. If each data item is used 9 more times in the near future after the first fetch - a property of temporal locality, then the average access time goes down to 11 cycles per fetch from 100 cycles per fetch.

The memory bandwidth has always been a show-stopper in computer architecture - hence the popular term memory wall. Exploitation of locality is the primary mechanism to overcome the memory wall. Dynamic data shaping takes this a step further and rearranges data along a view ideal for the current program context. This helps speed up the program execution by a factor of 10 or more. Platform independence of these optimizations makes them very appealing for performance enhancement.

1.4.2 Dynamic Binary Optimization

Binary Translation (BT) is a technique to convert binaries available in one ISA into another ISA[17]. A widely used sub-category of Binary Translation is Binary Instrumentation, which is a special technique to observe a binary's behavior by inserting probes into it. Some of the most common applications of Binary Translation are in profiling program performance, branch or memory trace generation, protection, emulation and debugging. Some Binary translators like QEMU[5] and Shade[11] offer the advantage of fast emulation. Many binary translators specialize in migrating the binary from one architecture to another. IA-32EL[4] translates IA-32 code to IA-64 code. There are also some binary translators which perform same ISA translation to help in virtualization, as in the case of VMware GSX, and in optimization, as in the case of Dynamo[3] and Adore[20]. DBT (Dynamic Binary Translation) based tools can be further classified based on their underlying architecture. They may either be Instrumentation based, Migration based, Fast Emulation based or Dynamic Optimization based. PIN[21] and DynamoRIO[6] are some Instrumentation-based DBT tools. IA32EL is a Migration-based DBT tool. QEMU and Shade are Fast Emulation based tools. Dynamo[3], HDTrans and Adore are Dynamic Optimization based tools.

Binary Translation can either be static or dynamic. Static BT (Static Binary Translation)[17] performs interpretation, which happens one instruction at-a-time. Dynamic BT or Dynamic Binary Optimization (DBO)[17] is optimized for repeated instruction execution one block-at-a-time. Static BT translates once, runs many times and allows aggressive code optimizations. But it is less adopted due to Code discovery problems. DBO is more widely adopted since their implementation techniques are efficient and well understood. Dynamic Binary Optimization (DBO) performs dynamic translation and execution of application binaries by actively carrying out runtime code instrumentation. The entire program is divided into basic blocks, with each basic block delimited by a conditional statement. DBO maintains a code cache where each basic block is copied into, before the DBO framework performs the required instrumentation and then executes it. The DBO system ensures that only translated code from the code cache is executed. Indirect branches may form part of some control flows, and these can't be resolved statically. So the DBO system is invoked to handle such scenarios. The DBO system ensures that the code cache contains the branch destination. The DBO system also forms a longer trace by merging the most frequently executed basic blocks together. This arrangement helps reduce runtime overheads by ensuring that the most common hot paths are executed completely from the code cache without invoking the DBO system.

DBO helps improve performance with optimization and profile-directed feedback hidden from the user. DBO also ensures compatibility with legacy architectures by making Architecture a Layer of Software. The software translator translates once and saves in memory. This reduces hardware complexity and power use. Also, no changes are needed in existing code while using DBO. DBO is very reliable and helps to work around some hardware bugs. DBO makes best use of available runtime profile information and post-link-time program information. Static optimization only emulates a single source architecture, while DBO can emulate multiple source architectures.

DBO tools also maintain metadata which helps analyze the state of memory locations during program execution[24]. Some of the benefits delivered using this metadata include, identifying if a memory location is allocated, if it contains secure data, or the number of times it has been accessed. The granularity and size of metadata varies vastly from one tool to another. The huge variations in granularity of instrumentation information maintained may range from a coarse granularity at a function-level, page-level, or object-level, in some tools, or at a basic block, word, or even bit-level granularity in other tools. Separate memory regions are used for metadata allocations to avoid interference with the data layout assumed by the original program[10]. Whenever the program code operates on data, the DBO tools would insert code that executes the right operations on the corresponding metadata to ensure accuracy. The exact length of the inserted code may vary from one tool to another.

There are numerous benefits to Dynamic Binary Optimization, as detailed in [17] and [1]:

1. Legacy code where source is unavailable can be optimized.

2. Dynamic Optimization still helps maintain high code quality.

3. DBO is not limited in optimization scope. It can cross boundaries across Indirect Calls, Function Returns, Shared Libraries and System Calls.

4. Translated basic blocks can be layed out contiguously in order they are naturally visited. This helps ICache Performance.

5. DBO is also compatible between VLIWs of different sizes and generations.

6. DBOs are easily upgradable. If better compiler algorithms are found, only a software patch is needed to install them. Also, future architectural improvements are transparent to the user.

7. DBO is very reliable. If a bug is found in dynamic translator, a software patch is sufficient to fix it. Also, some hardware bugs can be worked around by the translator.

8. DBO also helps in attaining High Chip Yield since it is a software based approach and not hardware based. Smaller chips with higher yield can be realized.

9. DBO helps keep the Hardware Simple and Fast. Intelligence is in software, allowing simple in-order implementations.

10. DBO offers a Wide scope for ILP. It can look at arbitrarily long fragments of code.

11. Ability to detect and optimize program phases.

As with any approach, there are some minor setbacks with Dynamic Binary Optimization, as briefly discussed in [17] and [1] and detailed below:

1. High level semantic information (such as exceptions) may not be available.

2. Takes away cycles from program. This is because DBO takes memory and resources from the emulated machine. DBO is especially slow at start. So there are potential realtime difficulties.

3. Debugging can be difficult since the Target machine code is several times removed from source code and the behavior can be non-deterministic in a real system.

4. Even virtual machines can be emulated in static optimization, while only real machines can be emulated using DBO.

5. Static optimization can be done at the Full System level, while DBO is a User based approach.

6. Static optimization is OS Independent, while DBO is OS Dependent.

7. Also, the target architecture should have hardware support for frequently used features of each legacy architecture such as opcodes, Condition code registers, Floating point formats, Timer registers, Segment registers and Address translation.

8. Slow translators and interpreters require high code reuse to amortize the time they take.

9. Large caches in these translators allow more code reuse at the cost of memory and lower adaptability to code changes.

Some of the most commonly used DBO tools have been described in greater detail in the following sections.

1.4.2.1 DynamoRIO

DynamoRIO builds basic blocks of the target application, and then translates each basic block on demand into the code cache. It then links the translated blocks together. This activity happens in parallel with that of their original counterparts in order to replicate the original control flow within the cache. Incremental updates happen at the code cache as new blocks of the target application are executed. This happens until the application runs entirely within the cached copy. In particular, indirect branches are especially well handled by the DynamoRIO. The many targets of an indirect branch are specified by addresses in the memory space of an application. The addresses of the branch targets in DynamoRIO would always reference the memory-untranslated code of the original application outside the code cache. This is because the data flow of the target application is identical to the native run. DynamoRIO has a lookup routine within its code cache that helps find the translated code fragment corresponding to a branch target. Indirect branches are redirected to this lookup routine where they find the translated code fragment to make a jump to, thereby avoiding execution from returning to the original application.

1.4.2.2 QEMU

QEMU detects code changes on the page by performing code translation using one of two available strategies. In the the first strategy, QEMU marks the page as read-only and then handles the fault as if it were a write event. This strategy is very similar to that employed by DynamoRIO. In the second strategy, QEMU makes use of the QEMU softmmu layer's software TLB to effectively map the guest page table to the host page table. QEMU uses this softmmu layer to translate targets in a guest write to a page of memory to that of the corresponding host page. Also, this translation point can be configured to trap into QEMU for code change handling.

1.4.2.3 Pin

Pin translates all code into traces. Pin instruments the heads of traces containing dynamically generated instructions to check if any of those instructions have changed. Pin also efficiently handles situations where executable permissions are removed from pages containing Dynamically Generated Code. Pin would invalidate all traces containing code fragments translated from such pages. Pin is an optimal tool when used during periods of frequent code generation when compared to other tools that instrument every store. This is because, traces will be executed much less frequently than stores. But, Pin is disadvantageous for scenarios where the JIT engine is dormant and the generated traces are repeatedly executed as the cost increases dramatically. This is because, instrumented checks rarely discover code changes and are executed far more often than stores. DynamoRIO and QEMU are susceptible to the concurrent writer problem, since they rely on detecting code changes at the time of write. Pin avoids this pitfall since it relies on detecting code changes at the time the translated traces are executed. Thus, individual traces can be selectively flushed from the code cache while using Pin.

1.4.2.4 Valgrind

Valgrind synchronizes its code cache with dynamically generated code using two strategies. The first strategy instruments every dynamically generated basic block with a check for modified code. This strategy is very similar to that of Pin's. The second strategy is more efficient and involves compiling the target application with a source code annotation that is translated into a code cache flush event. But the relatively higher efficiency of the second strategy does not lead to significant improvements in performance. This is because, there is a massive slowdown in Valgrind's translation of basic blocks through a three-value IR. For maintaining compatibility, DynamoRIO implements some of Valgrind's annotations.

1.4.2.5 HDTrans

HDTrans[27][28] performs IA-32 to IA-32 binary translation with very simple and effective translation techniques. It is a very lightweight system and also uses established optimizations such as trace linearization and code caching. HDTrans is the Dynamic Binary Optimization framework used in our implementation primarily due to its modularity, simplicity, resourcefulness and open-source nature . HDTrans executes in a coroutine fashion with the binary image of the application to be translated. It maintains basic blocks, which are a sequence of straight-line instructions bracketed by branches. These blocks are translated into a Basic Block Cache(BBCache). HDTrans also maintains a directory of all such translated basic blocks indexed by source program counter. The HDTrans system doesn't remove translated basic blocks from the BBCache. The BBCache and the translation directory are discarded only in scenarios where the BBCache becomes full. The translation process starts over when this happens. Other existing translators follow a complex translation strategy with intermediate code generation, trace optimizations and register reallocation. In these translators, most of the execution time is spent in the code cache, thereby reducing the benefits gained from translation significantly. HDTrans avoids these pitfalls by avoiding intermediate code generation, target code optimization and register re-allocation. HDTrans performs efficient register management and maintains a small cache footprint.

The HDTrans translator is table-driven to reduce the overall cache footprint. The rules for decoding each instruction and the emitter function to be used in each case are maintained in this table. Each entry in the table occupies a single cache line, and a maximum of three entries are visited for every instruction decoded[27]. Instrumentation may not be explicitly needed in some scenarios. In these cases, most of the instructions are translated by copying them without alterations into the BBCache. The translator needs to remain in control of the application in scenarios involving instructions with control flows. HDTrans has dedicated routines which ensure that such scenarios are appropriately handled. HDTrans ensures that all dynamically active basic blocks are translated as execution proceeds and a steady state is arrived at. HDTrans significantly reduces the overall cost of translation, even though instrumentation is restricted to one instruction at a time. HDTrans performs trace linearization, but doesn't support any other target code cache optimization schemes. This is in direct contrast to the several optimizations done by Pin. Pin performs optimization of the instrumentation code and also supports several other sophisticated run time optimizations.

Some of the architectural features of HDTrans are explained in greater detail in the following section.

1.4.3 HDTrans - Architectural Overview

HDTrans performs simple translation, yet achieves satisfactory performance. HDTrans emits code that is competitive with the best existing translators, but has significantly lower startup and translation overheads. The architectural features of HDTrans that help in handling different branching schemes and program flows have been summarized in this section, as detailed in [27] and [28].

1.4.3.1 Direct Branches

Whenever HDTrans encounters direct branches, it seeks to maximize the re-use of translated basic blocks. HDTrans linearizes conditional branches assuming that the branch is not taken. It the destination is already translated, it emits a jump to it. Otherwise, HDTrans follows unconditional jump targets and later elides the jump. It doesn't follow call targets and keeps translating past the call instruction. For conditional jumps, the control flow branches to an exit stub and fixup occurs later. HDTrans records the destination as a new basic block in its basic block directory.

1.4.3.2 Indirect Branches

In the case of indirect branches, the destination is not known until runtime. HDTrans uses a hash table called the sieve to search for the translated destination. The hash is based on the untranslated destination address and uses 2^{15} buckets. The Sieve is implemented as blocks of code rather than blocks of data. This helps reduce register pressure and prevents D-cache pollution. Also, HDTrans also doesn't inline most frequently used destinations.

1.4.3.3 Return

The Return is the most important indirect branch in terms of dynamic frequency. Stack introspection is a huge problem associated with returns. So, HDTrans uses a Return Cache to deal with these problems. The Return Cache is a fixed size D-space direct-mapped hash table of BBcache (Basic Block Cache) addresses. It is indexed by the untranslated start address of the returning procedure. The Return Cache is very small in size and uses only 2^8 buckets.

1.4.3.4 Multi-threading

HDTrans features Inter-thread translation sharing since the variance of multiple threads existing across application domains is very high. The various versions of HDTrans adopted different design trade-offs between memory consumption and simplicity. The latest version of HDTrans, as used in this implementation favors simplicity, with each thread having its own BBcache (Basic Block cache) and machine state. So, there is no need to deal with thread interference during BBcache flush. There is also no measurable degradation in performance when compared to the single threaded version.

1.4.3.5 Signal Handling

Using HDTrans, signals are treated as threads of execution scheduled at the point of arrival. Each signal thread is run in its own BBCache and machine context. This signal handling support doesn't place any additional overhead on the HDTrans translator. Also, HDTrans doesn't support introspective signals.

1.4.3.6 Hybrid Translation

HDTrans incorporates Hybrid Translation in its internal engine. It uses static translation to avert run-time translation overhead and high startup costs. It translates as many blocks as possible statically. So, the overhead of the dynamic translator is reduced to indirect branch overhead. An overwhelming majority of dynamically executed basic blocks are identified using efficient disassembly techniques.

1.4.4 Memory Organization

An understanding of memory organization is essential to analyze the source of increased latency and data access energy across applications. This section discusses memory organization in an elaborate manner, as seen from a multi-core processor's perspective. Figure 1.3 provides a high level representation of the memory hierarchy in a multi-core processor.

The following sections describe the various components of the memory hierarchy in a detailed manner.

1.4.4.1 Registers

Processor Registers account for a small amount of memory that can be accessed faster than other sources in the memory hierarchy. Almost all processors load data from a larger memory into registers, where it is used for arithmetic computation as part of machine instructions. Manipulated data is then stored back into Main Memory. Modern processors also have duplicates

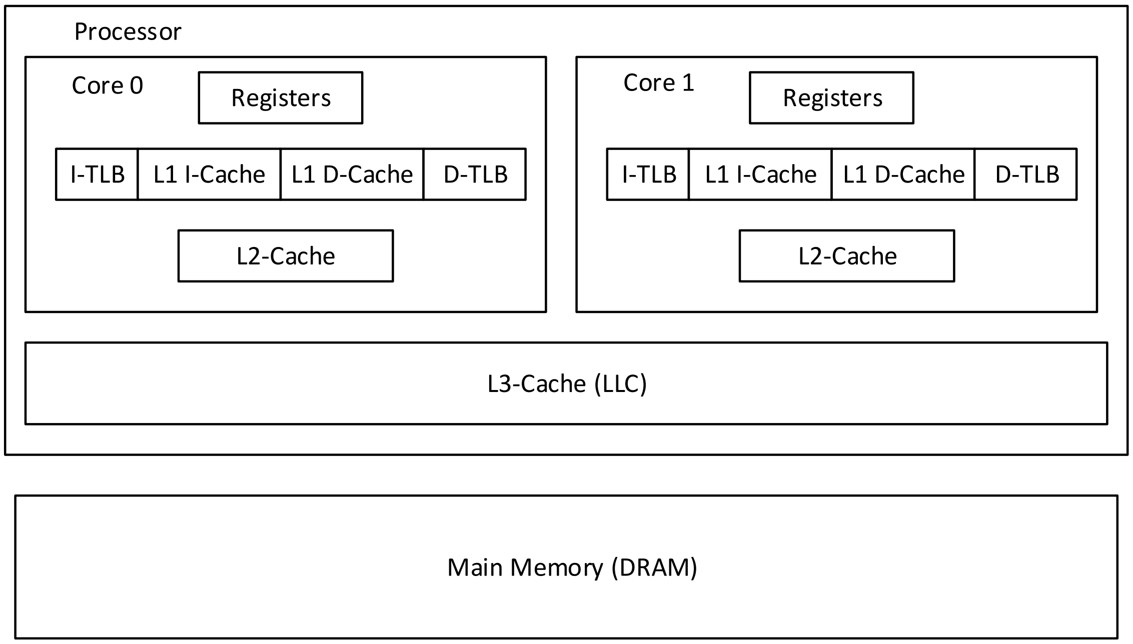

Figure 1.3 High-Level Memory Organization in a Multi-core Processor

of these architectural registers in order to improve performance through register renaming, allowing parallel and speculative execution. Also, most frequently used data is held in registers to improve performance. The number of registers available on a processor and the operations that can be performed using those registers has a significant impact on the efficiency of code generated by optimizing compilers. The widely used Intel x86 processors have 8 General Purpose Registers in 32-bit mode and 16 registers in 64-bit mode. The Intel Xeon Phi processor has 16 registers, while the Intel Itanium processor has 128 registers.

There are different kinds of registers - User Accessible and Internal, depending on the content stored in them or the instructions that operate on them. User accessible registers can be read or written by machine instructions. They are classified into Data registers, Address registers, General purpose registers (GPRs), Conditional registers, Floating point registers (FPRs), Constant registers, Vector registers, Special purpose registers (SPRs), Model-specific registers and Memory Type Range Registers (MTRRs). Internal registers are classified into Instruction registers and Registers related to fetching information from RAM. The RAM registers are further classified into Memory buffer registers (MBR), Memory Data Registers (MDR) and Memory Address Registers (MAR).

1.4.4.2 Caches

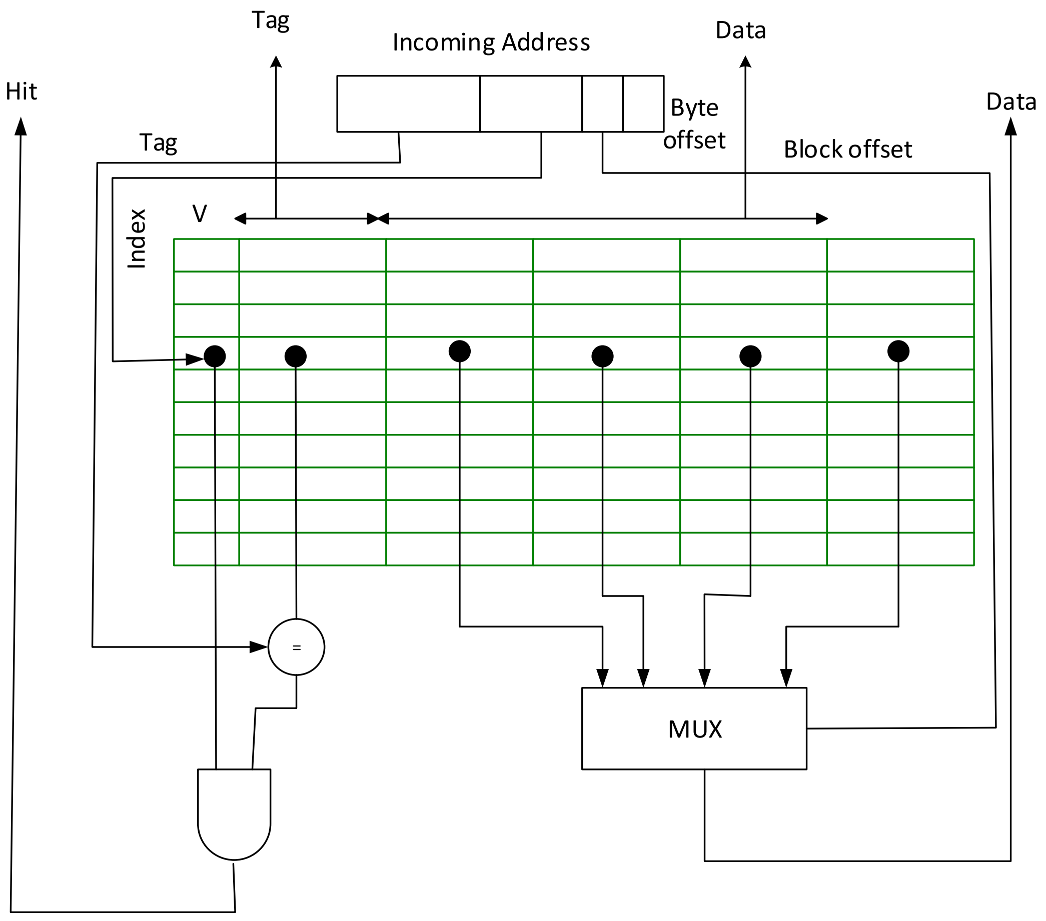

Figure 1.4 High-Level Cache Architecture

Caches are used by a processor to reduce the average time to access data from the main memory. The cache is a smaller, faster memory which stores copies of the data from frequently used main memory locations. Most processors have independent instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels. Data is transferred between the main memory and cache in blocks of fixed size, called cache lines. A cache entry is created when a cache line is copied from main memory. Each cache entry will have the copied data as well as a memory location tag. Whenever a cache hit occurs, the processor immediately reads or writes data in the cache line. Whenever a cache miss occurs, the cache allocates a new entry and copies in data from the main memory, before fulfilling the request. On a cache miss, the Least-Recently Used (LRU) heuristic is commonly used to evict an existing cache entry to make room for a new entry. A set of cache entries are grouped together to form cache ways, which decide the Associativity of the cache. Caches follow different write policies to determine when data is written to main memory from the cache. In a write-through cache, each write to the cache causes a write to main memory. In a write-back cache, writes are not immediately passed over to main memory, and are marked as dirty. The data in these locations is written back to main memory only when it is evicted from the cache. In the case of multiprocessor systems, some cache coherence protocols are used to avoid stale data from being maintained in caches associated with different processors.

The high-level architecture of a cache is very similar to that shown in Figure 1.4. During a cache access, an index is used to find an entry in the cache's data store, and then the tags for the cache line found are compared.

The Associativity of a cache level is also used along with the replacement policy in deciding the cache location where a main memory entry would be stored. The cache is Fully Associative if the replacement policy is free to choose any entry in the cache to hold the copy. The cache is Directly Mapped if each entry from the main memory can go in just one place in the cache. An N-way Set Associative cache is a compromise in which each entry from main memory can go to any of N places in the cache. Associativity increases beyond four-way have much less effect on the hit rate.

Multiple levels of caches are needed, because, larger caches have better hit rates but longer latency. Multi-level caches usually operate by checking the fastest L1 cache first. It that misses, the next fastest L2 cache is checked, and so on, before external memory is checked.

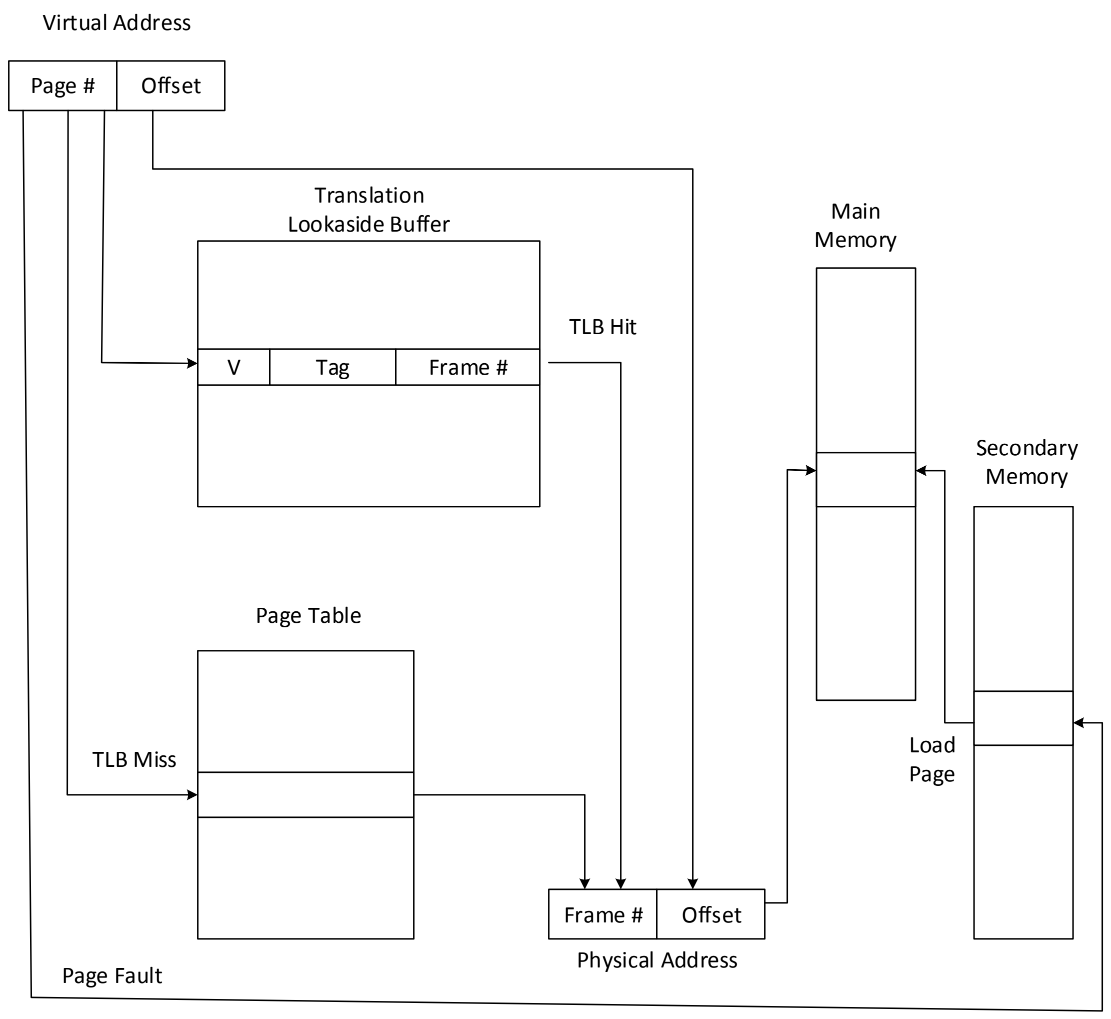

Figure 1.5 High-Level TLB Architecture

1.4.4.3 Translation Lookaside Buffer (TLB)

The Translation Lookaside Buffer (TLB) is a cache that is used to improve the virtual address translation speed. The TLB is frequently implemented as Content Addressable Memory (CAM), where the search key is the virtual address and the search result is a physical address. A TLB hit occurs when the requested address is present in the TLB, and the retrieved physical address can be used to access memory. A TLB miss occurs when the requested address is not in the TLB, and the translation proceeds by looking up a page table in a process called Page Walk. The Page Walk process involves computing the physical address, and then entering the virtual to physical address mapping in the TLB. The Page Table keeps track of where the virtual pages are stored in physical memory. The TLB is a cache of the Page Table and only a subset of the Page Table contents are held there. In a Harvard Architecture, separate virtual address spaces may exist for both instructions and data. This leads to the need for a distinct Instruction Translation Lookaside Buffer (ITLB) and a Data Translation Lookaside Buffer (DTLB). Similar to caches, TLBs may also have multiple levels.

Figure 1.5 shows the architecture of a common TLB.

1.4.4.4 Main Memory(DRAM)

Main Memory is directly or indirectly connected to the Central Processing Unit through a pair of memory buses - an address bus and a data bus. The CPU sends a memory address through an address bus and then reads or writes data in memory through the data bus. A Memory Management Unit (MMU) recalculates the actual memory address to provide an abstraction of virtual memory. The DRAM stores each bit of data in a separate capacitor within an integrated circuit. DRAM consumes relatively large amounts of power and has long access times compared to registers and caches.

1.5 Thesis Organization

The remainder of this thesis builds upon the core ideas introduced in this chapter. Chapter 2 highlights some related work to deal with some of the problems discussed here. Chapter 3 provides more details on the Design and Implementation of our scheme and also features an in-depth discussion on the various phases involved in this work. Chapter 4 showcases the performance evaluation framework and some results obtained while using scheme. Chapter 5 summarizes the benefits observed and some future work in this direction.

CHAPTER 2. REVIEW OF LITERATURE

This chapter presents some related work targeted towards dealing with some of the problems introduced in the previous chapter.

2.1 Introduction

Our methodology differs from prefetching based techniques in the following way. Prefetching populates the cache with unnecessary data for large structures in which only few fields are accessed repeatedly. This is expensive in both performance and energy. Bandwidth between the main memory and cache has become a very critical resource for multi-core computing architectures. The DDVA structure which can also be cached reduces the strain on memory bandwidth by only tracking data structure fields which are used. Prefetching can be used in conjunction with DDVA to trigger fetches to DDVA structures rather than the original data structure. We investigate the performance and energy advantage of DDVA in this work for the SPEC 2006 suite. The next few sections introduce some related work, which target objectives similar to ours.

2.2 Related Work

Our Data Shaping approach seeks to reduce execution time and energy by dynamically emitting spatially-adjacent data from runtime data stores. Earlier work focused on overlapping data access with computation by introducing newer software-based cache designs for non-blocking, prefetching, identifying and storing frequent instructions, and also for managing spatial and temporal locality through independent parts. Some techniques were aimed at optimizing specific data structures in pointer-based recursive applications and in those with array references.

Code transformations, runtime data and iteration reordering-transformations and some interleaving schemes to reduce DRAM row-buffer conflicts were also targeted in other related work. Even though some of these approaches may be successful in reducing memory access latency and execution time, they fail to address the relatively high energy use of these applications.

The following sub-sections discuss some selected related work in an elaborate manner.

2.2.1 Non-blocking and Prefetching Caches

Non-blocking caches and prefetching caches[7] are two techniques for hiding memory latency by exploiting the overlap of processor computations with data accesses. A non-blocking cache allows execution to proceed concurrently with cache misses as long as dependency constraints are observed, thus exploiting post-miss operations. A prefetching cache generates prefetch requests to bring data in the cache before it is actually needed, thus allowing overlap with pre-miss computations. There are also some hybrid approaches that combine the benefits of both these schemes.

2.2.2 Array Cache

A software driven cache design, called the Array Cache[9] was also proposed . It uses a separate cache space to store and handle array references with constant strides that are prefetched accurately with the help of the compiler and with extremely low runtime overhead. This design was primarily targeted towards scientific computation applications, where most of the data references are array references with constant strides.

2.2.3 Register Preloading

Register preloading[8], incorporates a set of hardware and software techniques to effectively tolerate long first level memory access latency. The techniques include speculative execution, loop unrolling, dynamic memory disambiguation, and strip-mining. This approach claims to provide excellent tolerance to first level memory access latency up to 16 cycles for an issue in a 4 node processor.

2.2.4 Code Transformation & Compiler-based Solutions

Another work, as described in [25] proposes code transformations to increase parallelism in the memory system by overlapping multiple read misses within the same instruction window, while preserving cache locality. This approach claims to deliver execution time reductions averaging 20% in a multiprocessor and 30% in a uniprocessor due to significant increases in memory parallelism. A runtime approach to improve computation and data locality in irregular programs based on the inspector-executor method used by Saltz has been proposed in [13]. This work improves computation and data locality, and also eliminates most of the runtime overhead. A compile-time framework that allows run-time data and iteration reordering transformations has been proposed in [29] to enhance locality in applications with sparse data structures.

2.2.5 Pointer-based Prefetching

A software controlled prefetching scheme targeted towards pointer-based applications with recursive data structures has been been proposed in [22]. This method claims to help achieve a 45% improvement in execution time. A HotSpot instruction cache has been proposed in [30] that identifies frequently accessed instructions dynamically and stores them in the smaller L0 cache. This approach helps achieve a 52% reduction in instruction cache energy without performance degradation.

2.2.6 Dual Data Cache

A new cache organization, called the Dual Data Cache has been proposed in [14], with independent parts for managing spatial and temporal locality. This work also implements a lazy caching scheme, to cache data only when benefits are realized. It also maintains a locality prediction table, with information about the most recently executed load/store instructions.

2.2.7 Zero Cycle Load

A hardware assisted mechanism that reduces the latency of load instructions has been provided in [2]. This approach, called zero cycle load, helps complete load instructions upto two cycles earlier than traditional pipeline designs. So a result is produced prior to reaching the execute stage of the pipeline, allowing subsequent dependent instructions to proceed unfettered by load dependencies.

2.2.8 Prediction-based Prefetching

A predictive approach has been employed to reduce File System Latency in [16]. This method uses past file accesses to predict future file system requests and prefetches data prior to the request for data, masking access latencies. This method claims to deliver a 280% improvement in access latency over LRU and also a 50% reduction in cache size.

2.2.9 Integrated Prefetch & Cache Memory Controllers

Another implementation in [19] integrates a prefetch unit with the L2 cache and memory controllers to address the issue of slow DRAM accesses. It issues prefetch requests only when the channels are idle, prioritizes them to maximize DRAM row buffer hits and gives them low replacement priority. This method helps achieve an average of 43% speedup.

2.2.10 Page Interleaving Schemes

DRAM row-buffer conflicts are another important reason leading to a high memory access latency. A permutation-based page interleaving scheme that reduces row-buffer conflicts and exploits data access locality in the row-buffer has been proposed in [31]. This approach helps reduce the memory stall times 68% and 50% compared with conventional cache line and page interleaving schemes, respectively.

CHAPTER 3. DESIGN AND IMPLEMENTATION

This chapter details the various phases involved in the Design and Implementation of this work. The following sections present an in-depth discussion on Profiling, Architecture, Data Shaping, DBO Framework Integration and the Performance Evaluation Framework.

3.1 Introduction

Our work involves identifying procedures in benchmarks with significant non-spatially adjacent data access and with notable contributions to overall execution time, integrating the HDTrans dynamic binary optimization framework with the targeted applications and building a contiguous dynamic dataview of such non-spatial data. Significant efforts were also channeled towards modeling the performance of three efficient data stores (the Dynamic Data View Array - DDVA, Tagless D-Cache and Scratchpad Memory) and building performance analysis frameworks for measuring execution time and energy for data accesses. These phases are outlined at the high level in Table 3.1.

The later sections describe these phases in more detail.

3.2 Profiling

The preliminary phase of our work involved rigorous efforts towards profiling various applications of the SPEC2006 benchmark suite and identifying procedures or functions with a significant amount of non-spatially adjacent memory access involved in their computation. It was also ensured that the target functions chosen had contributed significantly to the overall execution time of the application, so that noticeable increases in overall performance can be observed by subjecting them to the data shaping process. It was also ensured that gcc-based ap-

Table 3.1 Phases involved in the design and implementation of data shapers

Phase	**Implementation Activities**
Profiling	SPEC2006 benchmarks were profiled to identify procedures with non-spatial data access and those which contribute significantly to the overall execution time.
DBO Framework Integration	The HDTrans Dynamic Binary Optimization framework was integrated with the selected benchmarks to enable basic block creation and program optimization at runtime.
Data Shaping	A Dynamic Data View Array(DDVA) was created to cache frequently accessed non-spatial data to emit spatially adjacent data replacing irregular access patterns at runtime.
Performance Evaluation Framework	An evaluation framework that provides an effective measure of execution time and data access energy for the original and data shaped benchmarks was designed.

plications were chosen, to avoid any possible compatibility issues with the gcc-targeted dynamic binary optimization framework, HDTrans, used in our implementation. gprof[15], a widely used linux-based profiling tool was utilized to profile various applications from the benchmark suite. gprof provides elaborate information on the percentage contributions of various procedures to the overall execution time, along with the individual time that each procedure had run for.

After extensive profiling and analysis for non-spatially adjacent data access, the mcf and h264_ref benchmarks were shortlisted to be targeted for optimization using our data shaping approach. More elaborate information on the chosen benchmarks, along with their targeted procedures, and their contribution to the overall execution time is detailed in Table 3.2.

Table 3.2 SPEC2006 benchmarks profiled for non-spatial data access and execution time

Benchmark	**Target Procedure**	**Overall Execution Time**		**Call Count**
		% Contribution	**Contribution in seconds**	
mcf	primal_bea_mpp	65.39	3.76	118647
h264ref	SATD	4.76	9.49	89478825

3.3 Overall Architecture

The overall architecture used in our implementation in conjunction with Dynamic Binary Optimization is as shown in Figure 3.1. Our implementation evaluates three models, namely, the Dynamic Data View Array (DDVA), Tagless D-Cache and Scratchpad memory. All of these three models ensure that the original architectural storage locations do not change.

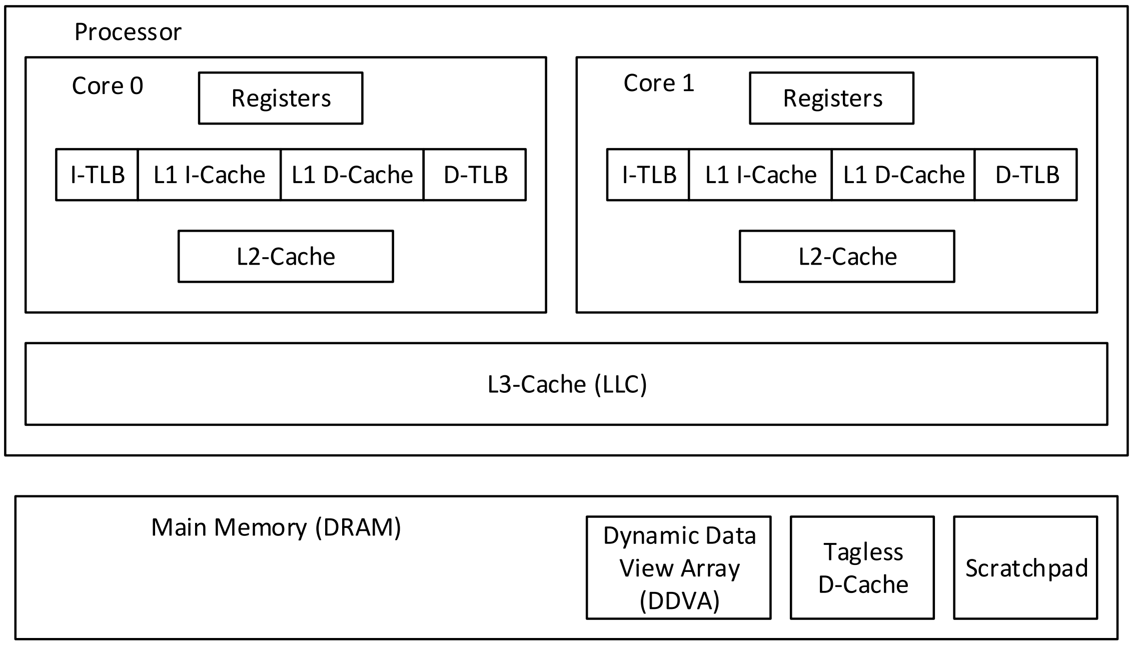

Figure 3.1 Overall Architecture used in the Data Shaper implementation

3.3.1 Dynamic Data View Array

We introduce a new data structure called the Dynamic Data View Array (DDVA), which stores data in a linearly spatial manner. The DDVA caches frequently accessed data views. Addresses in the wrapped block of code are patched to point to DDVA for future access. The index into the DDVA will be stored in the wrapper state when the wrapper engine decides to allocate a DDVA for a wrapped code. The wrapped code block in Figure 3.2 generates data access addresses $A_{i_1}, A_{i_2}, ..., A_{i_N}$ referred to as the algorithmic data view, which may have no spatial locality. A cache line fetched to service a cache miss for address, Aij, may incur

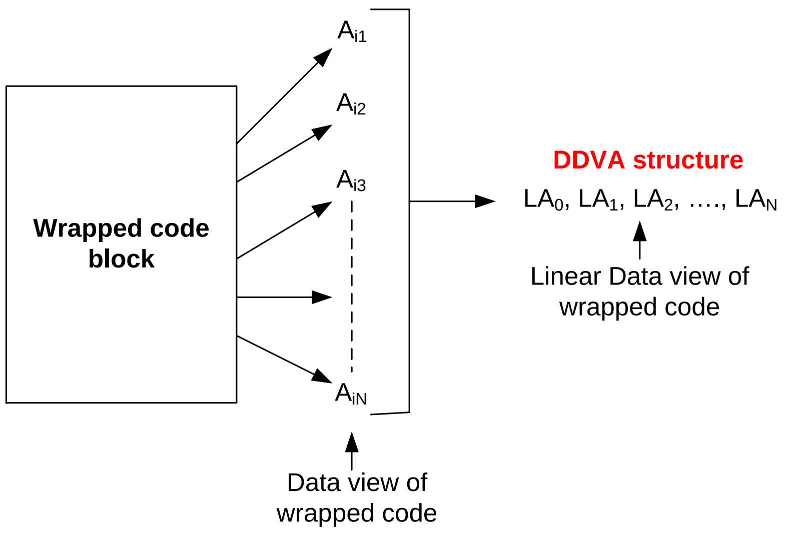
Wrapped code block
A_{i1}
A_{i2}
A_{i3}
A_{iN}
Data view of wrapped code
DDVA structure
LA_0, LA_1, LA_2, …., LA_N
Linear Data view of wrapped code

Figure 3.2 DDVA - Algorithmic data view to Linear data view mapping

energy costs of tag access and also energy cost of wasted data bandwidth since only a fraction of the data in the cache line may be accessed due to lack of spatial locality. The DDVA in figure 3.2 is a mapping from the algorithmic data view to a dynamic linear data view where the addresses $A_{i_1}, A_{i_2}, ..., A_{i_N}$ are mapped to linear addresses $LA_0, LA_1, LA2, ..., LA_N$ using an array in the wrapper code. This implementation models these dynamic data views using linear array accesses to enforce spatial locality in an otherwise poor spatial data access pattern. Data access instructions in the wrapped code are modified to access data from the DDVA and emitted into the basic block code cache. Note that although DDVA is mapped in main memory address space, it can be cached through cache hierarchy levels in the transparent manner.

3.3.2 Tagless D-Cache

Our implementation also models a Tagless D-Cache to serve as a source of spatially adjacent data. This implementation is loosely based on a similar approach targeting instruction fetch from a tagless I-cache, as detailed in [18]. This method sought to deal with spatially non-adjacent instruction references by replacing these with tagless I-cache references. So, a tagless D-cache design can be found to be similarly effective in dealing with non-spatially adjacent data references. Tagless cache design for data reduces cache tag comparison energy by exploiting spatial and temporal locality of accesses. Data access locality at basic block granularity can be profiled and frequently accessed basic blocks are aggregated into specially marked pages. Data in such pages can be accessed with out tag comparison in D-cache, thus reducing energy consumption. This D-cache approach for data accesses reduces the L1 cache access energy significantly.

The Tagless data cache (TDC) models these dynamic data views using linear array accesses to enforce spatial locality in an otherwise poor spatial data access pattern. The wrapper state is modified to include a pointer to index into the TDC as shown in Figure 3.3. Data access instructions in the wrapped code are modified to access data from TDC and emitted into the basic block code cache. Architecturally, some banks of the cache can be flagged to be tagless. The cache controller then knows that the address mapping of the entire bank is guaranteed to contain a single address prefix (tag). The access time of such a bank is no different than the tagged cache access, but it consumes less energy. We maintain a virtual time counter to count the access time of these accesses based on a Cacti reported model.

3.3.3 Scratchpad Memory

Scratchpad Memory(SPM) is a high speed local memory store used for temporary storage and rapid retrieval of data. SPM is similar to the L1 cache, as it is the next closest memory to the ALU after the processor registers. Scratchpads don't contain a copy of data stored in the Main Memory and have Non Uniform Memory access latency. Scratchpads are explicitly manipulated by applications and are employed for simplification of caching logic.

In this implementation, Scratchpad Memory provides quick data access times and also reduces data access energy. Scratchpad sizes of upto 1 kB can be supported in this implementation for data access without overheads. Once again, a virtual counter maintains the access times for all scratchpad data view accesses based on a Cacti derived model. Note that in reality, these stores - tagless cache and scratchpad are maintained within main memory within our DBO environment. However, energy and time for these accesses is modeled as if they were implemented architecturally.

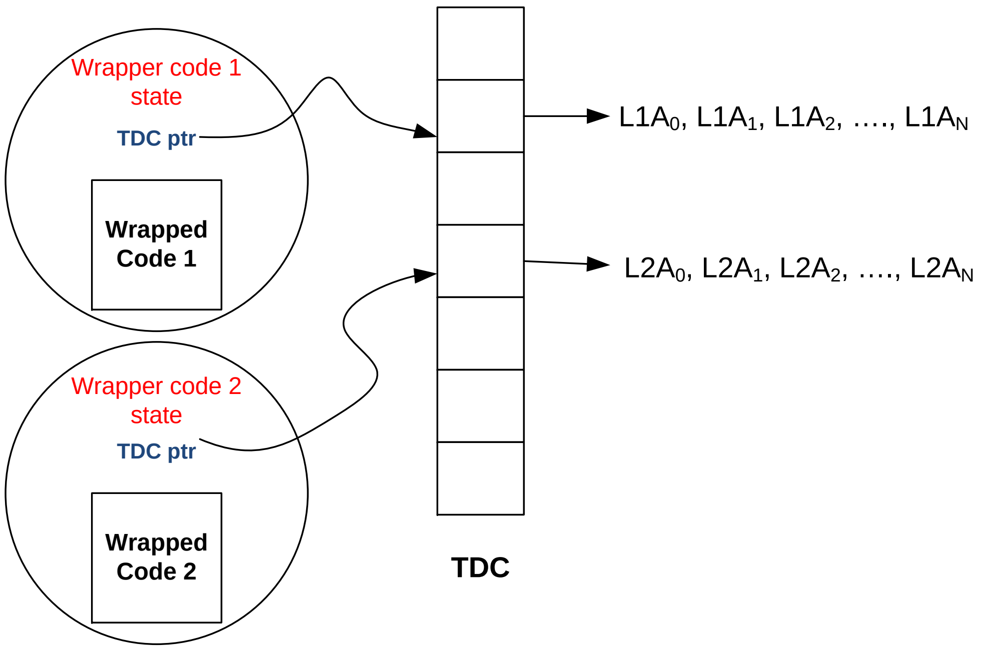

Figure 3.3 Wrapper code state - Pointer into the tagless data cache (TDC)

3.4 Data Shaping

Data Shaping involves replacing frequently accessed non-spatially adjacent data with data from a dynamically-built spatial dataview, the DDVA, at runtime. This is achieved by identifying such non-spatial data access regions in the target procedure and placing special wrapped region function calls around them to make the HDTrans system aware of the data access instructions to be replaced at runtime. A linear spatially-adjacent dataview which copies over the non-spatially accessed data is defined between a set of wrapper region function calls. Frequently, this wrapper region region also incorporates the alternative logic to replace the wrapped region logic with. These are in the form of x86 instruction opcodes to be emitted at runtime, since the code is already in a compiled state when this data access swapping occurs.

3.5 DBO Framework Integration

The HDTrans Dynamic Binary Optimization environment needs to be setup prior to invoking the wrapper and wrapped region system calls. Once invoked, the HDTrans system remains active across multiple runs of the resident function. As discussed in the previous sections, HDTrans builds basic blocks from straight line code fragments that are separated by conditional statements. HDTrans maintains all of these basic blocks inside a Basic Block Cache (bbCache). HDTrans provides APIs to copy over the wrapped code and modify the code, dependent on the user's needs. HDTrans also provides dedicated control-transfer system calls, which transfer control to either the modified wrapped code or to the original unmodified wrapped code, depending on the more efficient flow. Special care needs to be taken while placing absolute jump instructions and system calls inside the wrapper code. This is because, the jump offsets need to be relative to the current position inside the bbCache.

3.6 Performance Evaluation Framework

This work also involved designing multiple performance modeling frameworks, to enable accurate tracking and analysis of time and energy. Using these frameworks, average time and energy data was collected across multiple runs of the mcf and h264_ref benchmarks for multiple sets of inputs. These frameworks are described in more detail in the following sections.

3.6.1 Execution Time Framework

The execution time framework is based on the read_timer system call[26] belonging to the PMU library. It internally uses the rdtsc primitive to get the running count of the number of clock cycles elapsed. The difference in the number of clock cycles was analyzed, both at a wrapped region granularity, and at an application-level granularity. This difference was used to compute execution time in terms of the number of seconds taken for a specific processor frequency.

3.6.2 Energy Framework

The energy framework is based on the Cacti [23] model of computing the energy consumed for accessing data referenced from different types of memory for specific cache attributes such as the Cache size, Block Size and Associativity. Cacti provides an accurate measure of the energy consumed for accessing the tag and data sections of a cache line. Our Data Shaper based implementation significantly reduces the need for tag comparisons. The energy needed for accessing such spatially adjacent data is effectively the same as that taken for accessing the data section of the cache line. The original benchmark, on the other hand, has much higher access energy due to significant contributions by both data and tag accesses.

CHAPTER 4. RESULTS

This chapter furnishes the Execution Time and Access Energy data as observed in the original and Data-Shaped applications using our Performance Evaluation Framework detailed in the previous chapter.

4.1 Introduction

The Performance Evaluation Framework was used to capture the execution time and energy statistics for both the original SPEC2006 benchmarks, as well as those subjected to the Data Shaping process. The mcf and h264_ref benchmarks correspond well to the requirement of having significant non-spatially adjacent data access. The behavior of these benchmarks was studied extensively for varying inputs. The results observed are detailed in the following sections.

4.2 Execution Time

The execution time performance of the mcf and h264_ref benchmarks before and after the data shaping process for input datasets of different sizes is as shown in Figures 4.1 and 4.2 respectively. The data shaping was done using a DDVA-based data store. This execution time was obtained after computing the number of cycles elapsed using the read_timer system call from the PMU library. It can be seen that an average of 20% reduction in execution time was observed.

Execution Time across the three models of the default Dynamic Data View Array (DDVA), Tagless D-Cache and Scratchpad Memory (SPM) was modeled for the h264_ref benchmark for different input dataset sizes as shown in Figure 4.3. This execution time was computed from Cacti for each read access from the various data stores and then modeled for the entire application.

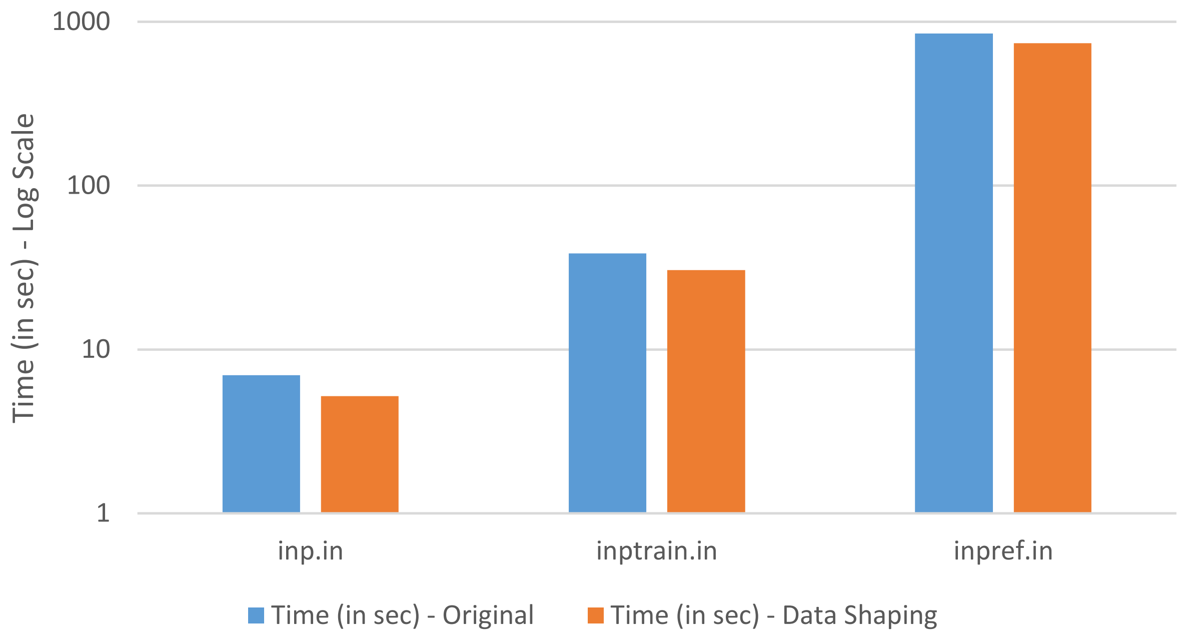

Figure 4.1 Execution Time Performance for input datasets using the original and data shaped versions of the mcf benchmark

4.3 Energy Use

The energy performance of the mcf and h264_ref benchmarks before and after the data shaping process for input datasets of different sizes is as shown in Figures 4.4 and 4.5 respectively. The data shaping was done using a DDVA-based data store. The total dynamic read energy per access was obtained from Cacti and then used for modeling the benchmark's energy performance. It can be seen that an average of 5% reduction in energy was observed.

Access Energy performance across the three models of the default Dynamic Data View Array (DDVA), Tagless D-Cache and Scratchpad Memory (SPM) was modeled. The access energy for the h264_ref benchmark for different input dataset sizes is as shown in Figure 4.6. This overall energy was also computed by using the Cacti read access energy for the various

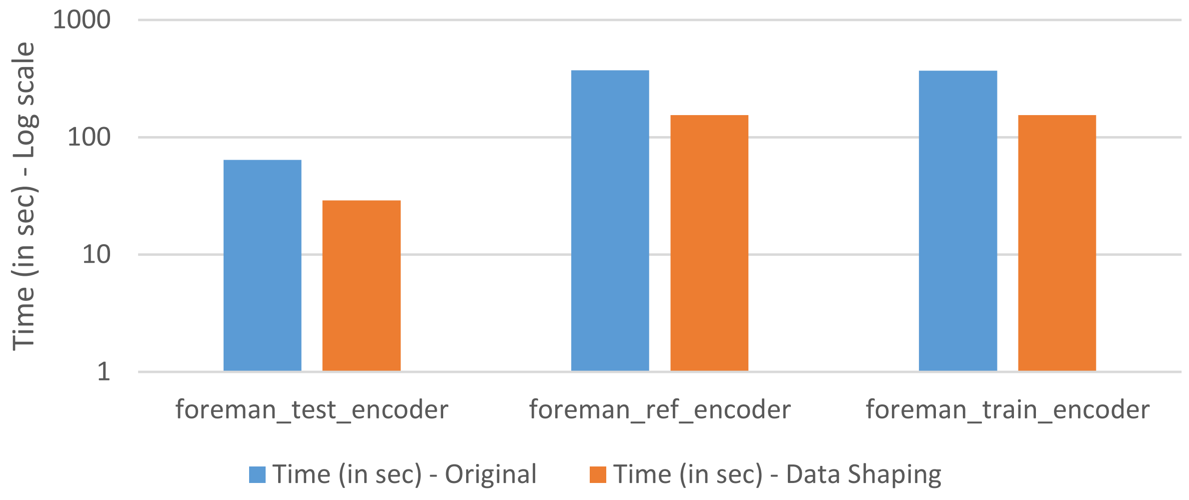

Figure 4.2 Execution Time Performance for input datasets using the original and data shaped versions of the h264_ref benchmark

data stores and then extended to the entire application. It can be observed that the Tagless D-Cache model had the lowest energy consumption when compared to the DDVA and Scratchpad Memory. Elimination of tag comparisons and a lower latency in accessing data are the primary reasons for this reduced energy.

4.4 Evaluation

From Figure 4.1, it can be seen that the Data Shaped version of the mcf benchmark has a significantly lower execution time compared to the original version, which had significant non-spatially adjacent data accesses. This improvement in execution time performance can be attributed to the relatively higher spatial locality introduced by dynamically creating the DDVA data structure and emitting it in place of the irregular data patterns at runtime. Since required data is always fetched as part of a bigger sized block, along with adjacent data, their total resultant access times are lower. It can also be observed that factor of improvement in execution time is much higher for smaller input data sets when compared to larger inputs. This is because the degree of spatial locality possessed by these smaller data sets are much higher.

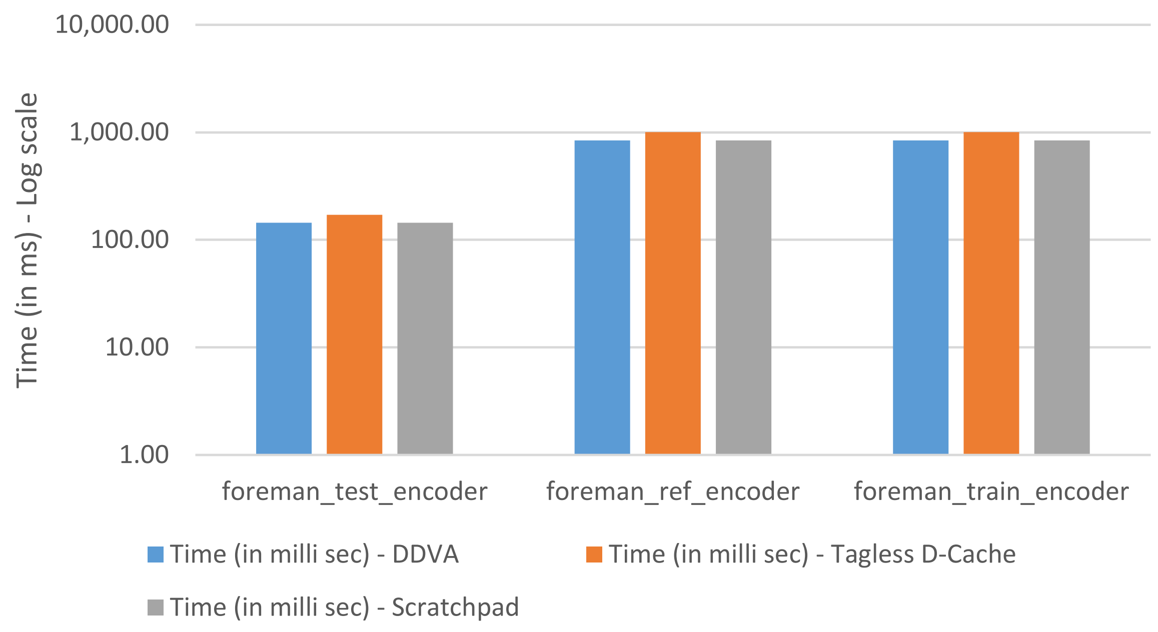

Figure 4.3 Execution Time Modeling for input datasets using the DDVA, Tagless D-Cache and Scratchpad Memory for the h264_ref benchmark

We could make similar observations by analyzing Figure 4.2, which gives the execution time performance of the h264_ref benchmark. It could be observed that the Data Shaped version of the benchmark has led to significant reduction in execution time compared to the original application. Also, the performance improvements are constant across the different input data sizes. This is because, the degree of spatial locality improvements were constant, regardless of the input size.

Figure 4.3 shows the differences in modeled execution time between the three proposed data stores - DDVA, Tagless D-Cache and Scratchpad. It can be observed that the DDVA and Scratchpad have significantly better execution time performance when compared to the Tagless D-Cache, due to their higher degree of spatial locality. These performance benefits can also be observed across the three different input data set sizes.

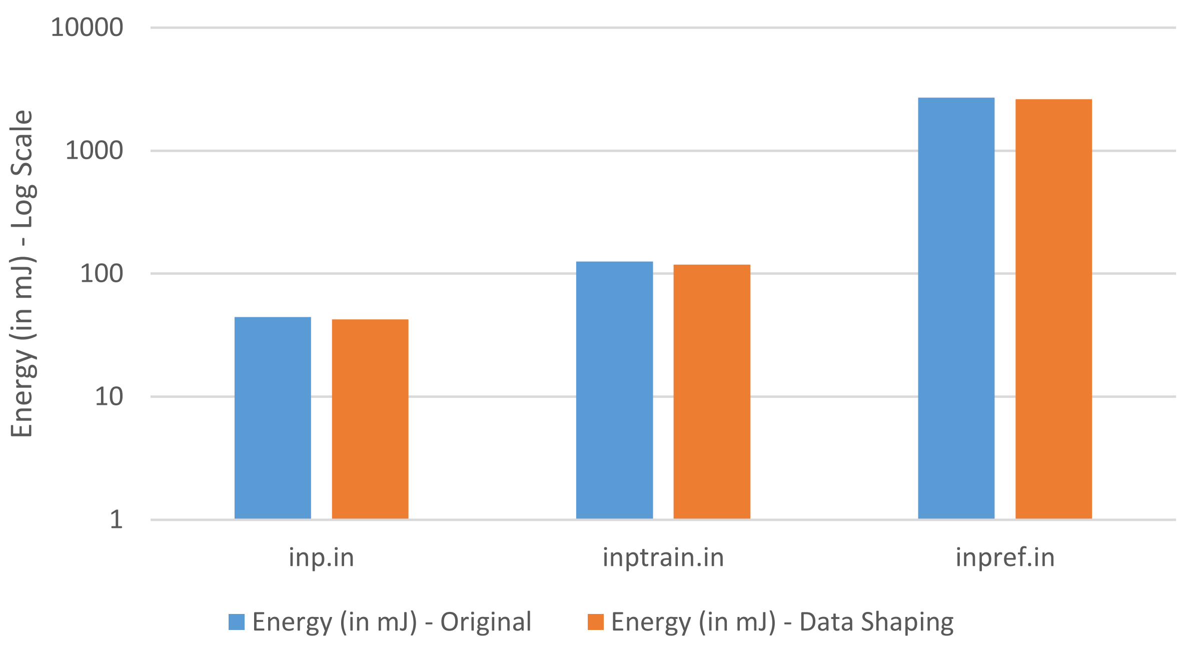

Figure 4.4 Access Energy for input datasets using the original and data shaped versions of the mcf benchmark

The Access Energy Performance of the original and data shaped applications of the mcf benchmark can be studied from Figure 4.4. It can be seen that, a small but significant reduction in energy can be observed in the data shaped application. This is due to the significantly lower cache tag comparisons needed in fetching data. It can also be seen that these benefits are visible across different-sized input datasets as well.

Similar improvements in Access Energy Performance for the Data Shaped Application of the h264_ref benchmark can be seen in Figure 4.5. Access Energy Performance benefits are distributed across different-sized input data sets as well.

Figure 4.6 presents the differences in modeled access energy across the three proposed data stores - DDVA, Tagless D-Cache and Scratchpad. It can be observed that the Tagless D-Cache has the lowest access energy among the three data stores. This can be traced to the total elimination of tag comparisons in the Tagless D-Cache. Thus, the Tagless D-Cache retains its better Access Energy Performance across all sizes of input datasets.

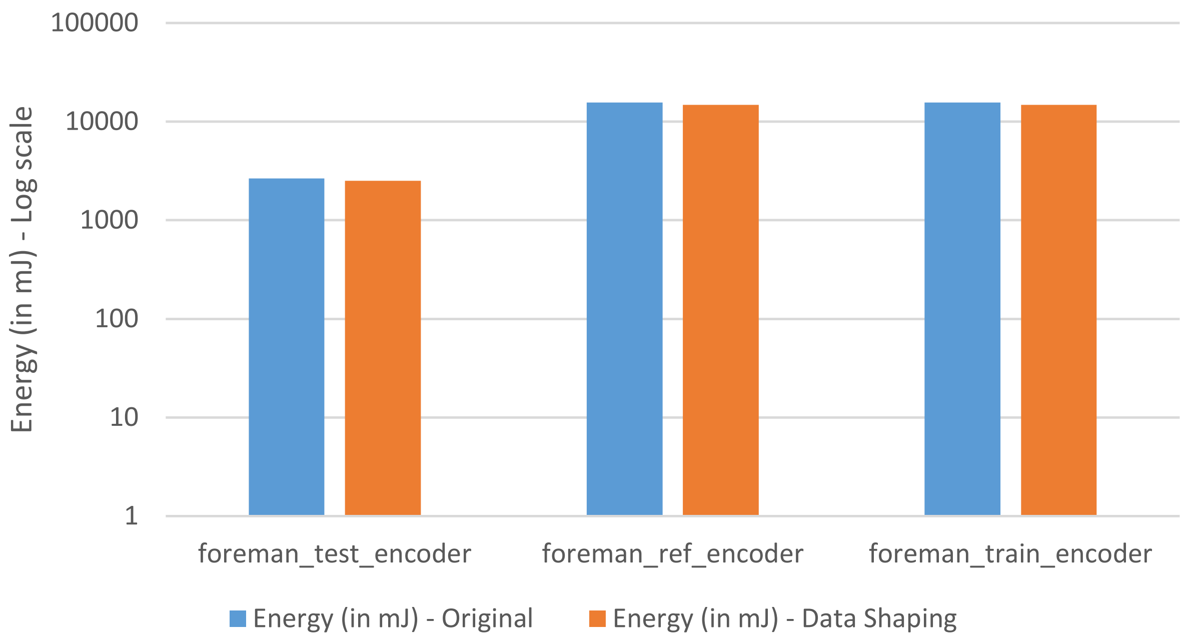

Figure 4.5 Access Energy for input datasets using the original and data shaped versions of the h264_ref benchmark

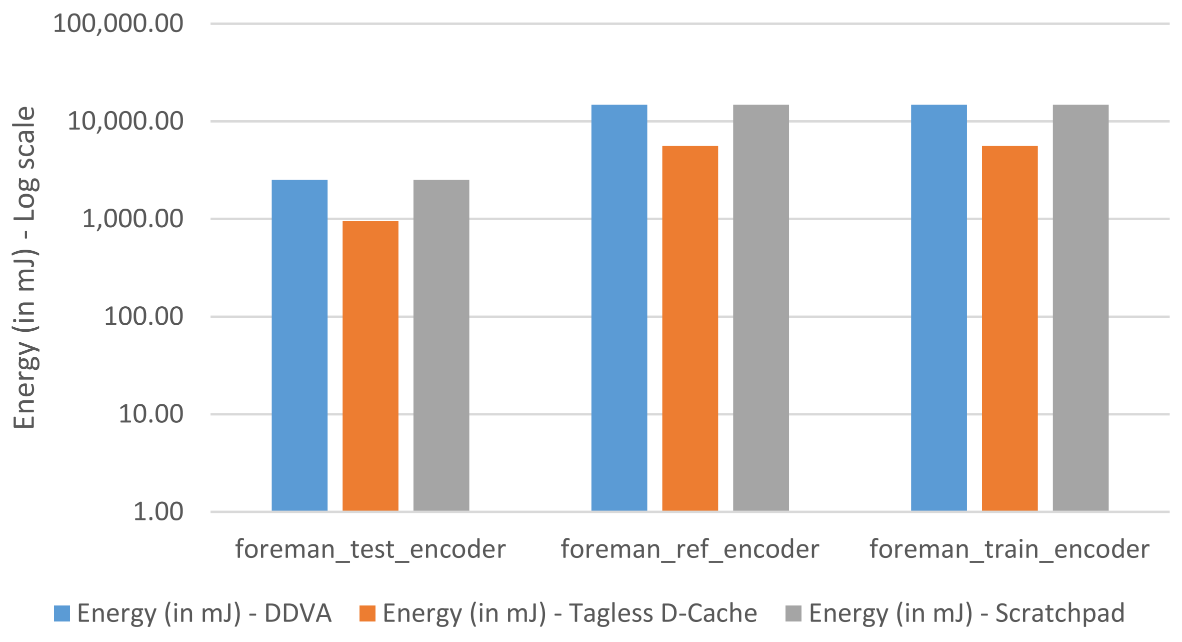

Figure 4.6 Access Energy Modeling for input datasets using the DDVA, Tagless D-Cache and Scratchpad Memory for the h264_ref benchmark

CHAPTER 5. CONCLUSION

This chapter summarizes the targets achieved by this implementation and some possible future work in this direction.

5.1 Summary

This work has demonstrated the effectiveness of Data Shaping by utilizing data stores like the DDVA, Tagless D-Cache and Scratchpad Memory to cache the most frequently used non-spatially adjacent data accesses in a linearly adjacent manner. This process leverages the benefits of the HDTrans Dynamic Binary Optimization framework to perform efficient data shaping at runtime. This work has demonstrated significant reductions in execution time and data access energy in some commonly used SPEC2006 benchmarks. This implementation effectively eliminates the shortcomings of non-spatial data access by replacing such patterns in hotspots of applications with spatially adjacent data from the modeled data stores built at runtime. Execution time improvements by 20% and access energy improvements by 5% illustrate the efficiency of this approach over earlier work. This implementation would be very valuable in scenarios where runtime optimization is needed without adding any additional static overheads.

5.2 Future Work

Future work could involve building a utility to dynamically identify regions of non-spatial access and temporal locality to serve as hotspots for optimization. The scalability of this data shaping process to newer spatially adjacent data stores proposed in research literature could be studied. The positive contributions of Data Shaping towards improving several other system parameters, such as memory bandwidth could be explored. Also, the effectiveness of the HD-Trans framework in supporting performance enhancements in widely used applications, such as the Wikimedia suite, Amazon public data sets and various social networking applications could be studied. Also, improvements to the data collection framework in dynamically capturing various other performance results could be explored.

BIBLIOGRAPHY

[1] Altman, E. R. and Ebcioglu, K. Dynamic binary translation and optimization. *Micro-33*.

[2] Austin, T. M. and Sohi, G. S. Zero-cycle loads: microarchitecture support for reducing load latency. *MICRO 28*, pages 82–92.

[3] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo:a transparent runtime optimization system. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 00)*.

[4] Baraz, L., Devor, T., and Etzion, O. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium-based systems. *36th International Symposium on Microarchitecture*, pages 191–201.

[5] Bellard, F. Qemu: a fast and portable dynamic translator. *Proceedings of the USENIX Annual Technical Conference*, pages 41–46.

[6] Bruening, D., Duesterwald, E., and Amarasinghe, S. Design and implementation of a dynamic optimization framework for windows. *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*.

[7] Chen, T.-F. and Baer, J.-L. Reducing memory latency via non-blocking and prefetching caches. *Technical Report*.

[8] Chen, W. Y., Mahlke, S. A., and Hwu, W.-m. W. Tolerating first level memory access latency in high-performance systems. *21st Annual International Conference on Parallel Processing*.

[9] Chi, C.-H., Ho, C.-S., and Lau, S.-C. Reducing memory latency using a small software driven array cache. *28th Annual Hawaii International Conference on System Sciences.*

[10] Chung, J., Dalton, M., Kannan, H., and Kozyrakis, C. Thread-safe dynamic binary translation using transactional memory. *HPCA.*

[11] Cmelik, B. and Keppel, D. Shade: A fast instruction-set simulator for execution profiling. *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137.

[12] Corporation, S. P. E. https://www.spec.org/cpu2006/.

[13] Ding, C. and Kennedy, K. Improving cache performance in dynamic applications through data and computation reorganization at run time. *PLDI '99*, pages 229–241.

[14] Gonzalez, A., Aliagas, C., and Valero, M. A data cache with multiple caching strategies tuned to different types of locality. *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 217–226.

[15] Graham, S., Kessler, P., and McKusick, M. gprof: A call graph execution profiler. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, 17(6):120–126.

[16] Griffioen, J. and Appleton, R. Reducing file system latency using a predictive approach. *Technical Paper*, pages 197–207.

[17] Hsu, W. C. Dynamic binary translation and optimization. *Technical Presentation.*

[18] Jones, T., Bartolini, S., Maebe, J., and Chanet, D. Link-time optimization for power efficiency in a tagless instruction cache. *CGO, IEEE Computer Society*, pages 32–41.

[19] Lin, W.-f., Reinhardt, S. K., and Burger, D. Reducing dram latencies with an integrated memory hierarchy design. *The Seventh International Symposium on High-Performance Computer Architecture*, pages 301–312.

[20] Lu, J., Chen, H., and Yew, P. Design and implementation of a lightweight dynamic optimization system. *The Journal of Instruction-Level Parallelism.*

[21] Luk, C., Cohn, R., and Muth, R. Pin: building customized program analysis tools with dynamic instrumentation. *PLDI 05*, pages 190–200.

[22] Luk, C.-K. and Mowry, T. C. Compiler-based prefetching for recursive data structures. *ASPLOS VII*, pages 222–233.

[23] Muralimanohar, N., Balasubramonian, R., and Jouppi, N. P. Cacti 6.0: A tool to model large caches. *Technical Report*.

[24] Nethercote, N. Dynamic binary analysis and instrumentation. *Technical Report*.

[25] Pai, V. S. and Adve, S. Code transformations to improve memory parallelism. *Journal of Instruction-Level Parallelism 2*, pages 1–16.

[26] Paoloni, G. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. *Intel White Paper*.

[27] Sridhar, S., Shapiro, J. S., and Bungale, P. P. (2007). Hdtrans: a low-overhead dynamic translator. *SIGARCH Comput. Archit. News*, 35(1):135–140.

[28] Sridhar, S., Shapiro, J. S., Northup, E., and Bungale, P. P. Hdtrans: An open source, low-level dynamic instrumentation system. *VEE2006*.

[29] Strout, M. M., Carter, L., and Ferrante, J. Compile-time composition of run-time data and iteration reorderings. *PLDI 2003*, pages 91–102.

[30] Yang, C.-L. and Lee, C.-H. Hotspot cache: Joint temporal and spatial locality exploitation for i-cache energy reduction. *ISLPED '04*, pages 114–119.

[31] Zhang, Z., Zhu, Z., and Zhang, X. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. *MICRO 33*, pages 32–41.