INTEGRATION the VLSI journal (xxxx) xxxx–xxxx

Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

Shaping data for application performance and energy optimization in dynamic data view framework

Varun Venkatesan[a], Swamy D. Ponpandi[b,*], Akhilesh Tyagi[b]

[a] Intel Corporation, Bellevue, Washington
[b] Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011 USA

A R T I C L E I N F O

Keywords:
Dynamic binary optimization
Data shapers
Spatial locality
Performance optimization
Time and energy

A B S T R A C T

Memory access bottlenecks are often due to the result of mismatch between the processor hardware's view of data and the algorithmic/logical view of data. 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. DDV (Dynamic data view), a runtime dynamic binary translation and 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 cache access energy. Some of the commonly used benchmarks from the SPEC 2006 and SPLASH-2 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 for SPEC 2006 and 11% reduction in time was achieved along with a 6% reduction in energy for SPLASH-2.

1. Introduction

Application data access pattern influences design parameters of micro-architecture such as cache size, number of levels in cache hierarchy, data path. Data access pattern can vary widely depending on how data is manipulated and hence, depends on nature of the application. Non-spatial data access has been a leading contributor to memory access latency in most applications, resulting in increased program execution times and diminished response times. Potential performance (latency) issues of memory hierarchy can often be traced back to linear view of address space. Hardware implementation tends to be much simpler for a linear layout of address space. It is for this reason that most commercial processors have a spatially linear view of data. Applications, on the other hand, extensively use optimal algorithms, tailor-made for program efficiency. This results in a spatially non-adjacent view (logical view) of data manipulated by 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 increased memory access latency, increased program execution times, lower effective memory bandwidth and increased power consumption for data accessed from memory hierarchy. This performance degradation is more pronounced in emerging big data applications, due to a magnified mismatch resulting from the highly non-spatial algorithmic data views.

The primary contribution of this work is the development of an effective approach to reduce the data access time in applications that have a logical view of data which markedly deviates from a linear data view. The negative performance impact of non-spatial data access has been curtailed by creating a *Dynamic Data View* to replace the irregular data access patterns at runtime. *Dynamic Data View* collates irregular data access pattern to provide a linear data view for the processor micro-architecture. Hence, spatially adjacent data presented by *Dynamic Data View* at run time improves the performance of applications by reducing memory access bottlenecks. The storage structures which support *Dynamic Data View* are evaluated in this work for potential improvement in performance and cache access energy. This study evaluates the performance of three data stores that host the dynamically shaped data – the ***Dynamic Data View Array***

* Corresponding author.
E-mail addresses: varunkumhar@gmail.com (V. Venkatesan), swamy@iastate.edu (S.D. Ponpandi), tyagi@iastate.edu (A. Tyagi).

http://dx.doi.org/10.1016/j.vlsi.2016.12.001

(*DDVA*), Tagless D-Cache and Scratchpad Memory. The applicability of this scheme to various applications in the SPEC2006 and SPLASH-2 benchmark suite [1,2] is studied.

Traditional cache solutions fetch blocks of data (cache line) hoping for spatial locality in the immediate vicinity of current miss address. Prefetch based techniques also initiate cache line fills to reduce the probability of a miss by attempting to align the data access window with the prefetch widow. However, the key idea is still spatial locality around the vicinity of the current miss address. Violation of this assumption renders the surrounding data near the miss address useless and results in poor performance/energy for applications with data access patterns that result in such violation. Hence, traditional cache solutions do not address this problem.

DDV is a run time solution to address the diminished utilization of space in cache from the perspective of spatial locality. Run time data access monitoring code (referred to as wrappers in this paper) can be inserted into code regions for shaping data access. There are no restrictions on the data structure which is used to shape data, provided, spatial locality is satisfied. Dynamic data view Array structure mimics a cache line to shape data access, which is the example used in our work. Data shaped and collected in DDVA structure is the actual access pattern of an application which is guaranteed to have spatial locality for a given epoch. Once the data has been shaped for spatial locality within DDVA, all future accesses to data are redirected to the DDVA structure instead of the actual address locations in the original code. The cached version of the DDVA structure now has complete spatial locality. Note that the reshaped data set in DDVA should also have temporal locality through repeated use in order to amortize the cost of shaping over the future accesses. This technique can be seamlessly added to dynamic translation based execution frameworks such as Java virtual machine, hdtrans (x86 dynamic execution framework, used in our work) etc.

2. Related work

The processor memory gap has been increasing at an ever faster rate over the past decade. This gap has further widened due to the inability of architectural and software paradigms to achieve maximum possible memory bandwidth. Previous work to reduce this performance gap has focused on architectural modifications and software techniques to improve application performance by reducing memory bottlenecks. Our proposed approach seeks to reduce both execution time and energy by dynamically collating runtime data into spatially adjacent data.

Prior works have mainly focused on overlapping data access with computation to hide memory access latency due to non-spatial access pattern. Prefetching instructions and data is one of the most researched techniques to hide memory latency [3]. Simplified versions are commonly implemented in compilers and micro-architecture. Prefetch instructions can be inserted in application code through compiler based static analysis. Such instructions trigger prefetch of data before actual consumption of data. Software-based cache designs also have been proposed for identifying and storing frequently reused instructions. Similar techniques can also be used for managing spatial and temporal locality by caching data before consumption. Micro-architecture based prefetch uses history based windows to trigger data or instruction prefetch. Prefetching can be fine tuned to optimize pointer-based recursive applications by targeting specific data structures and array references. Code transformations, iteration reordering-transformations to mask memory latency and some interleaving schemes to reduce DRAM row-buffer conflicts were also targeted in other related work. Even though several such approaches have been moderately successful in reducing and hiding memory access latency and thus, execution time, they have fallen short in addressing the energy use of these applications with non-spatial data access.

Non-blocking caches and prefetching caches [4] 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.

A software driven cache design, called the Array Cache [5] 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 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.

Another work, as described in [6] 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 [7]. 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 [8] to enhance locality in applications with sparse data structures.

A software controlled prefetching scheme targeted towards pointer-based applications with recursive data structures has been proposed in [9]. This method claims to help achieve a 45% improvement in execution time. A HotSpot instruction cache has been proposed in [10] 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.

The authors of [11] propose a prefetching scheme which uses history based information on cache misses to predict cache miss in new pages. This scheme targets irregular data access pattern. They use multiple prediction tables of variable length history information to predict cache misses. More history generally results in better prediction. This technique is targeted primarily at performance of applications and requires changes to hardware architecture. Similarly, the work in [12] uses a stream buffer hardware structure to map irregular data accesses to linear addresses in the stream buffer. They use several hardware structures for training the prefetch engine for future prediction of access patterns. Both works [11,12] report performance improvement for SPEC 2006 benchmarks.

Tagless cache designs are targeted at reducing access energy of L1 cache. The authors of [13,14] propose to eliminate tag comparisons for instruction and data cache by modifying TLB structure. TLB contains extra information about cache ways with valid cache lines that are tagless through exclusive mapping of a page or address space to a way. Such modifications augment new hardware structures to support this tagless access design. Such cache design reduces energy consumption on hit and miss due to the fact that tag comparisons are eliminated. TLB contains the cache way information thus eliminating the need to fetch all ways during test for a hit. The authors of [13,14] report 60% average dynamic energy savings using TLB modifications.

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 is cached during execution of application, 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