# Memory Latency: to Tolerate or to Reduce?

Amol Bakshi, Jean-Luc Gaudiot, Wen-Yen Lin, Manil Makhija,
Viktor K. Prasanna, Wonwoo Ro, and Chulho Shin*

Department of Electrical Engineering -Systems
University of Southern California
Los Angeles, CA 90089-2563
{abakshi, gaudiot, wenyenl, makhija, prasanna, wro, chulhosh}@usc.edu

*Abstract*—

**It has become a truism that the gap between processor speed and memory access latency is continuing to increase at a rapid rate. This paper presents some of the architecture strategies which are used to bridge this gap. They are mostly of two kinds: memory latency reducing approaches such as employed in caches and HiDISC (Hierarchical Decoupled Architecture) or memory latency tolerating schemes such as SMT (Simultaneous Multithreading) or ISSC (I-structure software cache). Yet a third technique reduces the latency by integrating on the same chip processor and DRAM. Finally, algorithmic techniques to improve cache utilization and reduce average memory access latency for traditional cache architectures are discussed.**

*Keywords*— **Memory Access Latency, Simultaneous Multithreading, Decoupled Architecture, Memory Bandwidth, and Processing in Memory.**

## I. Introduction

The speed mismatch between processor and main memory has been a major performance bottleneck in modern computer systems. Processor performance has been increasing at a rate of 60% per year during the last ten years. However, main memory access time has been improving at a rate of only 10% per year [WIL 00]. The memory access latency is thus reaching a hundred or more CPU cycles, thereby reducing opportunities to deliver ILP [HAL 00] causing the *Memory Wall* problem [SAU 96].

Current high performance processors allocate large amounts of on-chip cache to alleviate the memory wall problem. These efforts work efficiently only for applications with high locality. Without sufficient locality, applications such as multi-media processing, text searching or any other data intensive computation suffer [HON 99]. Much work has been conducted to tolerate or reduce this memory latency. In this paper, some of the most commonly encountered memory latency reduction technologies are reviewed and compared.

Multithreading is based on the idea of exploiting thread-level parallelism to tolerate memory latency [KWA 99]. Simultaneous Multithreading (SMT) combines the multiple instruction issue paradigm with the memory latency hiding technique of multithreading [TUL 95]. This architecture can solve the potential problems of wide issue superscalar architectures. If there is a cache miss caused by a memory access instruction in a thread, then instead of having idle cycles in a superscalar processor, an SMT processor can find alternative instructions from another thread. Moreover, an SMT architecture attempts to overcome low resource utilization by having additional threads available for issue. The detailed characteristic of SMT architectures will be discussed.

Multithreaded architectures have also been proposed as a means to overlap computation and communication in distributed memory system. In other words, the remote memory accesses latency can be tolerated by utilizing multithreaded architecture. By switching to the execution of other ready threads, the remote memory access latency can be hidden from useful computations as long as there is enough parallelism in an application. To reduce the remote memory latency in distributed memory system utilizing multithread architectures, the I-structure software cache (ISSC) system has been developed [LIN 98]. Section III describe the detailed technology of multithreaded architectures and ISSC.

A Decoupled Architecture separates the memory access function from the normal processor operations [KUR 94]. A Memory access processor performs data pre-fetching ahead of the computing processor and provides less memory access latency. In this paper, the HiDISC, a novel decoupled architecture, is used as a case study of this approach. Computing instructions, memory access instructions and cache management instructions are partitioned and fetched into each of the processor levels.

The other approaches attempt to improve both the memory bandwidth and the latency using the advanced VLSI technology in memory manufacture. A significant changes of DRAM architecture consists in attacking the height of the memory wall with enhanced technology. As indicated in [BUR 96], the memory stall cycles can be seen as either raw memory latency stall cycles or limited bandwidth stall cycles. Most current DRAMs such as Synchronous RAM, Enhanced Synchronous RAM and Rambus DRAM have reduced the stall cycle by widening the memory bandwidth [CUP 99]. Since prevailing latency-hiding techniques such as prefetching, speculative loads and multithreading increase the memory traffic, increasing the bandwidth offers an effective solu-

*This paper is based upon work supported in part by DARPA grant #F30602-98-2-0180 and #F33615-99-1-1483

tion for those architectures.

Another important effort towards the enhancement of the memory performance consists in moving the memory close to the processor. Advanced VLSI technology enables the integration of processor and memory on a single chip. This approach, referred to as Intelligent Memory (IRAM) or Processing in Memory (PIM) [KAN 99], dramatically reduces the memory access latency as well as the memory bandwidth.

Finally, a great deal of research has focused on cache-conscious program control and data transformations that create greater spatial and/or temporal locality of access to the application's data set. Conventional cache architectures are based on principles of spatial and temporal locality, which are not true for a large class of applications. By using cache-conscious data layouts and control transformations, such applications benefit from greater cache utilization, leading to a reduced demand on memory bandwidth and reduced access latency. Section VI discusses cache-conscious data and control transformations in greater detail.

## II. Simultaneous Multithreading

### A. Background

Increasing die capacity (approximately a factor of 4 on an average of 3 years) has continued creating opportunities for processor designers. Commodity RISC microprocessors pack more and more power by using the increased wafer density, replicating critical components and exploiting Instruction-Level Parallelism. Some recent processors replicate almost entirely their execution pipeline. However, the success of such architectures has been limited to CPIs (Clocks Per Instructions) slightly less than 1 for some of the most modern 4-issue superscalar architectures such as the PowerPC 620 [DIE 95]. The reason can be traced to the limited Instruction-Level Parallelism in some important benchmark applications and to the exponential complexity of the hardware needed to deliver more runtime parallelism. Indeed, "conventional" approaches suffer from some significant drawbacks:

- *VLIW* machines are known to have several inherent disadvantages, such as the binary compatibility problem, the inability to dynamically detect and utilize runtime parallelism, the increase in code size for speculative execution support, and the performance degradation due to asynchronous events such as cache misses. However, VLIW approaches have recently seen a revival mainly because of their simplicity in hardware. Hardware simplicity implies low power dissipation. For certain sets of applications, VLIW can be a killer architecture that allows high performance and low power consumption at the same time.
- *Superscalar* architectures rely on run-time (hardware) dependency checking which gets exponentially more difficult as the size of the instruction window increases (which is required in order to deliver higher performance and to feed more pipelines). Further, it can be easily predicted that in the next few years, processor manufacturers will be able to fabricate even wider-pipeline superscalar processors such as 8-issue or even 12-issue. However, recent simulation studies have shown that widening the pipeline alone would not significantly improve the performance.
- *Superchips* or *on-chip multiprocessors* are another approach to utilize increasing die capacity. The basic principle consists in integrating on a single chip a number of simple scalar processors. However, recent studies have shown that it is not either the best way to achieve high performance. Indeed, superchips show extremely poor utilization when running sequential applications.

### B. General Concept

A newer approach, Simultaneous Multithreading (SMT) or Multithreaded Superscalar architecture [TUL 95][GUL 96] may offer a solution to these problems: instead of extracting Instruction-Level Parallelism (ILP) from a single thread of a program, an SMT processor concurrently executes several instructions from different program threads (by maintaining several PCs and extended or multiple register files). When an instruction from one thread stalls due to a long-latency operation, the thread is suspended and no more instructions of the thread are fetched from the instruction cache whereas instructions from other threads are placed into the pipelines. Although at each cycle instructions from only one thread are fetched on a round-robin basis, the instructions from different threads are executed in the pipeline under the control of the dispatch logic. This is because no fixed allocation between threads and execution units has been made.

Unlike on-chip multiprocessors, even without recompilation of the original code, SMT would guarantee single thread performance by prioritizing the critical program when running a multiprogrammed workload. By virtue of SMT's ability to run multiprogrammed workload, we can obviously get multiprogram speed-up: while a VLIW or a Superscalar processor would be compelled to execute programs one after the other, they can be run by an SMT machine as one "pseudo-job." SMT is the approach that can extract the most fine-grained parallelism from multiple programs.

SMT should offer higher performance than VLIW because, unlike VLIW, independent instruction streams are not bound and stalled together, thus making the control of execution simpler and efficient. Further, in SMT, the overhead of dependency checking in hardware is greatly reduced because it works on multiple independent threads (instructions from a thread are interspersed with instructions from other independent threads, thus lowering, if not eliminating, the overhead

of dependency checking.)

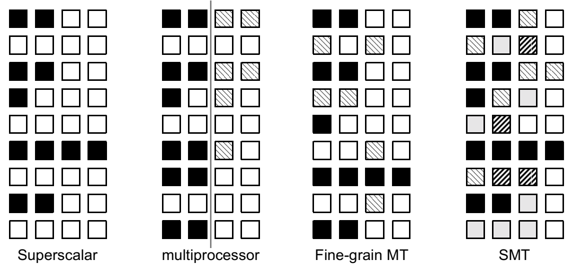


Fig. 1. Issue Pattern in Various Architectures

Fig. 1 shows the issue pattern of SMT and other architectures that compete with it. As shown in the Fig. 1, superscalar architecture has no way to hide the horizontal waste caused by limited instruction-level parallelism. In an on-chip multiprocessor system, even though multiple threads are available, the vertical waste is not hidden because resources are isolated within each processing element. In addition, horizontal waste is not hidden though the waste would not be as significant as in wide-issue superscalar. In fine-grain multithreading, while vertical wastes are effectively eliminated by alternative threads, horizontal waste does not disappear [TUL 95].

Some design issues should be investigated:

- A study of thread partitioning
- Hardware unit tradeoffs
- Usefulness of branch prediction
- Size of caches (both instruction and data cache)

## III. I-Structure Software Cache

### A. General Concept

Non-blocking multithreaded execution models, like TAM [CUL 91], P-RISC [NIK 89], and EARTH [HUM 95], have been proposed to support multithreaded execution in a conventional RISC-based multiprocessor system without the need for any specific hardware support for fast context switching. In these models, remote memory requests are structured as split-phased transactions to tolerate communication latency. Thread activations are data driven in these models: A thread is activated only when all the data elements it needs are available locally. Indeed, once a thread starts to execute, it executes to the end. In such a *non-blocking multithreaded* execution model, once the execution of a thread terminates, no thread contexts need to be saved before switching to the execution of another active thread. Therefore, multithreaded execution is achieved without the need for any specific hardware support.

A good execution model must be based on a good memory system to achieve high system performance [DEN 95] An *I-Structure memory system* [ARV 89] provides split-phase memory accesses to tolerate communication latency. It also provides non-strict data access, which allows each element of a data structure to be accessed once the data element is available without waiting for the whole data structure to be produced. Each element of an I-Structure has a presence bit associated with it to indicate the state of an element, such as *Empty* and *Present*. Data can only be written into empty elements, and the slots are set to the present state after the data has been written into them. A read from an empty element is deferred until the data is produced. The split-phase accesses to the I-Structure elements provide fully asynchronous operations on the I-Structure. The non-strict data access on the I-Structure provides the system with a better chance to exploit fine-grain parallelism. The fully asynchronous operations on the I-Structures make it easier to write a parallel program without worrying about data synchronizations since the data are still synchronized in the I-Structure itself. The single assignment rule of the I-Structure provides a side-effect free memory environment and maintains the determinacy of the programs. All of these features make I-Structures, along with non-blocking multithreading, an ideal model for parallel computing.

While the combination of non-blocking multithreaded execution and I-Structure memory system appears to be a very attractive architecture for high performance computing, the major drawback of this system is that locality of remote data is not utilized. If all remote requests are translated into split-phased transactions and the fetched data is not saved between uses, excessive inter-processor traffic will be generated. The problem is compounded by the cost of sending and receiving network packets. Each packet may take from several dozens to thousands of cycles to traverse on network depending on the design of the network interface and communicating protocols [CUL 93]. Even though multithreaded execution could tolerate the communication latency, processors still need to spend the time to issue the request and retrieve the data from network. In most platforms without dedicated processors to handle the network messages, this communication interface overhead cannot be overlapped with useful computations. In some machines, a parallel environment is built on top of the TCP protocol and the *communication interface overhead* may be as high as hundreds of micro-seconds [KEE 95]. Even with some improved protocols, like Fast Sockets [ROD 97] and Active Messages [VON 92], it still costs 40 to 60 micro-seconds to send a message to the network. For instance, in the multi-packet delivery implementation of Active Messages in the CM-5 machine, it costs 6221 instructions for sending a 1024-word message in the finite sequence [KAR 94]. Since all requests are actually sent to remote hosts through the net-

work, all the sending and receiving requests incur the communication interface overhead and will result in high network traffic.

Therefore, an I-Structure cache system which caches these split-phase transactions in non-blocking multithreaded execution is required to further reduce communication latencies and reduce the network traffic. We designed and implemented an I-Structure Software Cache (ISSC) [LIN 98] to cache the split-phase transactions of global I-Structure data accesses in this architecture. This cache system would provide the ability for communication latency reduction while maintaining the communication latency tolerance ability in this architecture. By caching those split-phase transactions, the ISSC significantly reduces the number of remote requests, and hence, the amount of communication overhead incurred by remote requests is reduced. Our ISSC is a pure software approach to exploit the global data locality in non-blocking multithreaded execution without adding any hardware complexity for further improvement of system performance.

### B. The ISSC Runtime System

The runtime system works as an interface between the user applications and the network interface. A block of memory space is reserved by this run-time system as the software cache space. It filters every remote request and reserves memory space in local memory as a cache of remote data. A remote memory request is sent out to the remote host only if the requested data is not available on the software cache of the local host. Instead of asking for the requested data item only, the whole data block surrounding the requested data item is brought back to the local host and stored in the software cache. Therefore, spatial data locality can also be exploited. The detailed design of ISSC and its implementation are described in [LIN 98], [LIN 00].

In [LIN 00], we implemented ISSC and studied its performance on EARTH-MANNA machine. In this study, we demonstrated a software implementation of I-Structure cache, i.e. ISSC, can deliver performance gains for most distributed memory systems which do not have extremely fast inter-node communications, such as network of workstations [CUL 93], [KEE 95], [ROD 97], [KAR 94].

ISSC caches values obtained through split-phase transactions in the operation of an I-Structure. It also exploit spatial data locality by clustering individual element requests into block. Our experiment results show that the inclusion of ISSC in a parallel system that provides split-phase transactions reduces the number of remote memory requests dramatically and reduces the traffic in the network. The most significant effect to the system performance is the elimination of the large amount of communication interface overhead which is incurred by remote requests.

## IV. HiDISC Architecture

### A. Basic Architecture

Decoupled architectures are based on the idea of exploiting the existing parallelism between memory accessing and pure computations [KUR 94]. These architectures usually consist of two processors. One is the *access processor* which performs address calculations and load/store operations, while the other is the *execute processor* which executes normal calculations and produces results. Several previous decoupled architectures have demonstrated high performance for scientific benchmarks such as Lawrence Livermore Loops [McMA 72].

The HiDISC (Hierarchical Decoupled Instruction stream computer) uses a processor between each level of the memory hierarchy to keep each level of memory supplied with data that the processor above it needs [CRA 97]. Three individual processors for computing, memory access and cache management are combined in this novel high performance decoupled architecture. HiDISC is an architectural technique designed to enhance the efficiency of the cache and deliver better Instruction Level Parallelism by offering better average memory access time. Computing instructions, memory access instructions and cache management instructions are partitioned and fetched into each of the processors. HiDISC is designed to resolve the memory wall problem by instruction-level parallelism. Therefore, this new architecture can provide significant improvement especially in data intensive program such as image recognition and data management.

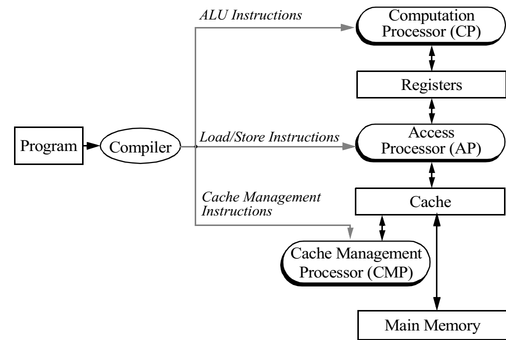


Fig. 2. The HiDISC System

Fig. 2 shows the HiDISC system. A dedicated processor for each level of the memory hierarchy is responsible for passing data to the next level. The *Computation Processor* (CP) is designed to execute all primary computations except for memory access instructions. The *Access Processor* (AP) performs the basic memory access operations such as

load and store. This processor is responsible for passing data from the cache to the CP. The *Cache Management Processor* (CMP) is dedicated to keep the cache ready for the AP operation. By allocating the additional processors to memory hierarchy, the instruction overhead of generating addresses, accessing memory and prefetching is removed from the Computation Processor. The processors are decoupled and work fairly independently of each other. The prefetch distance can adapt to the program runtime behavior. The instruction stream for the each level is supplied by the compiler.

### B. HiDISC Compiler

The HiDISC compiler generates the three HiDISC instruction streams from the sequential assembly code. The HiDISC assembly code and object code format are based on the MIPS format. The HiDISC stream separator is the critical component of the HiDISC compiler. The stream separator works on the assembly code.

The Program Flow Graph (PFG) is generated using the well-known algorithms of [FER 86]. The PFG contains information about control and data dependencies.

The first step is to determine which values are addresses or used in the computation of an address. The control and data flow graphs can be used to trace the registers used as address values or used to compute the address values. After the address values have been determined, the CP and AP instruction streams can be partitioned. Loads, stores, and instructions that compute address values are assigned to the AP instruction stream. Instructions that compute data values are assigned to the CP instruction stream. Conditional branches that use address values are assigned to the AP instruction stream and the EOD token is used to pass results to the CP. Data that is used to compute conditional branch conditions can be transferred from the CP to the AP using the memory system.

The CMP stream is derived from the AP stream since the CMP performs similar operations except that data is prefetched instead of put into a queue and store addresses need not be put into the Store Address Queue. Global data cannot be written by the CMP, and the AP and the CMP must be synchronized whenever the AP writes global data. Greater efficiency can be achieved if synchronization is done after sections where global data is written and not after individual stores.

## V. Memory Technology

Current aggressive DRAM design technologies are attacking the other side of the memory wall - the access latency and the memory bandwidth. In this section, the current leading technologies in DRAM design such as Double Data Rate (DDR) DRAM, Synchronous-Link DRAM (SLDRAM) and Direct Rambus DRAM (DRDRAM) are discussed. Most of new DRAM technologies achieve higher bandwidth with developing new scheme on the interface or access mechanism. This results in a significant enhancement of bandwidth.

However, a recent study shows that the memory latency enhancement is not that significant in new DRAM technology [CUP 99]. Bandwidth alone cannot provide the solution for memory wall problem. Some performance bottleneck is caused by stalling during the DRAM latency for critical data [CUP 99]. One possible solution is integrating memory and processor on a single chip. Intelligent RAM (IRAM) and Processing in Memory (PIM) is the prominent research direction based on the above idea.

### A. Double Data Rate DRAM (DDR DRAM)

Double Data Rate DRAM (DDR DRAM) is the new DRAM technology which can transfer data both on the rising and falling edge of the clock. Therefore, the data rate can be doubled compared to that of SRAM. The specification of DDR2 by the JEDEC (Joint Electronic Device Engineering Council) notes that it will uses a 64 bit wide bus driven at 400Mhz with a new pin interface, and signalling method. This has a potential bandwidth up to 3.2 GBytes/sec.

### B. Synchronous-Link DRAM (SLDRAM)

Synchronous DRAM (SLDRAM) is one of the next-generation DRAM architectures that can deliver the speed and bandwidth requirements of tomorrow's high performance processors. SLDRAM is an open standard which has been developed through the cooperative efforts of leading semiconductor memory manufactures and system designers. SLDRAM builds on the features of SDRAM and DDR RAM. The bus interface is designed to run at 200-600 MHz with a 16 bit-wide datapath.

### C. Direct Rambus DRAM (DRDRAM)

Rambus DRAM (RDRAM) is a memory interleaving system integrated onto a single memory chip. RDRAM supports up to four outstanding requests with pipelined micro architecture and operates at much higher frequencies than conventional SDRAM [HON 99]. In the Rambus architecture, the constant timing relationship between signal pins is ensured. This property makes the Rambus channel run at a much higher frequency than SDRAM [CRI 97]. RDRAM is able to load a new stream of data before the previous stream has completed, resulting in less waiting time and therefore faster access speeds.

Current Direct Rambus (DRDRAM) technology uses a 400 Mhz -2 bytes data and a 1 byte address channel. With transferring at both clock edges, it can achieve 1.6 Gbytes/sec bandwidth from a single DRAM. The Direct Rambus DRAM presents direct control of all row and column resources con-

currently with data transfer operations. DRDRAM partitions the bus into different three parts, three transactions can simultaneously utilize the different portions of the DRDRAM interface [CUP 99].

### D. Intelligent RAM (IRAM)

Advances in VLSI technology increase the amount of DRAM memory on a single chip. As a result, merging both memory and processor on the same chip has been developed for the enhancement of memory bandwidth and latency. This approach has been called Intelligent Memory (IRAM) or Processing in Memory (PIM) [KAN 99]. In the past, only small amounts of DRAM would fit into a single chip with the CPU. Therefore, IRAM was mainly considered as a building block for multiprocessors [PAT 97]. Nowadays, the space occupied by the processor core logic is only about one third of the die. The space occupied by the huge amount of cache can be used for DRAM space. DRAM have 30 to 50 times more capacity than the same chip area for caches [KOZ 97].

All the memory accesses remain within a single chip and the memory bus width is not restricted by pin constraints. As indicated in [PAT 97], the bandwidth and latency can sharply improve in IRAM architectures. The bandwidth can be as high as 100 to 200 Gbytes and the memory latency for random addresses is possibly less than 30ns. The dramatic enhancement of memory performance can provide better solution for data intensive streaming applications. To utilize the huge internal bandwidth of IRAM, a vector architecture can be adopted. A vector IRAM (VIRAM) has indeed been proposed as a cost effective system which incorporates vector processing units and the memory system on a single chip [KOZ 97].

VIRAM's general-purpose vector processor unit provides high performance on computations with fine grain data parallelism [THO 99]. Therefore, this architecture is beneficial for the multimedia application. Vector IRAM provides a number of critical DSP features. It achieves high-speed multiply accumulates through instruction chaining. Auto-increment addressing through strided vector memory accesses is also supported [KOZ 98]. As indicated in [THO 99], the performance of VIRAM processor in computing floating-point FFTs is competitive with that of existing floating point DSPs and specialized fixed-point FFT chips. The advantage of VIRAM over DSP is the programmability as general purpose processor. It also provides a virtual memory system.

One more benefit except the latency and the bandwidth is the energy efficiency of IRAM. Since the frequency of off-chip communication is much less, the energy efficiency can be improved by factors of 2 to 4. Vector IRAM achieves more energy efficiency by reducing the fetching and decoding operations. Vector IRAM uses vector instructions that indicate a large number of independent operation. This energy efficiency characteristic with multimedia application support makes VIRAM attractive as the future mobile computing processor.

One drawback of IRAM is the bound of on-chip memory capacity. Some applications may ask more memory space than the on-chip memory space of an IRAM. Conventional off-chip memory can be used as the next memory hierarchy. The cost and implementation technology also need to be considered for the future development.

## VI. Cache-conscious data and control transformations

Various techniques to optimize memory performance for cache-based multi-level memory hierarchies involve data and control transformations that result in efficient use of the cache hierarchy. Based on a knowledge of the architectural parameters and cache mapping functions, cache-conscious algorithmic techniques have been developed that help in designing custom data layouts and control transformations for a given application, resulting in reduced memory access latency.

Loop transformations (e.g., loop permutation, fusion, tiling) improve locality by changing the order of execution of loop iterations, thereby changing the access pattern. A loop transformation affects only the loop nest to which it is applied, and both temporal and spatial locality may improve as a result. However, they affect all arrays in a loop nest, some of them perhaps adversely. Tiling [LAM 91] is a common loop transformation for matrix access that divides the loop iteration space into smaller tiles (or blocks). The working set for a tile is the set of all elements accessed during the computation involving the tile. Ensuring that the working set for the tile fits into the data cache, maximizes reuse of array elements within each tile, thereby reducing capacity misses. Tiling by itself does not reduce conflict misses. Especially when the array dimension is an integral multiple of the cache size, tiling causes consecutive rows of a tile to map to the same set of cache locations. Cache conflicts in the context of tiled loops are categorized into self-interference (cache conflicts among array elements in the same tile) and cross-interference (cache conflicts among elements of different tiles of the same or different arrays). Cache conflicts that result from loop transformations are reduced by applying suitable data transformations [RIV 98].

Data transformations improve spatial locality by matching the data layout in memory to the data access pattern and are also used to reduce conflict misses. Copying [TEM 93] is a common technique used in conjunction with tiling to reduce (eliminate) self-interference misses. By copying the data elements of the tile into a contiguous buffer, copying ensures that the working set of the tile maps into distinct locations in the cache. Padding is a data alignment technique that involves adding dummy elements into the data structure. When used

with tiling, padding reduces cross-interference between different tiles of the same array (or different arrays) by creating a (ir)regular offset between the starting locations of different tiles (arrays).

The data access pattern of an application severely impacts the performance of the memory hierarchy. For example, in matrix multiplication of the form $A \times B = C$,each array has a different access pattern. A cache-conscious implementation for this application will necessitate tailoring of the loop structure and array layouts in a way that will achieve an optimal tradeoff between the cache performance of each array. Use of block or hyperblock data layout [MOO 98] for the three arrays helps eliminate a significant fraction of cache misses that occur in straightforward implementations. Architectural parameters such as cache size, determine the parameters of the data layouts used.

Dynamic data layout (DDL) [PAR 00] is another example of a high-level optimization technique that results in effective cache utilization. DDL was developed for improving performance of the Discrete Fourier Transform (DFT) application which, like many other scientific applications, exhibits data access patterns that do not possess spatial locality. Computation of DFT using factorization such as Cooley-Tukey [COO 65], results in non-unit stride in accessing the data in the computation of the Fast Fourier Transform (FFT). This results in significantly more cache misses than accessing data with unit stride, reducing the effective memory bandwidth. In the DDL approach, cache-conscious algorithmic design techniques are utilized to determine a factorization of the DFT and the data layout to make effective use of the cache. The data layout is modified between the computation stages of FFT to improve the effective bandwidth. Modifying the data layout dynamically has been shown to yield performance benefits greater than the data reorganization overhead. Using dynamic data layout for DFT computation leads to a factor of 4 improvement over straightforward implementations and up to 33% improvement over FFT using other optimizations, such as copying, on several platforms. Also, the static approach, with a fixed data layout, has been shown to yield factorizations which are not optimal and alternate factorizations with dynamic data layout result in higher performance. Finally, the cache-conscious design approach of DDL yields a portable, high-performance FFT implementation which is competitive with other high-performance implementations without using any low level optimizations.

Perfect Latin Squares (PLS) [KIM 89] is another technique originally proposed in the context of conflict-free parallel array access from multiple memory banks. PLS-based data layouts for arrays allow simultaneous, conflict-free access to rows, columns, main diagonals and major subsquares of an NxN array using N memory modules. When N is an even power of two, address computation can also be done in constant time with a simple circuit. Perfect Latin Squares are also useful in the context of optimizing memory performance for cache-based uniprocessor machines. Theoretical and simulation results [ADV 00] of the use of PLS-based data layouts and address computation for a generic matrix access application has shown significant reduction in cache conflict misses.

Cache-conscious data layouts, by virtue of being tailored for individual application domains, are very effective in improving performance by reducing memory access latency. However, coming up with generic algorithmic techniques that can automatically derive optimal data layouts, given an access pattern, is a challenging research problem.

## VII. Conclusion

Two memory latency hiding techniques - Hierarchical decoupled architecture and Simultaneous multithreading - have been proposed as the solution for the growing gap between processor and memory. Both of the above architectures exploit instruction level parallelism by utilizing run-time thread support within a single program. A decoupled architecture uses additional processor logic for the memory access thread. SMT depends on the runtime context switching mechanism during the stalling period due to a memory access. In both these architectures, the instruction level parallelism should be guaranteed by the compiler.

Although these two architectures provide an efficient method to overcome the memory wall problem, a further solution will be ultimately the raw enhancement of DRAM performance. Also, cache-conscious algorithmic techniques for data and control transformations have the potential for significantly contributing to the performance improvement of applications.

## References


[ADV 00] Algorithms for Data Intensive Applications on Intelligent and Smart Memories (ADVISOR), Univ. of Southern California. http://advisor.usc.edu

[AMA 98] AMARAL, J. N. et al. Portable Threaded-C release 1.1 Technical note 05. Computer Architecture and Parallel System Laboratory, University of Delaware. Sep, 1998

[ARV 89] ARVIND, R. S. et al. I-Structure: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, Oct, 1989

[BUR 96] BURGER, Doug; GOODMAN, James R.; KAGI Alain. Memory Bandwidth Limitations of Future Microprocessors. *23rd Annual International Symposium on Computer Architecture*, 1996

[COO 65] COOLEY, J. W.; TUKEY, J. W. An Algorithm for the Machine Calculation of Complex Fourier Series. Math. Comp., 19, 1965.

[CRA 97] CRAGO, S.P. *HiDISC: A High-Performance Hierarchical, Decoupled Architecture*, Ph.D. Thesis, University of Southern California, December 1997.

[CRI 97] CRISP Richard. Direct Rambus Technology: The New Main Memory Standard. *IEEE Micro*, Nov. 1997

[CUL 93] CULLER, D. LogP: Towards a Realistic Model of Parallel Computation. *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May, 1993

[CUP 99] CUPPU, Vinodh; JACOB, Bruce; DAVIS, Brian; MUDGE, Trevor. A Performance Comparison of Contemporary DRAM Architecture. *26th Annual International Symposium on Computer Architecture*, 1999

[DAV 00] DAVIS, Brian et al. DDR2 and Low Latency Variants. *Workshop on Solving the Memory Wall Problem*, 2000

[DEN 95] DENNIS, J. B.; GAO, G. R. On Memory Models and Cache Management for Shared-Memory Multiprocessors. CSG MEMO 363, Laboratory for Computer Science, MIT., March 1995

[DIE 95] DIEP, T. A. et al. Performance Evaluation of the PowerPC 620 Microprocessor, In *Proceedings of 22nd Annual International Symposium on Computer Archtitecture*, June 1995

[FER 86] FERRANTE, J.; OTTENSTEIN, K. J.; WARREN, J. D. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, July 1986.

[GIL 96] GILOI, W. K. et al. MANNA: Prototype of a Distributed Memory Architecture with Maximized Sustained Performance. In *Proceedings of Euromicro PEP96 Workshop*, 1996

[GUL 96] GULATI, M.; BAGHERZADEH, N. Performance Study of a Multithreaded Superscalar Microprocessor, In *Proceedings of International Symposium on High-Performance Computer Architecture*, 1996

[HAL 00] HALLNOR, Erik G. A Fully Associative Software-Managed Cache Design. *27th Annual International Symposium on Computer Architecture*, 2000

[HON 99] HONG, S. I.; McKEE, S. A.; SALINAS, M. H.; KLENKE, R. H.; AYLOR, J.H.; WULF, W.A. Access Order and Effective Bandwidth for Streams on a Direct Rambus Memory. *5th International Symposium on High-Performance Computer Architecture*, 1999.

[HUM 95] HUM, H. H. J. et al. A Design Study of the EARTH Multiprocessor. In *PACT 95*, June 1995

[KAN 99] KANG, Yi et al. FlexRAM: Toward an Advanced Intelligent Memory System. *International Conference on Computer Design*, 1999.

[KIM 89] KIM, K.; PRASANNA KUMAR. Perfect Latin Squares and Parallel Array Access. In*Proceedings of the 16th Annual International Symposium on Computer Architecture*, May 1989.

[KOZ 97] KOZYRAKIS. Christoforos. E. et al. Scalable Processors in the Billion-Transistor Era: IRAM, *IEEE Computer*, 1997.

[KOZ 98] KOZYRAKIS. Christoforos. E.; PATTERSON, David A. A New Direction for Computer Architecture Research. *IEEE Computer*, 1998.

[KUR 94] KURIAN, Lizy; HULINA, Paul T. ; CORAOR, Lee D. Memory Latency Effects in Decoupled Architectures. *IEEE Transactions on Computers*, Vol 43, No. 10, Oct. 1994

[KWA 99] KWAK, Hantak; LEE, Ben; HURSON, Ali R.; YOON, Suk-Han; HAHN, Woo-Jong. Effects of Multithreading on Cache Performance. *IEEE Transaction on Computers*, Vol. 48, No. 2, Feb. 1999

[LAM 91] LAM, M.; ROTHBERG, E.; WOLF, M.E. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, April 1991

[LIN 98] LIN, Wen-Yen; GAUDIOT, Jean-Luc. The Design of An I-structure Software Caches System. In *Workshop on Multithreaded Execution, Architecture and Compilation*, 1998

[McMA 72] McMAHON, F. H. Fortran CPU Performance Analysis. Lawrence Livermore Laboratories, 1972

[MOO 98] MOON, S; SAAVEDRA, R. H. Hyperblocking: A Data Reorganization Method to Eliminate Cache Conflicts in Tiled Loop Nests, USC-CS-98-671, USC Computer Science Technical Report, February 1998.

[PAR 00] PARK, N; KANG, D.; BONDALAPATI, K.; PRASANNA, V. K. Dynamic Data Layouts for Cache-conscious Factorization of DFT, *International Parallel and Distributed Processing Symposium (IPDPS 2000)*, May 2000.

[PAT 97] PATTERSON, David et al. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April 1997

[RIV 98] RIVERA, G; TSENG, C. -W. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

[SAU 96] SAULSBURY, Ashley; PONG, Fong; NOWATZYK, Andreas. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of 23rd Annual International Symposium on Computer Architecture*, 1996

[TEM 93] TEMAM, O.; GRANSTON, E.; JALBY, W. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. *Proceedings of Supercomputing '93* November 1993.

[THO 99] THOMAS, Randi; YELICK, Katherine. Efficient FFT on IRAM. *1st Workshop on Media Processors and DSPs*, 1999

[TUL 95] TULLSEN, Dean M.; EGGERS, Susan J.; LEVY, Henry M. Simultaneous Multithreading: Maximizing On-Chip Parallelism, In *Proceedings of 22nd Annual International Symposium on Computer Architecture*, 1995

[WIL 00] WILES, Maurice J. The Memory Gap. *Workshop on Solving the Memory Wall Problem*, 2000