NATIONAL SCIENCE FOUNDATION WORKSHOP ON HIGH PERFORMANCE MEMORY SYSTEMS FINAL REPORT

Wm. A. Wulf, Steven Moyer

Computer Science Report No. CS-93-35
June 18, 1993

National Science Foundation Workshop on High Performance Memory Systems

Editors:
Wm. A. Wulf and Steven Moyer
University of Virginia
April 12-13, 1993

1 Introduction

During the last decade the performance of microprocessors has increased by a factor of 50% to 100% *per year.* During the same period the performance of affordable (DRAM) memories has risen less than 10% per year. This growing disparity between processor and memory speeds will be one of the most pressing problems, and hence most influential factors in computer architecture, compilers, and algorithms for at least the next decade.

Given the importance of the overall problem, NSF sponsored a workshop to examine the full spectrum of relevant issues — including chip technology, architecture, caching, new concepts in hierarchy, compiler optimization, programming languages and techniques, and algorithms. Since "memory performance" is not a research area *per se* it does not have a single, separate research community. Although many of the important players know each other and meet pairwise at various conferences, there is no forum for a comprehensive analysis of the problem. The NSF Workshop on High Performance Memory Systems provided such a forum.

The objective of the workshop was to perform a comprehensive assay of the situation and suggest a synergistic research agenda for each of the areas that collectively will make a significant impact. This report contains the recommendations of the attendees.

The steering committee for the workshop consisted of Bill Wulf (U. Virginia), Jim Aylor (U. Virginia), John Hennesy (Stanford), Randy Katz (Berkeley), Ken Kennedy (Rice), Ed Lazowska (U. Washington) and Alan Smith (Berkeley). The workshop convened in Charlottesville Virginia on April 12-13, 1993.

2 Background

As noted above, although the amount of memory available has increased apace, the growing disparity between processor and memory speeds is likely to be a dominant concern for computer architects, compiler writers, algorithm designers, and users for at least the next decade.

Further compounding the issues is the increasing use of high performance microprocessors in scalable parallel systems. Both the absolute performance of these processors and their phenomenal price/performance ratios make them natural candidates for building large scientific machines. This use, however, further strains the memory system and adds the complexity of bandwidth as well as latency, dealing with the interconnection network, and the possibility of nonuniform access times.

Finally, the memory must be considered in the context of the total system; I/O as well as processor demands on it must be considered. Hence, issues traditionally considered in the domain of operating systems also play into memory system performance.

While it is not clear whether some new, unanticipated approach will arise, it seems more likely that the "solution" to these problems will lie in a synergistic interplay of evolutionary refinements of existing techniques -- caching, data prefetching, memory conscious architectures, compiler optimizations, etc. In particular, caching has proven to be a robust, general mechanism for decades; larger, smarter and multi-level caches will undoubtedly continue to play an important role. Nonetheless, it is time to consider complementary ideas.

3 Workshop Mechanics

The objective of the workshop was to bring together experts on each of the facets of the problem to formulate a set of recommendations for a research agenda to address an overall solution (those attending are listed in Appendix A).

Prior to the workshop participants were asked to submit a brief white paper summarizing their perception of (a) the status of their area of research relative to the main topic, and (b) the potential for future work in the area. Copies of the papers are located in Appendix C.

The format of the workshop was designed to maximize interaction and facilitate the production of this report. Lengthy presentations were avoided in favor of group interaction.

To stimulate discussion, four plenary panel sessions were organized around topics raised in the white papers:

- Session 1: General Problem (Chair John Hennessy)
- Session 2: Software Status (Chair Ken Kennedy)
- Session 3: Research Directions (Chair Alan Smith)
- Session 4: Memory Systems (Chair Randy Katz)

The participants were then divided into three working subgroups covering software, memory systems, and cpu/architecture issues. These subgroups were charged with formulating the recommendations of the workshop, and the remainder of this report is organized around their reports. Panelists from the above four sessions were distributed among the subgroups to ensure that hardware concerns were represented in the software recommendations, etc.

In addition to the recommendations, a number of interesting remarks by various attendees were recorded; these are assembled in Appendix B. They are not direct quotes, but capture the essence of the point and provide additional insight into the nature of the problem(s) and research agenda.

4 Processor Subgroup Report

Attendees: Keith Cooper, Mike Foster, Ken Kennedy, Howard Sachs, Pen-Chung Yew, Bill Wulf

Charge: The group was asked to identify important issues in overall system architecture, especially including instruction set design, that impact memory performance.

4.1 Observations and Recommendations Concerning NSF Funding

Due in large measure to the dynamic nature of the computer field, there has been a closer relation between the "science" and "technology" of the discipline than in many other fields. This close relation is mirrored in a similarly close relation and interaction between academia and the private sector — between research and development.This relation has been both intellectually and commercially extremely fruitful.

The nature of the discipline, and its close relation to practical and even commercial development, raises questions about the proper nature and role of NSF funding for the field. Specifically, NSF has traditionally focused on "pure", "basic", "long term" research; thus one can ask whether the sort of problems discussed at this workshop, and indeed all of those involved is "systems building" are appropriate for NSF support.

The answer is unequivocally *yes*!

For *this discipline, at this time in its development*, these issues raise fundamental questions about the organization of computation and the nature of the engines that can effect that computation. Specifically, the building and analysis of research prototype systems is an appropriate research methodology for understanding these questions.

At the same time, NSF should not fund near term development that duplicates industry. Therefore, we recommend:

- NSF's role is to build research prototypes that will not be built in industry.
- NSF should fund research that goes beyond the horizon of industry, and should explore alternatives not being pursued in industry.
- There should be emphasis on collaboration with industry to ensure the relevance of the project. This type of collaboration could be arranged after an initial exploratory concept-validation project.

For other disciplines at different stages of their maturity we recognize that this may not be the proper role for NSF. For this field it is.

4.2 Observations and Recommendations Concerning Architecture

The CPU/Architecture subgroup had six "top level" recommendations, with a number of specific examples of each. We first present the top level recommendations and then discuss the specifics.

- NSF should support research into scalable benchmark collections so that performance evaluation of architectural schemes can be made independent of advances in the underlying technology.

- NSF should support research into methods that expose and give the user or system software (OS and compiler) more control over the memory hierarchy, including registers, cache, memory, remote disk cache and mass storage.
- NSF should support research into architectural, software and programming implications of quantitative changes in base technology that lead to qualitative changes in the way that a technology is used.
- NSF should support research into alternatives to conventional memory hierarchy design as pursued by industry, particularly when a change in technology may obviate some of the assumptions behind the conventional design.
- NSF should support research that explores the implications of large-scale multiprocessors for the design and management of the memory hierarchy.

4.2.1 Scalable Benchmarks

This recommendation was actually first made during one of the plenary panel discussions, but the CPU/Architecture group felt that it was extremely important that it should be included among their recommendations. Almost all current benchmarks are static, and their performance improves with each successive hardware generation. This is, perhaps, useful for deciding which competing system to buy, but it does not help us to understand whether that system has effectively exploited the technology advances over its predecessors.

A "scalable" benchmark is one whose complexity scales with the technology in such a way that if it requires unit execution time on one generation's hardware, one would expect that it would also require unit execution time on the successor generations. If execution took *less* than unit time, the implication is that architectural or software innovation has exceeded the pure hardware advance. If execution took more than unit time, the implication is that the underlying hardware advance was not effectively exploited.

4.2.2 Exposing The Architecture

In the past one objective of architectural design was to make a machine relatively easy and safe to use; assembly language programming made this essential. However, few if any programs are still written in assembly language and compiler technology has become *much* better. It is therefore worthwhile reexamining this premise, and possibly expose much more of the underlying operation of the machine in those cases where doing so may improve performance. Examples possibly worth considering include:

- *prefetch*: Prefetch instructions have been discussed in the literature, but only one extant processor includes such an instruction — and it is a nop in the initial implementation. Moreover, expression of intent to prefetch of multiple data items (as opposed to scalars), should be investigated.
- *cache management*: Modern compilers sometimes know a good deal about when data will and will not be used. After a step of a blocked algorithm, for example, the compiler could inform the cache to flush certain data. In other cases, the compiler could control a set associative cache to avoid conflicts.
- *memory map manipulation:* A great deal of copying of data might be eliminated if safe user-level manipulation of the virtual memory map were possible.

These may or may not be good ideas in themselves — the real point is that the traditional assumption that these kinds of functions should be automatic and invisible to the programmer are not necessarily valid any longer.

4.2.3 Implications of Qualitative Changes

Sufficiently large quantitative change can induce qualitative change. There are enough quantitative changes happening simultaneously to suggest that we should investigate whether these are about to either invalidate prior assumptions or provide new opportunities to improve memory performance. For example:

- The next generation of processors will clearly be true 64-bit machines. This at least creates the possibility of 64-bit instructions, and raises the question of how so many bits would be used for a single instruction. For example, one could use a full 64-bit instruction to name very large register sets (effectively a local memory)?
- In the same vein, caches were the "right solution" in an era when compiler technology was less sophisticated than now. Is it conceivable that with current or foreseeable compiler technology, an explicitly managed local memory (or large register set) would be more effective? A cache is, after all, a *reactive* device; in many cases a compiler knows likely future execution paths and could be proactive.

- The broad adoption of object-oriented programming could potentially reduce locality (at least data locality), and require rethinking caching strategies. Data derived from traces of "pre-OOP" programs may not be representative of future program behavior.

Again, these may not be good ideas in themselves — the point is that it is time to examine some of the generally accepted assumptions.

4.2.4 Alternatives to Conventional Memory Hierarchies

The current memory hierarchy has served us very well indeed, and it may continue to do so. Nevertheless, the degree of the growing discrepancy between processor and memory speeds suggests that in its rôle as funder of longer term research, the NSF should begin exploring alternatives. The creativity of the research community is probably the best source of new ideas, however a few examples might be:

- Explicitly managed local memories, as discussed above. In particular the trade-off between cache and large register sets should be explored.
- Reordering of accesses. Modern DRAM isn't really "random access" any longer; that is, the time to access a particular cell now depends upon the prior history of requests. Thus, to get the most from these devices the order of requests must be managed — both compile-time and run-time mechanisms to do this should be explored.

4.2.5 Implications of Large Scale Multiprocessors

The trend to use production microprocessors as the compute engines of scalable multiprocessors will continue. This trend compounds each of the issues mentioned above, plus adding another set of its own. For example,

- Prefetch doesn't really work for "do across" concurrency; a form of data forwarding is needed. What is the proper architectural support?
- Coherence in a scalable shared memory system can be costly. Perhaps with compiler help not all data references need to be kept coherent. Should we consider separating coherent and non-coherent load/store instructions and permitting the compiler to emit the latter when it can prove that coherence is not required?

5 Memory System Subgroup Report

Attendees: Jean-Loup Baer, Forest Baskett, Doug Clark, Garth Gibson, Charley Hitchcock, Norm Jouppi, David Wood

Charge: The group was asked to identify important issues in memory systems design, including storage systems.

The Memory System Subgroup began by asking why anyone, but NSF in particular, should focus research on the memory problem, and decided that there are at least three important reasons.

- Intensive applications on current machines can spend 1/2 to 2/3 of their time waiting for memory, e.g., Gray's sort benchmark on an Alpha system takes under 10 seconds, with 6.5 seconds of that waiting for memory. Future machines will be worse. This trend hinders the exploitation of rapidly increasing processor speed.
- As machines become more capable, we find larger and more complex problems to solve, e.g., national libraries, human genome, real-time speech and video, etc. Significant advances in technology, architecture, software, and algorithms are required to keep pace with these rapidly changing demands.
- The United States is the leading innovator in computer systems. We have a significant competitive advantage and should strive to keep it. This advantage has been fostered by significant government investment in research, and close ties between academic researchers and industry. NSF needs to continue to encourage cross fertilization between industry and academia.

In order to conduct research on memory systems, there must exist an "infrastructure" that includes:

- Application Characterization: Memory system researchers need a deeper understanding of key problems, such as national libraries, real time speech, Navier Stokes, HD-TV, multimedia, and the human genome project. Classical problems such as dense linear algebra are relatively well understood throughout the community. However, many researchers are unfamiliar with the characteristics of these other problems, particularly on parallel machines. Work that identifies the essential nature of these problems to the community at large would be extremely valuable.

- Design Verification Tools and Techniques: As hardware design becomes largely a matter of writing software, design verification has emerged as the most time-consuming and error-prone part of the process. More powerful design verification tools and techniques are needed to eliminate this bottleneck and reduce the design cycle.
- Compiler Infrastructure: Past research has shown that compiler optimizations can significantly improve memory system performance. However, because a good optimizing compiler represents many person-years of work, this research has been limited to those few institutions with mega-funding. Research in compiler/memory system interactions could be funded more efficiently by supporting a common compiler infrastructure, just as Berkeley UNIX and Mach have facilitated operating system research.
- Simulation Infrastructure: Studying compiler/memory system interactions requires both an optimizing compiler and a real or simulated memory system. Simulation infrastructure should be supported so that researchers are not forced to replicate the simulation framework.

Given this infrastructure, the group identified a number of major areas that NSF should fund or continue funding: DRAM interfaces, Processor/cache interfaces, Cache Hierarchies, Shared memory coherency and communication, Virtual memory and translation, Disk Caching and Parallel Storage, and Prefetching support. Below is a brief discussion of the key needs in each of these major areas.

5.0.1 DRAM interfaces

The standard RAS/CAS DRAM interface was designed with an emphasis on low-cost and high-density. As microprocessor cycle times continue to decrease, the limited bandwidth available from this interface becomes an increasing bottleneck. A new DRAM interface is central to enabling future systems to exploit the potential of next generation processors. While several new interfaces have been propose to address this problem, it does not appear that a consensus will soon emerge. The heart of this debate focuses on cost/performance trade-off: the PC market--which dominates DRAM sales--is very cost sensitive, while the workstation and massively-parallel processor markets demand higher bandwidth.

The group felt that academia could play a critical role in analyzing and evaluating the alternatives, perhaps developing a new interface that balances the needs of the competing markets. One reason to believe a compromise may be possible is that future PCs will support "multi-media" capabilities--such as real-time video--and hence require much higher bandwidth than current systems.

5.0.2 Processor/Cache Interfaces

Current generation microprocessors are designed for PCs or workstations--not high-performance MPP-class machines. These micros tend to be optimized for applications that exhibit significant locality, and consequently do not provide the bandwidth required by large-scale scientific computations. Research is needed to find an interface that appropriately balances cost and bandwidth requirements for diverse applications.

5.0.3 Cache Hierarchies

As transistor densities increase, the role of on-chip and off-chip caches change. For example, some researchers have suggested that multiple levels of on-chip cache are necessary to balance competing demands for bandwidth and capacity. Continued research is required to understand these trade-off, including traditional issues such as size, associativity, and write policy, as well as more aggressive techniques such as non-blocking caches, multi-porting, multiple banked caches, and hardware and software prefetching. In particular, some industry participants felt that non-blocking caches are difficult to implement and verify, making this a fertile area for collaboration between academia and industry.

In addition, research is needed to understand how trends in applications will affect cache performance. For example, object-oriented programming results in fundamentally different access patterns, which cache designers must take into consideration.

5.0.4 Shared Memory Communication and Computation

Shared memory is an important paradigm for large-scale parallel machines. However, researchers have not yet reached a consensus on how this paradigm should be supported. A key requirement in reaching this consensus is to understand the communication and computation demands of real parallel programs. While much has been learned about the behavior of dense linear algebra, there are many other important codes, e.g., Navier Stokes, that have fundamentally different structure. We need to understand the bandwidth requirements, access patterns, and synchronization requirements of these and other sparse codes. What demands do these applications place on the memory system and interconnection network? What are the correct metrics for evaluation? Aggregate memory bandwidth? Bisection bandwidth?

5.0.5 Virtual Memory and Translation

Microprocessor architectures are moving rapidly to support 64-bit virtual addresses. However, little research has been done to understand the impact of this massive address space on applications, operating systems, and memory system implementations and performance. Does a large address space enable new applications and algorithms, such as studying the human genome? What mechanisms must hardware support to efficiently exploit these large address spaces, e.g., super-pages? What are the implications of managing very large virtual address spaces? Of managing very large physical memories?

5.0.6 Disk Caches and Parallel Storage systems

Research into redundant disk arrays (or RAID) is not diminishing and should receive continued support. Industry is picking up the technology, but the vast majority of participating companies are too small to be capable of doing or funding broad-based research into this area. Larger companies have started in-house development groups, but their patenting efforts seem to limit their ability to foster this fledgling marketplace. There is an industrial organization, the RAID Advisory Board, that has the goal of developing a RAID marketplace. Unfortunately, this group of mainly technical marketing representatives led by industry consultants, is focusing on standardizing RAID technology rather than encouraging new basic research. Now, as we begin to see a broad spectrum of computer science researchers address this area, the first performance models and new organizational concepts for on-line transaction processing applications have appeared. Sustained funding for systems experimentation, theoretical issues in scaling to massive parallelism, and applications-directed solutions should ensure that basic advances continue to broaden the effectiveness of this new technology.

5.0.7 Prefetching: Hardware vs Software

Recent research results indicate that software-controlled prefetching can be extremely effective in hiding memory latency. Similar results indicate that hardware-initiated prefetching can also be effective. Determining the proper balance between hardware and software remains an extremely important research area, because of the significance of the potential performance improvements. Continued research is needed to determine the trade-offs of prefetching into different levels of the memory hierarchy, level of hardware support, granularity of data, and interaction with cache coherence protocols. It is extremely important that this research incorporate realistic implementation factors, making it an excellent opportunity for academic/industry collaboration.

6 Software Subgroup Report

Attendees: Susan Eggers, John Hennessy, Randy Katz, Monica Lam, Alan Smith, Steve Scott

Charge: The group was asked to identify important issues where software can impact memory performance, including algorithm design, compilation techniques, coding practice, etc.

We feel strongly that entire of discipline of software research needs to pursue a more quantitative, comparative methodology in the research process. The current state of the field, by and large, focuses on algorithm development coupled with some limited prototype implementations, but little comparative empirical studies are pursued. New work is not placed in the context of quantitative improvements to the existing body of knowledge. Experiments are not repeatable in a scientifically meaningful way. And all too often, experiments lack real and relevant measurements.

To remedy this situation, the Software Subgroup makes the following recommendations:

- Software researchers should work with real systems, or at the very least, large pieces of software representing major subsystems.
- Software research should be driven by real world applications and realistic workloads.
- To obtain access to these, it is important for software researchers to work with industry.
- The research community should demand comparative, quantitative results in its publications and research proposals. New ides should be evaluated on their ability to demonstrate a real improvement to the existing state of the art.
- The research community should focus on collecting real system measurements, not only those that are easy to measure. For example, all too many researchers focus on miss rates rather than total system performance.

- The community should support on-going technology exchanges with industry. Faculty researchers should work actively with industry, either by regular consulting, close collaborative research or "co-op" programs.

Systems research typically falls into two categories: simulation-based studies and prototyping activities. Both are important, and should be encouraged, but encouraged for answering the right questions.

To facilitate prototyping activities, the community needs substrates upon which to build its prototypes rather than starting from scratch. In this regard, the operating system community is in a stronger situation than the compiler community. V-node file systems and microkernel operating systems make it easier to prototype (and trace) new file systems. A similar research infrastructure is needed for compiler researchers. Part of the reason for lack of progress in this regard for the compiler community is that we are just now realizing that a compiler is a "big program." The kind of modular decomposition found in modern operating systems has no analogies in modern, memory conscious compilers.

The Working Group recommends that some research funding should be directed to those activities that characterized by building infrastructure in support of prototyping activities. All funded research should be "credible," along the lines of the criteria described above. Today, there exists a significant funding hole for medium-scale software projects, at the level of a few faculty and several grad students. Such projects could be successfully pursued should the appropriate infrastructure be put in place. By focusing on credibility, we believe that existing research would improve in quality. While the construction of such research infrastructure may be beyond the usual scope of NSF support, NSF support could complement such activities.

In the above, we have focused on prototyping activities. There continues to be a strong need for work in fundamental inventions and evaluations across all the software areas. However, it is important to avoid research on theoretical approaches that are demonstrably in conflict with reality.

6.1 Technical Directions

We begin by listing those technical directions that are worthy of continued research support. Each area is characterized by three metrics: impact, speculativeness, and effort. Impact qualifies the potential effect on the field of research in this area. To some degree we have downgraded impact when we thought that solid research effort in the area were already underway. Speculativeness qualifies the risk of the area. A highly speculative area means that some ideas may turn out to be highly useful while many will fail. And finally, effort describes the level of funding needed to demonstrate quality of ideas.

The five research areas are the following. The first three are particularly singled out for high priority efforts:

- Architecture-directed Compiler Research
  Potential Impact: High; Speculativeness: Low; Effort: High
- Research in File and I/O Systems
  Impact: High; Speculativeness: Medium-Low; Effort: Medium
- Language Research
  Impact: Very High; Speculativeness: Very High; Effort: Low-High
- Research on Applications Behavior and Improvement
  Impact: Medium; Speculativeness: Medium; Effort: Medium
- Parallel and Memory Sensitive Performance tools
  Impact: Medium; Speculativeness: Low; Effort: Low

These are described in more detail in the following sections.

6.1.1 Architecture-Directed Compiler Research

There are a catalog of techniques under investigation that have the potential for improving application performance, both for memory and secondary storage. These include the following:

- Prefetch
- Loop Transformations
- Data Restructuring
- Coherence Management
- Code Scheduling
- Run-time Determination of Compilation

- Whole Program Compilation. In particular, we need to understand the behavior of the whole program in order to do a credible job of data restructuring and extraction of parallelization.

In general, many of these techniques are well understood for programs that manipulate arrays. However, they are not understood at all for more application-oriented data structures.

6.1.2 File and I/O Systems

The I/O access gap continues to increase, and is likely to continue to widen for the foreseeable future. The important research issues include the following topic areas:

- Disk Cache. In this area, there still exists a need to understand the algorithms for cache operation, how to exploit parallelism in the design of the disk cache, the effect of cache parameter choice on total system performance, and the issues of disk cache consistency in a parallel environment.
- Evaluation of new file and I/O techniques. New I/O architectures like RAID and file system approaches like log structuring need to be evaluated with real workloads, such as production commercial and scientific environments.
- Caches and Buffers. The role of disk caches and operating system-managed I/O buffers need to be evaluated within the context of new technologies like RAID and/or LCFS.
- File Migration Studies. The studies done in this area are over ten years old, while the underlying technologies have undergone radical challenges. They should be revisited in the context of new workload demands and new multi-level storage technologies.
- Shared data management. New algorithms need to be developed and evaluated for (real) management of shared data in distributed and MP systems.

6.1.3 Language Research

We believe that there are significant opportunities in developing new language extensions (as opposed to new computer languages) that are applications specific. In particular, we are interested in extensions that ease the expression of parallelism and enhance locality.

We call these "application structures." For example, little is known about how to specify and compile tree data structures in modern compilers. Another example is sparse matrix support, with the related need to develop compiler algorithms to discover and exploit non-regular access to data structures. There is a strong relationship between research in compiler techniques and research in extending computer languages.

Besides the data structure specifications, a second area of language research involves the specification of the user's model of the underlying memory system. How much does the user need to specify? How much does s/he need to know about the details of the memory hierarchy? An aspect of any such a model must be like an onion skin. In other words, the user can exploit more details in order to get more help from the system. The hints that are specific to a given machine should be discouraged, while hints that are specific to a given applications should be encouraged. In any event, no hint should change the semantics of the program.

6.1.4 Applications Behavior and Improvement

Since we can't fix the application after it is written, an important research direction is to work on ways to express the algorithm correctly the first time. Such specifications are not just architecture driven, but should be influenced by applications as well.

A critical need is to find new ways to provide performance feedback to applications developers. They must understand the behavior of their applications in order to drive the cycle of improving program performance.

6.1.5 Parallel and Memory Sensitive Performance tools

Good work in performance tools is currently underway, but more needs to be done. The tools are critical if applications designers are to be able to write better programs. They play a major role in providing the feedback mechanism to users. In particular, applications developers should be able to use such tools to obtain direct quantitative measures of the effects of their hints on the performance of the application.

Performance tools make possible the kind of quantitative, credible research we called for in Section I above. There should be some emphasis on documentation/distribution of existing tools, to make them available to a wider research community than the group who originally developed the tools. This should be considered part of the necessary "infrastructure" for supporting the software research community.

7 Summary

A short workshop cannot hope to invent and enumerate all aspects of a research agenda for a problem as critical and complex as this one. We hope that we have made three essential points, however:

- The problem is a critical one.
- Business as usual is unlikely to solve this problem; new ideas are required.
- This is the sort of problem requiring building, measuring and/or experimenting with "real" systems, and NSF should fund this sort of research.

8 Acknowledgments

Ms. Kimberly Gregg made all of the arrangements for the workshop, and kept it running smoothly. Graduate students Mike Alexander, Mark Bailey, Sanjay Jinturkar, Sally McKee, Steve Moyer, Carmon Pancerella, Ramesh Peri, and Alec Yasinsac took the notes and made the recordings from which this report was compiled as well as assisting in many other ways. The attendees express their gratitude!

Appendix A: Participants

Those attending the workshop were:

Jean-Loup Baer	University of Washington
Forest Baskett	Silicon Graphics
Doug Clark	Digital Equipment
Keith Cooper	Rice University
Susan Eggers	University of Washington
Mike Foster	National Science Foundation
Garth Gibson	Carnegie Mellon University
John Hennessy	Stanford University
Charles Hitchcock	Dartmouth College
Norm Jouppi	DEC WRL
Randy Katz	Berkeley
Ken Kennedy	Rice University
Monica Lam	Stanford University
Edward Lazowska	University of Washington[1]
Howard Sachs	Integraph
Steve Scott	Cray Research
Alan Smith	University of California, Berkeley
David Wood	University of Wisconsin
Bill Wulf	University of Virginia
Pen-Chung Yew	University of Illinois at Urbana-Campaign

1. Unfortunately, Professor Lazowska had to cancel at the last moment.

Appendix B: Interesting Remarks

During the workshop, a number of interesting interchanges occurred, and we have tried to capture a few of them here. These are not direct quotes, but attempt to fairly represent the point that the speaker made. Have grouped these by topic area.

General Remarks

Howard Sachs: OOP needs simulation and research dollars and it'll be a big win for industry since we're heading down this path. We still don't know the extent of the OOP effect on memory systems.

Ken Kennedy: We should fund more compiler research (said with grin).

Susan Eggers: In order to determine general funding directions, we need to: determine what the problems really are (by doing research to explore this). Let the problems drive our hardware and software solutions. We need to emphasize "bottom line" technology, use real workloads, use execution time as the performance metric, and develop good software platforms for real measurements and comparison.

Howard Sachs: We need more compiler and language PhD students. Will there exist a DRAM problem in the year 2000? yes and no. Will 90% of industry have a problem? no -- because they'll be using PCs. So where's the problem? workstations object-oriented programming multiprocessors (not for 8-processor SMPs, but definitely for 500-processor MPPs -- and this will be a bandwidth problem, rather than a latency one).

Ken Kennedy: Base technology drives what we're doing in the architecture community. Should this also be driven by compiler and applications people?

Ken Kennedy: Our research has to be linked to technological trends in order for it to have any impact -- if we look ahead too far and technology takes a different turn, even interesting research might become "useless".

Alan Smith: The major problem with proposals is reviewing. If you get better reviewers, better projects will be funded, and better research will be done.

Alan Smith: A bunch of graduate students using a toy file system does not represent real workloads.

Alan Smith: We need a co-op program in industry for faculty, similar to one for students.

Howard Sachs: We need standards in order to influence semiconductor manufacturers. We have to solve problems architecturally at a level above that of the DRAM cell.

Concerning Research Infrastructure

Dave Wood: How do we evaluate the benefits of hardware support for memory performance? We NEED a simulation platform. We need a standard intermediate form to communicate much of the compiler analysis to machine-dependent backends in order to make use of all the hardware's resources. Right now it's hard to separate the analysis from the algorithms that make use of that information.

Concerning Benchmarking

Steve Scott: We want benchmarks to scale with the underlying raw technology in order to gauge how effectively we're USING that technology. We need to extend benchmark suites to other areas; for instance, we want a graphics benchmark suite, or an object oriented benchmark suite.

Concerning Architecture

Pen Yew: Packaging constraints (clusters) correspond well with application code. The memory hierarchy should be visible to OS, compiler, and user/language.

Bill Wulf: Compiler/architecture interactions are *the* thing to look at.

Ken Kennedy: Memory organization should only be exposed to the user if the OS/compiler is lousy. We need to hide the memory hierarchy from users.

John Hennessy: Once you tell user about memory hierarchy and locality, what/how do you tell them? We need a memory model to explain to programmers.

Bill Wulf: The question is where structurally dissimilar hierarchies can be described in the same way.

Susan Eggers: There should be a difference between an actual memory model of a machine and an abstract memory model. An abstract memory model should be used for the programmer -- an actual model is too complicated to explain to a programmer.

John Hennessy: If we tell the user something about the memory hierarchy, it should be relatively SIMPLE (the same goes for parallelism). "Local" versus "Remote" should be a good enough programmer's model.

Bill Wulf: Don't forget bandwidth-limited applications. The more improvements that are made in DRAM organization, page mode, etc., the more important it is to use these components wisely in order to take advantage of their capabilities to deliver bandwidth.

John Hennessy: Many techniques to increase bandwidth also increase latency. We can buy bandwidth with dollars, but things don't scale linearly.

Steve Scott: Money can buy bandwidth, but not latency.

Concerning Multiprocessor Architecture

Pen Yew: Designing better memory systems for parallel machines requires better understanding of parallel program behavior, better benchmarks (we need a common suite), more and better performance measures and simulations on future and existing machines.

Steve Scott: $P < 2M$ (meaning that the importance of parallelization is less than twice the importance of memory optimization, in terms of overall importance).

Alan Smith: Bus traffic is so high that multiprocessors are only MARGINALLY useful.

Concerning Caches

John Hennessy: There has been more cache research and many more cache papers (than I/O papers) because it's much easier research to do.

Susan Eggers: Stressing temporal locality isn't enough. There are too many restrictions on how we can use the cache. The user doesn't get the expected performance.

Steve Scott: We're seeing the same problems with VM -- the user should write code with locality.

Garth Gibson: Building larger caches doesn't scale performance linearly. Doing this assumes that the workload is constant. The bottom line is that linear increases in cache size are not sufficient for database and scientific computing applications.

Alan Smith: Published data indicates that the miss rate goes down with the square root of the cache size. Is designing DRAM for use with cache different from designing DRAM for stand-alone memory systems?

Bill Wulf: (concerning cache coherence) Parts of applications don't need coherence; can we productively separate access into those that need coherence and those that don't? If so, then we needn't pay for it all of the time. This is another area where compiler/architecture interaction is important.

Concerning Memory Component Design

Forest Baskett: The biggest thing we need to accomplish is to provide guidance to DRAM venders. There is a whole set of potential [DRAM organization] improvements possible (cache, organization, voltage, pinouts, etc.) -- DRAM organization is a relatively new field.

John Hennessy: The problem is getting everyone to agree on what we should do, and then sell that to the PC market -- if each improvement remains a "niche technology", we're doomed.

Forrest Baskett: In order to influence technology on this huge growth curve, we have to provide easy, cheap solutions, or else it won't be worth it for venders to make changes. Venders need a volume market.

John Hennessy: IC and DRAM technologies are giving an exponential growth curve with respect to size -- and we *don't want* this to stop.

Steve Scott: We need to sell ideas to customers in order to get DRAM manufacturers to listen. The PC industry drives the DRAM business, and manufacturers are scared to introduce anything that isn't standard RAM.

Concerning What Software can Do

Keith Cooper: Hardware design is based on probability, but users care about performance on THEIR code (the characteristics of which may not match the assumptions made when the hardware was designed).

Keith Cooper: What can compilers do? change reference patterns, understand specific locality, change layouts and alignments at runtime. We need to do things differently, and stop doing "more of the same" with respect to larger or greater numbers of caches.

John Hennessy: People claim every few years that they have understood the compiler field completely and there is nothing left there; but then they keep going back to it. I feel that there is lot to be understood and done yet.

Howard Sachs: Industry is desperate for compiler writers.

Concerning IO

Alan Smith: Research in FILE and I/O gets much less attention than cache memory systems, but is no less important.

Appendix C: White Papers

Tolerating memory latency

High Performance Memory Systems Workshop

Jean-Loup Baer
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Wa 98195

March 1993

1 Motivation

Peak processor performance has now surpassed the 100 MIPS level. With the current VLSI developments, several functional units, instruction and data caches, and further hardware support functions can be included on a processor chip. Processors are backed up by very large (several megabytes) second-level caches and main memory. In the case of shared-memory multiprocessors or distributed memory systems, the connection to memory can be implemented in a variety of ways (bus, ring, mesh or multi-stage interconnection network, scalable coherent interface) depending on the size of the systems. With processor speed increasing very rapidly, and memory latency and bandwidth progressing at a slower pace, we must investigate techniques to reduce the effects of memory latency, these effects being exacerbated by physical constraints (e.g., chip crossings, latency in the interconnection networks) and logical constraints (e.g., cache coherence, hot spots, synchronization).

The introduction of caches, in the late sixties, was motivated by the goal to balance processor speed and memory access time. Caches have been extremely successful since they not only provide the desired performance enhancements but also they are completely transparent to the software, i.e., until now, the introduction of a cache has been an organizational matter, not an architectural decision. However, with larger caches, the cache access time again becomes disproportionate with the processor speed. Today, even with two-level caching, the cost of memory access is significant enough so that a system approach must be taken for the management of the cache hierarchy (e.g., flush, fence, prefetch, and poststore instructions).

In order to put these effects in perspective, consider the following table of memory latencies with a normalized processor cycle time of 1 (C1 is a first-level or on-chip cache; C2 is a second-level cache; MM is main memory).

	C1	C2	MM (close)	MM (far)
100 MIPS workstation (best guesses)	1	6	25-60	
Dash	1	5-15	8-29	26-132
KSR-1	2	20	150	570

Let us use as metric the average number of cycles per instruction, or CPI. The component of the CPI due to cache misses depends on two factors: miss ratio and memory latency. Even with the very small miss ratios that have been recorded on numerous benchmarks, it is not unusual to see the average CPI due to cache misses be as much as 2 or 3 because of the high latencies. Thus, the fact that latencies are very large compared to the processor's speed implies that cache misses have still an important impact on the loss of efficiency of the system. The memory latency problem that had been "solved" (in uniprocessors) first by the introduction of caches, then by cache hierarchies, still confronts the designers of high performance machines. This problem is compounded in shared-memory multiprocessors because latencies are larger if an interconnection network is used, or there is more resource contention (e.g., access to a shared-bus, hot spots in access to a memory module), or there is extra communication needed because of the cache coherence requirements and inter-process(or) synchronization. Since the trend is for the decrease in physical memory latency to lag behind the increase in processor speed, the CPI increment due to cache misses must be reduced by methods that decrease the miss ratio and/or hide the memory latency.

Looking at the previous table, we see that a memory reference that results in a miss at the first-level cache and a hit at the second-level has a penalty of roughly one order of magnitude. Any mechanism that is meant to reduce the latency between the two lowest levels of the memory hierarchy should then be non-intrusive. This calls for either a hardware scheme that is not on the critical path and that does not "steal" cycles from the execution of the instruction stream, or for a software scheme that can run concurrently with the ordinary stream (e.g., in a super-scalar processor or one with a load/store unit and multiple instruction issue). Of course, compiler optimizations can be added at no run-time cost to both software and hardware schemes.

When the penalty for referencing memory reaches two orders of magnitude, then spending two or three cycles on an extra instruction and/or associated address computation is not as critical. Therefore at this level, one can envision using sophisticated software methods and, if proven useful, hardware assists at the second-level cache.

Among the hardware-based methods that have been used or proposed, we can list:

- Cache hierarchies (now present in many medium to high-performance systems) and cache assists (write buffers, victim caches)
- Lock-up free caches (with various degrees of sophistication).
- Hardware-based prefetching (from simple sequential stream buffers to cache-like reference prediction tables).
- Relaxed memory consistency models (and the hardware assists they require).

On the software side, with an emphasis on the multiprocessor case:

- Software cache coherence schemes.
- Data placement (increasing locality and reducing false sharing).
- Prefetching and poststoring.

2 Prefetching

Looking only at prefetching, the previous discussion leads to the following matrix:

	C1-C2	C2-MM
Hardware	Yes Special sophisticated functional unit	Maybe Special unsophisticated functional unit
Software	Yes if super-scalar or multiple issue	Yes Sophisticated compiler

Although we intend in the long run to look at the four elements of the matrix, we have concentrated on the hardware-based prefetching at the C1-C2 level and on some aspects of software prefetching at the C2-MM level.

Our current and future research plans are as follows:

Level 1 to Level 2: Hardware-based prefetching

At this level of the memory hierarchy, we do not make any distinction between the single processor and the multiprocessor case. Coherence effects should not have a great impact on prefetching to the first-level cache if some form of inclusion is implemented. Hardware prefetching should be useful in reducing capacity and conflict misses in the (relatively) small and most likely direct-mapped first-level cache.

Our approach has focused mainly on loop-domain references in applications. It combines the dynamic determination of stride information and conventional instruction look-ahead. The variations on a common scheme consists of a support unit for the C1 data cache, consisting of a Reference Prediction Table (RPT), a Branch Prediction Table, and associated logic. The RPT is organized as a regular cache and records the referencing patterns. The key to hiding memory latency is to keep enough distance between the prefetches and the execution stream so that the prefetched data arrives just, or slightly before, it is needed.

The type of questions that can be asked are:

1. How complex should the associated unit be? For example, would a complex scheme, akin to the Two-Level Adaptive Branch Prediction, have a significant impact on non-numeric applications (we have evidence that a simpler scheme works well for numeric applications)?
2. Should prefetching be done in the D-cache, or in a buffer, or associated with victim caching to reduce conflicts misses caused by prefetching too early?
3. What significance have the usual cache parameters (capacity, line size, associativity) and the amount of bandwidth between the first and second level caches (regular or split bus)?

Level 2 to Main Memory: Aspects of software prefetching

Software-directed approaches rely on data access patterns detected by static program analysis. Within the context of the interface between a second-level cache and main memory, and most importantly in the case of a shared-memory multiprocessor but the same analysis will be valuable for a distributed memory architecture, a prefetching algorithm must answer the following questions: when to prefetch (certainly quite far in advance for tight loops), what data is a candidate for prefetch (the penalty of prefetching unused data is high since there is not only pollution in the cache but also increased utilization of the interconnect), and what is the size of the prefetched data. The same type of algorithm could be designed to poststore: which data should be broadcast, to whom, and when?

At this point, we are planning studies on an architecture based on a tree hierarchy of busses (we have an instruction level simulator for that system). The main focus of attention will be the correlation between prefetching and data placement (where in the hierarchy should we keep the prefetched data) in the various clusters and levels of the hierarchy. The data placement problem should also provide some interesting insights of programming machines relying on message passing between clusters of processors. Our initial approach will be pragmatic and rely on the application programmer to insert the prefetch operations and do the data placement. In the long run, we might want to investigate how this fits within languages such as HPF.

3 Summary

Means to reduce or tolerate memory latency is a challenge that cannot be avoided if we want to take full advantage of current technology. We feel that part of the millions of transistors that are now present in a single chip could be advantageously devoted to a special-purpose functional unit for prefetching. We also feel that software prefetching, and associated data placement, is crucial for improving the performance of programs running in either a shared-memory multiprocessor or a distributed memory machine.

Compilers, Microprocessors, and Memory Systems

Preston Briggs
Keith D. Cooper

Rice University
Houston, Texas 77251-1892

1 Introduction

One key performance problem in today's microprocessor-based computers is the shifting balance between the speed of a floating-point multiply instruction and the speed of a load instruction. A decade ago, load was fast and multiply was slow. That situation has reversed over the last five years. Within a couple of years, we may see systems with processors clocked at three nanoseconds talking to fifty or seventy nanosecond memory chips. To make effective use of these fast processors, the combined hardware-software system must hide this speed mismatch. That will require changes in our compilers, our operating systems, our microprocessors, and our systems architecture.

This white paper tries to assess the state of compiler techniques for hiding memory latency. It looks briefly at trends in memory system design. Finally, it suggests a strategy for deploying our resources, in both software and hardware, to improve our ability to manage latency in real computations.

2 What can compilers do today?

Recent years have seen a large body of work developed around the problem of providing adequate performance on cache-based systems. This section tries to assess what's possible in compiler-based cache management today.

Any attempt to improve cache behavior through compile-time techniques must begin by trying to understand when cache misses occur. Several groups have looked at this problem. Porterfield *et al.* proposed a simple scheme for discovering the *overflow iteration* – the iteration of a loop where a cache miss must occur [7, 1]. Mowry *et al.* use another scheme to estimate where cache misses must occur [6]. Both techniques rely on dependence analysis to spot temporal reuse. Spatial locality is usually found by looking at loop strides. Gannon and Jalby proposed a technique based on reference windows that is an alternative to dependence-based techniques [3]. Their reference windows allow them to notice spatial locality, too.

Discovering where to improve the code is just the first step. To make the code run faster, we must transform it so that its behavior better maps onto the memory hierarchy. Work in this area falls into three major categories: blocking, copying, and prefetching.

2.1 Cache blocking

Cache blocking (also called *tiling* or *strip mining*) is a technique for reshaping the iteration space of a loop to improve its locality [9, 10, 8, 2]. Some loops overrun their reuse – they would reuse values from cache except that those values have been knocked out of the cache by intervening references. Blocking improves the performance of these loops by bringing references to a single location closer together in time. While the specific transformations involved in blocking varies from paper to paper, they include loop interchange, strip mining, loop skewing, and loop reversal.

Blocking improves performance in one specific case: when reuse exists, but the combination of cache and program behavior results in replacement of data before its reuse. Such premature replacement comes from several sources:

1. Alignment of data can cause interference (both within a single data structure and between multiple structures).

This research is supported by ARPA through ONR grant N00014-91-J-1989.

Warning: This paper contains strong opinions on hardware. We have drawn these opinions from reading and listening. We have not done the extensive simulation studies that would be required to assess completely the practicality of our ideas.

2. The number of items used between successive references can exceed the cache's effective size, ensuring replacement before reuse.

3. The replacement algorithm implemented in the cache can be a poor match to the program's actual behavior.

Blocking directly addresses the second problem. By moving successive references closer together, it increases the likelihood of successful reuse – reuse within the cache. As a side effect, it may decrease the likelihood of the other two problems, but this is an indirect effect. Blocking changes neither alignments nor replacement policies; it reshapes the iteration space.

2.2 Copying

Copying [4] (or *streaming* [5]) attempts to improve behavior by treating the cache as a fast local memory. In effect, the generated code treats the cache as a single large array. Values are explicitly copied into the array before use and modified values are copied back to main memory after their last reuse. This lets the code sidestep problems of alignment, stride, and replacement policy; the compiler tries to dictate all data movement in and out of the cache. As a further, often ignored benefit, copying can remove TLB misses from inner loops – making performance far more stable.

Copying imposes some overhead. Lam *et al.* show that the benefits of copying often exceed the overhead [4]. To minimize the overhead, the map from subscripts in the original code to subscript in the pseudo-array used to represent cache should be simple. A further problem arises if the instructions used to load and unload the cache allocate cache lines themselves. This introduces a subtle problem with self-interference – the loads required to copy the data into cache can cause replacements of data already moved into the cache.

2.3 Prefetch (and flush)

Both blocking and copying work within a given architecture. The use of an advisory prefetch instruction offers an interesting alternative, albeit one that requires carefully designed hardware support.

Conceptually, advisory prefetch is simple. We add an additional instruction to the architecture – `prefetch` $\langle expr \rangle$. It initiates a fetch of the cache line containing $\langle expr \rangle$ into cache. Barring resource constraints, this allows the compiler to overlap execution with the time required for the fetch operation – a clear win. The principal drawback of an advisory prefetch scheme is that it requires modifications to the hardware.

Researchers have shown that even simple schemes for inserting prefetch instructions can be surprisingly effective [6, 1]. The decreased time spent waiting for a fetch to complete often more than compensates for the additional instruction issue slots required. A corresponding `flush` instruction might be used to control replacement when there is a mismatch between the hardware-enforced policy and the actual reference patterns.[1]

2.4 Assessment

Each of these techniques addresses the problem of mapping program locality onto hardware cache structures. Each has strengths; each has weaknesses. Compilers should use these techniques whenever possible; the payoffs are large enough to compensate for the extra complexity in the compiler.

Nonetheless, questions remain about the effectiveness of these techniques on production codes. The examples shown in papers and talks are almost all small loop kernels. The extent to which these techniques can improve production applications remains to be seen.[2] Today, we are basically limited to improving the memory performance of simple loop nests.

[1] Remember, LRU replacement is not necessarily what we want. It is simply a compromise based on statistical properties of programs. For a specific program, other replacement policies may work better.

[2] A particular concern is that real loops reference more aggregate data items that the associativity of real caches. Loop distribution may help in such cases – when it is safe and legal.

3 Trends in hardware

Compilers must target specific architectures. A decade ago, compilers largely ignored specific details of the memory hierarchy. Today, these details are critical. Thus, we should look briefly at trends in the design and construction of memory systems. We will focus on cache issues, but register sets and TLBs are equally important.

Trends in cache design are hard to decipher. New machines have large caches (PA-RISC), small caches (ALPHA & i860), direct-mapped caches (SPARC & PA-RISC), and set-associative caches (RS/6000 & i860). They implement diverse replacement algorithms: pseudo-random (i860XP), pseudo-LRU (i486), and LRU (RS/6000). Some even provide `prefetch` instructions (ALPHA).

Several larger trends do emerge, however. Cache lines are getting longer. Primary caches, in general, become larger across different generations of the same architecture. Many systems are being designed with secondary caches (SPARC-10 & ALPHA-based systems). The one dimension that is not growing is set associativity – there is a fundamental conflict between updating the context required for LRU replacement and increasing associativity. If, in fact, a general trend exists in associative cache design, it is away from real LRU replacement.

Each of these trends is an attempt to increase the likelihood that a given value resides in cache. If small caches are good, larger caches should be better. There is substantial justification for this reasoning. Unfortunately, most of the arguments in favor of these trends ignore one fundamental fact – in many cases, the compiler can analyze reference patterns, recognize broad styles of locality, and provide the hardware with hints.

4 A modest proposal

In the semiconductor industry, it is widely accepted that the number of devices on a single integrated circuit roughly doubles every eighteen months. Naturally, this trend has provoked speculation about how to use the massive number of transistors available on a single chip by the end of the decade. Current trends would move in the direction of larger on-chip caches and integration of multiple chips on a single die.

With all this extra chip real estate available, we propose taking a different approach to designing a memory hierarchy. The trends cited in Section 3 can be viewed as "more of the same." Simply put, larger caches should produce higher hit ratios. Unfortunately, this strategy produces diminishing returns – it requires ever more cache for an ever smaller improvement in the hit ratio. The other trends – longer lines, relaxing replacement policies – simplify the construction of larger caches.

Instead of building larger caches, we propose that new architectures provide hardware support for other kinds of locality. The support for `prefetch` in the ALPHA is a step in this direction. We should take it further. Rather than providing one principal path from main memory to the register set, we should provide several paths with different properties. For example, we should consider

- hardware support for gather and scatter operations between a small local memory and main memory. The success of copying (see Section 2.2) argues for this support.
- a small cache (short lines & large associativity) to support references that exhibit temporal locality without spatial locality, like pointers. Such a cache differs from a register in that it provides address resolution. Analysis of addresses at compile-time *is* imprecise.
- a small, fully-associative cache with compiler-controlled replacement. Self-interference due to alignment is a problem with small caches; it cannot be sidestepped easily. Let the compiler worry about replacement with `flush` instructions.
- support for `prefetch` and `flush` in *each* level of the hierarchy. Such support might include a separate issue slot dedicated to `prefetch` and `flush` operations.
- non-allocating load and store instructions for two reasons: (1) to handle values that exhibit no reuse, and (2) to let programs avoid easily detectable forms of self-interference. For example, the `pfld` operations on the i860XP have proven useful to the Portland Group's compiler [5].

- programmable *fifo*'s like those proposed in Wulf's WM architecture [11]. These can be viewed as sophisticated, programmable gather-scatter operations for non-reused values; an alternative view is that they are tools to let the compiler avoid generating prolog and epilog loops
- hardware support for coherence between these structures, where possible.

Of course, none of these make sense without a smart compiler. The classic primary cache cannot simply disappear, unless we mandate improvements in basic compiler technology. That seems an unlikely scenario.

In short, compilers can discover different styles of locality. Current cache structures support several of these well, but fail to help with others. Further, by forcing all references through a single structure, we decrease the cache's effectiveness on those references where it could help most. We should build hardware that offers different paths from main memory to registers – paths that match the different styles of locality known to a modern compiler. This lets the compiler provide the hardware with contextual information about locality – information encoded in the choice of instructions.

References

[1] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Systems*, April 1991.

[2] Steve Carr. *Memory-Hierarchy Management*. PhD thesis, Rice University, Department of Computer Science, September 1992.

[3] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[4] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architecural Support for Programming Languages and Operating Systems*, April 1991.

[5] Larry Meadows, Steven Nakamoto, and Vincent Schuster. A vectorizing, software pipelining compiler for LIW and superscalar architecture. In *Proceedings of RISC '92*, San Jose, CA, February 1992.

[6] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *SIGPLAN Notices*, 27(9):62–75, September 1992. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[7] Allan Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, Department of Computer Science, May 1989.

[8] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[9] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

[10] Michael Wolfe. More iteration space tiling. In *Proceedings of the Supercomputing '89 Conference*, 1989.

[11] William A. Wulf. The WM computer architecture. *Computer Architecture News*, 16(1):70–84, March 1988.

A Note on Microprocessors and DRAMs

Douglas W. Clark
Digital Equipment Corporation
doug@ad.enet.dec.com

March 3, 1993

The CPU-DRAM performance gap is clearly a problem on the horizon—Amdahl's Law warns us what will happen if we ignore one portion of the computation while trying to speed up the rest.

—Hennessy and Patterson, CA:AQA, 1990

There is great intuitive appeal in the notion that computers made of rapidly-improving processors combined with slowly-improving memory should themselves have a performance improvement rate between the two, possibly sagging toward the memory rate over time. Although John and Dave say that this problem is "on the horizon," it has clearly been with us for a decade at least. Thus it may be informative to look at the brief historical record.

While parallel-processor systems should see this effect, and bandwidth-limited applications should see this effect, in this note I will look only at access-time-limited applications on uniprocessors. I will also look chiefly at hardware issues.

What we'd like is a time-series of microprocessor implementations of the same architecture running the same workload, in order to avoid the confounding of effects. Each microprocessor should be measured in a computer containing contemporaneous DRAMs. If the Workshop Hypothesis is true, we would expect to see the rate of performance improvement of the computers lie somewhere between the rates of improvement of the processors and the DRAMs; perhaps the rate might even decline over time, as memory speed came to dominate processor speed.

The VAX microprocessor family offers an excellent opportunity to observe this phenomenon empirically. Between 1985 and 1992 a series of eight comparable VAX minicomputer/server systems employed four major VAX microprocessor designs in four generations of semiconductor process, plus several "shrinks" of a design into the next-generation process. The constant workload will be the SPEC benchmarks, vintage '89. Over the seven years, the cycle time of the microprocessors improved almost 20-fold, or at an annual rate of close to 50 percent. The memory subsystem speed barely improved at all during this time (due in part to the steady growth in memory size). Thus the memory access time for the first VAX microprocessor was 2 or 3 cycles, and for the most recent ones, it was more like 60 cycles. But contrary to the Workshop Hypothesis, *the performance of the computers, as measured by SPECmarks, improved at an annual rate of 75 percent.*

Here's another piece of data from two non-microprocessor VAXes I studied during the Reagan era. The VAX-11/780 had a cycle time of 200 nanoseconds, and took 6 cycles to access main memory on a cache miss. The more recent VAX 8800 had a cycle time of 45 nanoseconds and took 18 cycles to access memory. Yet in comparative measurements of these machines in timesharing applications, the average number of cycles-per-instruction spent waiting for memory actually *declined.*

What goes on here? Both pieces of evidence—the microprocessor benchmark time series and the big-machine timesharing comparison—seem to refute the hypothesis.

The principal and obvious answer is that the use of caches has more than compensated for the increasing performance gap. Caches, after all, make most accesses to main (DRAM) memory simply *disappear.* In the VAX microprocessor case, the number of cache levels between processor and main memory increased over the seven-year history from zero to two. At the same time, cache size and bandwidth increased, and more sophisticated buffering schemes were used. In the timesharing comparison, the 8800 had a cache 8 times bigger than the 780's, with bigger blocks and better write-buffering.

I suppose one might object to these data on the grounds that VAX is a high-CPI architecture to begin with, and so the Workshop Effect would be seen much more weakly than in a proper modern RISC family. One might further object that the SPEC benchmarks do not seriously challenge the memory system in any case. Neither objection holds water. The memory-waiting component is certainly a smaller portion of average instruction-execution time in a VAX than in a RISC, but remember that the data didn't show *weak* evidence for the Workshop Effect, they showed *contrary* evidence! (Is there a RISC time series that does demonstrate the effect?) And while a few of the SPEC benchmarks do quite well in modest-sized caches, some do not; recall also that the VAX microprocessor history includes machines with trivial caches and one with no cache.

To be sure, if one simply took the microprocessor of 1985 and mapped it directly into the technology of 1992, changing only its cycle time, the result would be a computer that spent nearly all of its time waiting for memory. But technology improvement gives us more than cycle time: it gives us logic density, and more density leads to bigger caches, more levels of cache, fancier buffering schemes, and other organizational improvements.

There are serious issues here, of course. The accretion of cache levels and write buffers adds significantly to the complexity of an implementation, even as it increases performance. Cost may increase too, as expensive board-level secondary and tertiary caches become necessary.

But as the VAX history shows, the Workshop problem, while quite real, has not been so severe as to have required extraordinary efforts in engineering or research. Indeed, in my view the hardest technical challenge over this period at Digital has been the steady reduction of the processor cycle time. Ordinary engineering efforts, chiefly in cache design, have been enough to enable system performance improvement to surpass processor performance improvement. Why will caching not continue to work?

At around the time that DRAMs were first used in main memories, there was a concern about a different performance gap: the one between the access times of memory and disks. Several "gap–filling" technologies were proposed (remember magnetic bubbles? charge-coupled devices?), but the gap was ultimately filled by greatly increasing the size of DRAM memory, thanks to its rapidly improving density and cost. The same thing seems to have happened in the short history of the processor-DRAM performance gap. Perhaps the future will be different, and radical new approaches will be needed. But proponents of this view must explain why, at least in the uniprocessor case, this has not been true in the past.

Compiler Support for Reducing, Hiding and Eliminating Memory Latency

Susan J. Eggers
Department of Computer Science and Engineering
University of Washington

1 Prefetching

Current compiler algorithms for cache-based prefetching can fairly accurately predict cache misses that depend on the configuration of the cache, i.e., capacity and conflict misses, by analyzing spatial and temporal reuse of data. Based on the analysis they only prefetch data that will miss in the cache, never prefetch data that is not used and isolate loop iterations that require prefetched data from those that don't via loop splitting (for example, [7]). They sometimes do less well in prefetching on shared memory machines, because of their greater sensativity to additional memory traffic and shared data invalidations[8].

Future work might include the following:

- Since memory latencies will continue to increase relative to CPU speeds, we should pursue compiler-based, rather than hardware solutions for prefetching. Compiler-based prefetching can better handle long latencies, because of its ability to examine a longer window of both instructions and data. It should be particularly successful, when coupled with new static branch prediction heuristics[2].
- Reduce the uniprocessor cache conflicts induced by prefetching via victim caches and increased associativity.
- Develop special prefetching algorithms for shared data, because shared data incurs misses caused by asynchronous (with respect to the CPU) invalidations. For example, exclusive prefetching of shared data with tight use-def sequences.

2 Code Scheduling with respect to Loads

Current code scheduling technology, whose purpose is to hide load latencies from the CPU, includes: (1) techniques that assume load latencies are fixed (such as load delay slot filling), often coupled with optimizations that separate loads and uses (for example, loop unrolling); and (2) techniques to handle uncertain latencies, such as balanced scheduling[5].

Future work:

- Extend all techniques that fill load delay slots to analyze code across basic block boundaries.
- Use load scheduling heuristics in conjunction with compiler optimizations that increase the size of basic blocks.
- Ditto with smart cache prefetching.
- Continue work on architectures that expose load latencies to the compiler, e.g., processors with multiple hardware contexts, lockup-free caches, load lookahead[1], scoreboarding and other techniques that enable out-of-order execution.

- This (the multiple hardware context part) means pursuing other areas concerned with support for medium-grained parallelism, for example, thread placement algorithms (both compiler and operating systems), fast context switching and medium-grained applications.

3 Eliminating Misses to Shared Data

Current work focuses on (1) reordering memory accesses to eliminate cache misses and global references through loop restructuring (for example, [4]) and loop blocking (for example, [6]); and (2) restructuring shared data directly (for example, [3]).

Future work should continue on all fronts.

- Compiler-based data placement algorithms, both those that reorganize control and restructure data.
- Language constructs that allow programmers to help the compiler.

4 Compiler Platform

All of this work, and work similar to it, would greatly benefit from a compiler platform, that was modular enough to allow easy insertion of new techniques, had good uniprocessor and parallel optimizations and was available to all university researchers. To continue my campaign for this, I'd like to enclose a white paper Jim Larus and I submitted to the 1992 NSF Workshop on Experimental Research.

4.1 Bibliography

These publications are examples of the point made in the text, not a complete list of the area by any means.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, June 1990.

[2] T. Ball and J.R. Larus. Branch prediction for free. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 1993. To appear.

[3] T.E. Jeremiassen and S.J. Eggers. Computing per-process summary side-effect information. In *Preliminary Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, pages 115–122, New Haven CT, August 1992.

[4] Y. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Preliminary Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, pages q1–q15, Santa Clara CA, August 1991.

[5] D.R. Kerns and S.J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 1993. To appear.

[6] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[7] M.S. Lam T.C. Mowry and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[8] D.M. Tullsen and S.J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *20th Annual International Symposium on Computer Architecture*, May 1993. To appear.

A White Paper

Workshop on High Performance Memory Systems

Garth A Gibson

School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

garth.gibson@cs.cmu.edu

Statement of Position

With today's rapidly advancing VLSI and multiprocessor technology, memory system design has become a critically important performance issue. While effective use of memory has been an important focus for computer systems development for thirty years, advances in memory performance greatly lag the rate of increase in system processing speed and the rate of increase of application memory use. Unchecked, this growing performance gap might lead to reluctance on the part of tomorrow's customers to pay for today's investment in new processor technology.

The importance of memory system performance has not been overlooked; many disparate computer systems research efforts have recently launched fresh efforts targeted specifically at mechanisms for decoupling overall system performance from the impact of slower components in the memory hierarchy. However, but for a few notable exceptions, modern systems as important as the much heralded "killer micros" feature simple memory system architectures little different from those of twenty years ago and radically out of balance with their systems' processing capabilities. Now is a good time for us to recognize the central role of memory systems research in tomorrow's systems development, to foster evolutionary and revolutionary research in memory systems, and to encourage established systems development not to neglect memory systems issues. We must do more than identify and overcome a single bottleneck in memory systems; we must install a broad emphasis on aggressive and continued developments at all levels of the memory hierarchy. In particular, concurrency at high systems levels must be brought to bear on all levels of the memory hierarchy by way of optimizing compilers, aggressive prefetching, and memory-sensitive algorithms.

1. Memory System Performance Problems

The basic operation of a processor calls for at least one memory access per instruction. As modern processors drive the cycles per instruction to one (or lower) and reduce cycle time by 40% or more each year [Gelsinger89], it becomes clear that memory bandwidth requirements are rising dramatically. In the context of multiprocessor systems sharing memory, processing speed, and the corresponding aggregate memory bandwidth required, is growing at well over 100% per year [Bell89]

The primary technology allowing cycle times to be reduced so aggressively, the miniaturization of circuit devices, provides an increasing amount of fast memory close to computational circuits. However, the size of this memory remains very limited relative to the size of most program's memory needs. The classical mechanism for providing to programs the appearance of large memory with performance approaching that of the fastest memory is the extensive use of caching across a memory hierarchy of progressively larger and slower technologies: on-chip storage, static RAM, dynamic RAM, and magnetic disk (and sometimes additional levels built of optical disk or optical or magnetic tape) [Burks46].

Memory hierarchies can satisfy rapidly growing processor bandwidth requirements if slower levels of the hierarchy can compensate for their less aggressive performance improvements with rapid capacity increases. This works if cache miss ratios (or equivalently page fault frequencies) at all levels of the hierarchy decrease at the rate that processor memory bandwidth requirements increase.

Unfortunately there are multiple reasons to be concerned that capacity increases are not adequate for the task. Broadly, program memory sizes are getting larger at over 50% per year [Hennessy90]. In many cases this means that program localities are spread over more data. This tends to increase miss ratios; thereby, spending memory hierarchy capacity increases on holding miss ratios constant rather than decreasing them. In one study of file cache miss ratios over time, an increase in average file cache size by a factor of about 15 in 6 years delivered virtually no decrease in miss ratio [Baker91].

In the case of large scientific applications, program size may be largely determined by available computing resources (processors and memory) [Gustafson89]; that is, the goals of these applications are constrained by the availability of memory bandwidth and processing power (my thesis being that the latter is less of a problem).

For scientific computations on data objects whose access patterns are not highly local, the effectiveness of additional memory (at any level of the hierarchy) is even less. Kung has shown that to decrease miss ratios proportionally to a processing power increase of a factor of S requires memory capacity to increase by a factor of S^2 for Gaussian elimination, and by a *power* of S for FFT [Kung86].

The range of important, specific applications whose localities are too large or diverse to benefit from relatively small increases in memory size is broad. Database applications, especially on-line transaction processing systems, frequently access customer data from large data sets in essentially random patterns [Garcia-Molina84].

The problems in the database area are dramatic. Gray has recently found that the database sort benchmark (until recently minutes on database machines and 30 seconds on multiprocessor supercomputers) can be executed on a "killer micro" (200 MHz Alpha, model 7000) in under 10 seconds: 1 second idle waiting on a disk array, 6 seconds stalled waiting for main memory, and 2.5 seconds computing [Gray93]. Of course, if the main memory stall time was greatly reduced, it would certainly expose unoverlapped disk wait time. With applications like this, increasing computation speed by another factor of two might reduce execution time by as little as 12% – Amdahl's law in action. Unchecked, the growing performance gap between processing and memory access might lead to reluctance on the part of tomorrow's customers to pay for today's investment in new processor technology.

2. Research Strategies for Memory Systems

The range of current research that is explicitly or implicitly addressing the memory system performance problem is already broad, though it needs to be both broader and deeper.

The most evolutionary approaches certainly deserve careful examination. Into this class I place memory device research: faster SRAM and DRAM, greater RAM parallelism, faster disk technology, technologies for new levels in the memory hierarchy (perhaps holographic storage or flash EEPROM). While some of these approaches will yield concrete alternatives, the impact on cost is crucial and unclear. I believe that in the absence of impending and ongoing device technology solutions, higher level approaches offer the best chance of keeping up with processor technology.

I am not confident of processor architecture solutions to memory system performance. Compressing the bandwidth needed per instruction (perhaps by a variety of encoding schemes) seems too likely to either slow down the rate of processor cycle time decreases (a big lose for applications relatively insensitive to memory performance) or to slow down the rate that processors apply transformations on data; that is, do work. Certainly latency tolerant architectures, the subject of research and development efforts spanning the last twenty years, deserve continued support. The critical issues here are 1) identifying the minimal functionality that must be provided in commodity processors to achieve latency tolerance and 2) ensuring that sufficient parallelism is available with minimal overhead penalties. I am concerned that latency tolerance requires nimble processors (lightweight state) but that fast processor designs extensively leverage data proximity (heavyweight state).

Not surprisingly, the stress applied to a memory system by an application can vary widely with different but equally correct algorithms. Blocking and tiling of large matrix computations provide this kind of leverage. The key to making this more widely successful is programming paradigms that either work well over a wide range of memory system designs or allow automated specialization to a specific memory system design.

The mechanism that I feel is most promising is aggressive prefetching. If data can be prefetched sufficiently early, application programs may not perceive substantial stalling. Once read latency is dealt with, the issue becomes one of providing enough bandwidth to efficiently satisfy the request stream. Although bandwidth is a substantial problem in non-sequential request streams, increasing device parallelism and interleaving data can be quite effective for a wide range of applications. The major challenges in aggressive prefetching is to break free of readahead, avoiding excessive overhead costs associated with wild-guesses, and automating the extraction of prefetch requests.

I believe that the best paradigm for aggressive prefetching is as hints. In this way the memory system can make dynamic resource availability decisions that simplify complexity and avoid congestion without sacrificing prefetching benefits when adequate resources are available.

Finally, static analysis of program behavior in optimizing compilers is likely to be the vehicle for maximizing the effectiveness of most other memory system improvement techniques. The above mentioned automation of memory-sensitive algorithms and prefetching hint extraction are clearly components of an optimizing compiler's toolkit. But optimizing compilers have the opportunity to be much more effective – data organizations can be manipulated to reduce cache pollution, to increase data uses during cache residency, and to aggressively eliminate state as it becomes no longer useful. The challenge for optimizing compilers may well be to raise the level of understanding of a program's task sufficiently to see appropriate opportunities and to develop cost benefit models powerful enough to trade additional processor cycles and instructions for probable memory system benefits.

3. References

[Baker91]	M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, J. K. Ousterhout, "Measurements of a Distributed File System," *Proc. of the 13th Symp. on Operating Systems Principles*, October 1991, pp 198-212.
[Bell89]	C. Gordon Bell, "Multis: A New Class of Multiprocessor Computers," *Science*, v 228, April 1985, pp 462-467.
[Burks46]	A. W. Burks, H. H. Goldstine, J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," *Papers of John von Neumann*, W. Aspray, A. Burks (eds), MIT Press, 1987, pp 97-146.
[Garcia-Molina84]	H. Garcia-Molina, R. J. Lipton, J. Valdes, "A Massive Memory Machine," *IEEE Trans. on Computers*, v C-33 (5), May 1984, pp 391-399.
[Gelsinger89]	P. P. Gelsinger, P. A. Gargini, G. H. Parker, A. Y. C. Yu, "Microprocessors Circa 2000," *IEEE Spectrum*, October 1989, pp 43-74.
[Gray93]	J. Gray, private communication, DEC San Francisco Research Lab, March 1993.
[Gustafson89]	J. L. Gustafson, "Bridging the Gap Between Amdahl's Law and Sandia-Laboratory's Result," *Communications of the ACM*, v 32, August 1989, pp 1015-1016.
[Hennessy90]	J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.
[Kung86]	H. T. Kung, "Memory Requirements for Balanced Computer Architecture," *Proc. of the 13th Ann. Int. Symp. of Computer Architecture*, June 1986.

John L. Hennessy
Computer Systems Laboratoy
Stanford University
Stanford, CA 94305

The Nature of the Problem

There are actually two separate problems that occur in memory systems: supplying sufficient bandwidth and maintaining low latency of access. In both cases, standard memory technology is not scaling with current processor technology. In the bandwidth case, this is because the density of memory devices continues to grow much faster than the bandwidth into or out of a device. In the case of latency, processor cycle times continue to decrease at a much faster rate than to memory access times. These comments apply for both main memory technology (DRAMs) as well as for on-line storage technology (disks).

Potential Solutions

Bandwidth: while there are likely to be some one-time improvements from this technology (e.g. synchronous DRAMs or RAMBUS), overallthe only long-term solution is to array the devices to increase the memory bandwidth. This is what an a multiple bank interleaved memory system, as well as a RAID both do. It has some disadvantages, since continuing increases in density of memory devices make it more difficult to build a memory system that is wide enough to supply the required bandwidth without making the memory system larger than desired (since the growth in density usually is created by making the devices are deeper rather than wider). Note that this approach is not free--generally increasing bandwidth takes more dollars.

In addition, caching (as well as possibly compression) provide techniques that can decrease the bandwidth requirements for lower levels of the hierarchy. Though they cannot reduce the bandwidth requirements that come directly from the processor.

Software also has a role: blocking, for example, reduces the bandwidth requirements on the memory system. The question is whether this technique is effectively limited to a small class of applications (dense linear algebra).

Two additional developments in processor design will also push the bandwidth requirements. First, there are several new microprocessors that can support more than one memory reference per clock cycle. These machines increase the cache and memory system demands. Second, as miss penalties scale up and CPIs scale down the cost of a blocking miss in terms of lost bandwidth increases. Thus, nonblocking caches, which supply higher bandwidth, are becoming a necessity. Although such caches require processor changes, the major implementation burden falls on the memory system, which must handle multiple outstanding requests.

Latency: I consider this problem to be at least as difficult if not more difficult than the bandwidth problem. The potential contributions from technology towards reducing the latency gap are smaller.

There is no silver bullet here, instead a combination of techniques are required to keep pushing this problem back. Today these solutions include: multilevel cache hierarchies and prefetching. In the future, I expect that other solutions such as multithreading or shared cache (or some other form of processor-memory system overlap) will be needed. Nonblocking caches also improve latency by reducing the miss cost.

Multilevel hierarchies are perhaps one of the most important ideas, since they allow us to design two levels in a hierarchy to address different concerns (access time for the upper level and miss rate for the lower level).

In addition, to improving bandwidth by allowing hits under misses, nonblocking caches also can reduce miss latency. Such a reduction occurs because misses can be overlapped providing the opportunity to pipeline memory requests thus avoiding serialization of requests.

Some form of multiprocessing, whether it be multiple processors sharing parts of a memory hierarchy or a multiple context processor that switches on a miss, will help to hide latency as well. The challenge lies in combining this techniques so that latency is reduced and performance is increased without the requirement for a costly increase in memory and interconnection bandwidth.

Finally, prefetching is one of the most promising ideas. It will probably be more applicable than blocking, though unlike blocking requires hardware support. The range of programs for which compiler directed prefetching can help remains a significant open issue.

Nature and Severity of the Disparity between Modern Microprocessors and Memory Systems

Charlie Hitchcock
Thayer School of Engineering
Dartmouth College

While the growing disparity between processor and memory speeds is cause for concern, there is some good news. Many applications perform well on modern microprocessors with traditional, if larger than ever, caches. Cache designs have benefited from sophisticated tools that track real application data streams, and from multi-level and other advanced caching techniques. Even as cache miss penalites increase, means of increasing hit ratios are created and microprocessors with every higher clock rates are supported.

But there is bad news, too. Not all applications exhibit great temporal and spacial data locality, and some applications have enormous data set sizes, well beyond the capacity of existing cache systems. Many of these applications are scientific calculations that perform calculations of regular streams of data. For such calculations, the latencies involved in fetching data from main memory can be larger than the inherent calculations latencies in the processor, so memory delays dominate. Another set of applications that suffer from memory delays are those that depend on "random" pointer chasing, such as event-driven simulators. Here there is little data reuse and again memory delays can dominate.

The underlying natures of applications programs and DRAMs shape the possible solutions to the problem of processor and memory speed disparity. While "slow" DRAM access times are the main cause of concern, DRAMs also have features which could be exploited to advantage.

The organization of a typical 16M bit DRAM includes 64 internal 256k bit DRAM arrays, each organized as 2k rows by 128 bits per row. Each of these 256k bit arrays uses the same row decoding logic, so they effectively form a 16M bit DRAM with 2k rows of 8k bits per row. Yet the 64 internal arrays could have their own row buffers providing 64 independent active rows. In effect, there is the potential to have many sections of a DRAM be active, either for accessing operands now or to prepare for future accesses, reducing latency and increasing bandwidth. Recent synchronous DRAMs have taken a first step in this direction, providing two "independent" banks of memory on a single die, allowing the overlapping of refreshing, page accessing, and reads and writes. But how to structure DRAMs and computer systems to take advantage of this?

DRAM architecture is not a great academic sandbox. DRAM design remains an extremely expensive and specialized field. New DRAM designs will continue to be dominated by mass market concerns, with innovation applied only when strongly justified (not for experimental purposes). At best, academics can simulate new DRAM architectures, hoping to create structures that have some possibility of being implemented. By contrast, compiler and computer architecture research are furtile academic ground.

Many research efforts are available to build from: improved caching, memory-conscious compiling, data prefetching, advanced RAM archtiectures, cache blocking, etc. Somewhere in a mix of this technology lies better solutions still.

Some Thoughts on Memory System Research

Norman P. Jouppi
Digital Equipment Corporation Western Research Lab
250 University Ave., Palo Alto, CA 94301
jouppi@decwrl.dec.com, phone (415)-617-3305

Problem statement

Memory system performance is becoming the most important factor in processor performance.

Table 1 lists some cache miss times and their cost in terms of instruction execution times. Over the last decade, cycle time has been decreasing much faster than main memory access time. The average number of machine cycles per instruction has also been decreasing dramatically, especially when the transition from CISC machines to RISC machines is included. These two effects are multiplicative and result in tremendous increases in miss cost. For example, a cache miss on a VAX 11/780 only costs 60% of the average instruction execution. Thus even if every instruction on a VAX 11/780 had a cache miss, the machine performance would slow down by only 60%! However, if recent RISC machine like the DECstation 5000/200 has a miss, the cost is over 10 instruction times. Moreover, these trends seem to be continuing, especially the increasing ratio of memory access time to machine cycle time. In the future a cache miss all the way to main memory on a superscalar machine executing two instructions per cycle will cost well over 100 instruction times! Even with careful application of well-known cache design techniques, machines with main memory latencies of over 100 instruction times can easily lose over half of their potential performance to the memory hierarchy. This makes both hardware and software research on advanced memory hierarchies increasingly important.

Machine	cycles per instr. with no misses	cycle time (ns)	mem time (ns)	miss cost (cycles)	miss cost (instr.)
VAX11/780	10.0	200	1200	6	.6
DECstation 5000/200	1.4	40	640	16	11.4
?	0.5	4	280	70	140.0

Table 1: The increasing cost of cache misses

Although multiple-instruction issue machines are a very popular topic of research, cache memory hierarchies are a much more important area of research due to Amdahl's Law. Recent research on cache hierarchies has largely focused on cache consistency issues for multiprocessors. This is a very important area of work, but for at least the next decade the vast majority of computer systems (not embedded systems) will be uniprocessor workstations and PC's. Other factors which discourage cache research are the impression in many people's minds that caches are "old ideas" and not interesting from a research standpoint, even though the base technology assumptions have changed by orders of magnitude since the early research was done. Moreover, there is lots of room for innovative research in memory hierarchies, not only with new hardware techniques, but especially in areas which involve tradeoffs between compilers, hardware, and operating systems.

Current research

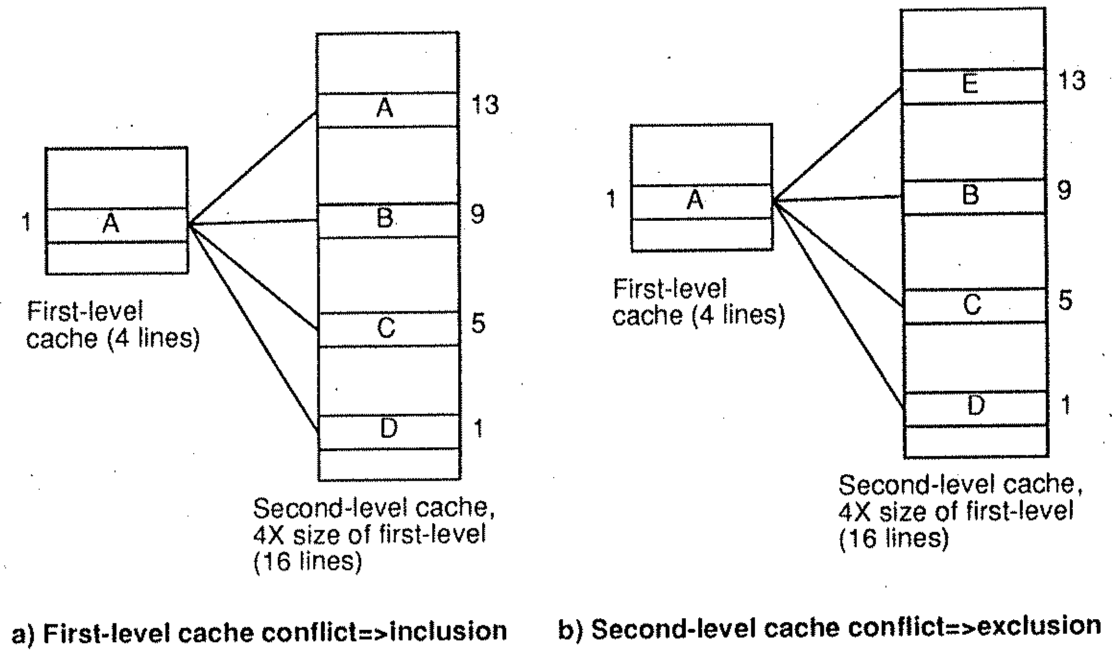
A current area of my research involves tradeoffs in two-level on-chip caching. There are a number of potential advantages of two-level on-chip caching with a mixed second-level cache over single-level on-chip caching. First, primary caches usually need to be split into separate instruction and data caches to support the instruction and data fetch bandwidths of modern processors. Many programs would benefit from data caches that are larger than their instruction caches, while some would benefit from instruction caches that are larger than their data caches. By having a two-level hierarchy on-chip where the majority of the cache capacity is in a mixed second-level cache, programs can allocate the majority of on-chip cache lines either to instructions or data depending on their requirements, as opposed to living with a static partition given by single-level on-chip cache sizes chosen at design time.

A second potential advantage of two-level on-chip caching is an improvement in cache access time. As existing processors with single-level on-chip caching are shrunk to smaller lithographic feature sizes, the die area typically needs to be held constant in order to keep the same number of bonding pads. When processors are initially designed, their on-chip cache access times are usually well matched to their cycle times. If the additional area available due to a process shrink is used to simply extend the first-level cache sizes, the caches will get slower relative to the processor datapath. Instead, if a second-level cache is added on-chip, the primary caches can scale in access time along with the datapath, while additional cache capacity is still added on-chip.

Perhaps the biggest potential disadvantage of two-level on-chip caching is that if the ratio in size between first-level caches and the second-level cache is small, much of the second-level cache will consist of instructions and data which are already in the primary caches. Then, most misses in the primary caches will also miss in the second-level cache. In this situation adding a second-level cache can "get in the way" by adding delay between a first-level cache miss and an off-chip access more than it helps by reducing the off-chip miss rate. In order to mitigate problems of duplication in on-chip multi-level caching a new technique called *exclusive two-level caching* has been developed.

In two-level exclusive caching, when a reference misses in the first level and hits in the second, the contents of the first-level cache line are transferred to the second-level cache while the second-level cache line is refilling the first-level cache. This results in a swap if the current contents of the first-level cache line and the desired contents of the first-level cache line map to the same second-level cache line. When a reference also misses in the second level, the line off-chip is loaded directly into the first level, while the first-level victim is sent to the second level.

Note that exclusive caching only occurs in this system if mapping conflicts occur in the second-level cache. Thus mapping conflicts in the first-level cache that do not conflict in the second-level cache do not have exclusion. Consider Figure 1-a. This system shows an direct-mapped first-level cache with four lines and a direct-mapped second-level cache with 16 lines. If address *A* is referenced, followed by a reference to address *B*, swapping data at address *A* back to the second-level cache will leave the second-level cache unchanged. (If both caches are write-back, then the contents of address *A* in the second-level cache will be updated from the contents of the first, but the address mapping will stay the same.) Similarly, if references are made to addresses *C* and *D*, inclusion will still occur between the first-level cache and the second-level cache.



1
A
First-level cache (4 lines)
A
13
B
9
C
5
D
1
Second-level cache, 4X size of first-level (16 lines)
a) First-level cache conflict=>inclusion
1
A
First-level cache (4 lines)
E
13
B
9
C
5
D
1
Second-level cache, 4X size of first-level (16 lines)
b) Second-level cache conflict=>exclusion

Figure 1: Exclusion vs. inclusion during swapping

Only references that are made to addresses that map to the same second-level cache line create exclusion (see Figure 1-b.) For example, if a reference is made to address *A* which maps to line 13 in the second-level cache, followed by a reference to address *E* which also maps to line 13 in the second-level cache, then both lines *A* and *E* can be stored in the first and second-level caches, while a conventional system could only store either *A* or *E*. If references to *A* and *E* alternate, they will repeatedly exchange places between the first and second-level caches.

Thus exclusive caching has two advantages over conventional replacement policies:

- Conflict misses in the second level cache are reduced since two lines can be present in the first two levels of the hierarchy that map to the same line in the second level cache. This provides a limited form of associativity.
- The capacity of the limited on-chip area is better utilized since there will be less duplication between the contents of the first and second level cache.

We are currently evaluating the performance of two-level exclusive caching for a number of system configurations.

Position Paper: Workshop on Memory Systems

On **I/O and Memory Systems in Massively Parallel Multiprocessors**

Randy H. Katz and Ethan L. Miller
Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

The performance of conventional processors depends critically on the concept of memory hierarchy to hide latency between memory access levels. Hardware designers (and more recently, compiler writers) think of the memory hierarchy extending from the register set to the instruction/data caches to the main memory system. Of course, operating system designers focus on the rest of the hierarchy, from OS-maintained file caches to disk-oriented secondary memory. In large-scale supercomputer centers, the memory hierarchy beyond the disk drives is even more complex, as it incorporates remote disk drives on storage servers connected to tape robots and other exotic tertiary memory technology. We argue that massively parallel machines have too long focused on the issues of local vs. remote memory without adequate attention to the issues of data storage beyond semiconductor memory.

Many commercially available massively parallel systems segregate disk and processing nodes. In other words, a node in a MPP may store data or perform computation, but not both. Examples include the CM-2, CM-5, and the Touchstone Delta. These tend to have large numbers of computation nodes matched to small numbers of I/O nodes, the latter with high bandwidth connections to local disks and the outside world.

This kind of architecture adds complexity to managing the memory hierarchy. Not only is semiconductor memory local and remote, but so is disk memory. For out-of-core computations, data must be staged from disk to the computation nodes before computation can begin. The algorithm developer must carefully consider the process of staging data to and from disk as well as managing the distributed memory.

Contrast the structure of a massively parallel machine with a distributed system of processors on a computer network. The hierarchy spans local and long-latency remote memory. It is not uncommon for software to choose between moving the computation to the data or the data to the computation. If these kinds of algorithms are to find application in MPPs, it will be important that each node be symmetric, with the same kinds of computational, memory, and storage capabilities.

Programs running on massively parallel systems use disk I/O for two main purposes -- checkpoints and as a kind of virtual memory. In the former, disks are used as a write-once medium to store intermediate results that are rarely re-read. In general, checkpoints do not yield very high sustained I/O rates, though burst rates may be high.

In the latter, disks are used to hold data sets that cannot fit in core. In essense, this memory space can be viewed as virtual memory (a concept not supported in CRAY-style supercomputers). A reference to non-resident memory generates a request to a remote CPU, which may be for a page from disk (either local or remote), or a page from local semiconductor memory. The the requesting CPU and the interconnection network, both kinds of requests are almost identical. The sole difference is the latency between the request and the response. For the interconnection network, the added delay is unimportant as long as the link is not blocked while waiting for a response.

As link interconnection speeds increase, the difference between memory and I/O requests will further shrink. In current practice, programmers cleverly place data within memory so that transmission between nodes is efficient, using as few links as possible. Faster link speeds will permit programmers to pay less attention to actual data placement. When combined with a global address space, faster network links will permit each processor to view the entire MPP's memory as a single entity, using virtual memory techniques to access non-local data. Of course, MPP programmers will still continue to need to maximize their use of local memory. The key point is that non-local data resident in another processor's memory and data on-disk can be made transparent.

At the moment, little is known about the interaction between I/O and processor/memory capabilities within MPPs. For example, Amdahl's famous I/O rule states that 1 Mbit/second of I/O bandwidth is required for each per MIPS of CPU. Therefore, a 100 GFLOPs machine (which is just around the corner) will demand 10 GBytes second of I/O bandwidth. How is this to be provided? How can enough physical devices be attached to the processor's interconnect? How can distributed memory system mitigate this demand for growth in I/O capabilities?

Some scientists claim that data sets size grows slightly slower than the number of FLOPS, increasing the demand for I/O. Others claim that the complexity of the calculations will grow, by factors of from two to five, thus I/O demands will not grow linearly with processing power. Once again, scaling is not well understood.

Methods need to be developed for parallelizing an application's I/O. A conventional single I/O stream is replaced by interleaved parallel streams. Thus logically sequential access, and its usual performance-improving mechanisms for caching and prefetching, will be disrupted when the actual stream becomes physically random. Finally, the effects of moving computation to data rather than data to computation are unknown.

Finally, applications developer's will need to consider disk and tape as part of the application's memory hierarchy when solving very large out-of-core problems. For example, some very large systems of dense linear equations (75,000 by 75,000 double precision matrices) have recently been solved on a CM-5 coupled to multiple DataVaults.

Software/Hardware Techniques for Improving the Memory Hierarchy Performance

Monica S. Lam

Computer Systems Laboratory
Stanford University

Abstract

As memory speeds continue to lag behind processor speeds, the need and opportunity for program optimizations shift from minimizing instruction execution cycles to minimizing memory access cycles. Microprocessor systems generally rely on using a memory hierarchy to minimize the average memory access time. Caches, however, tend to perform quite poorly on those applications that operate on large aggregate data structures. An effective solution for managing the memory hierarchy of uniprocessors and shared memory multiprocessors is to complement the generality of hardware cache mechanisms with compiler optimizations targeted to specific aggregate data structures. These software optimizations include locality improving techniques such as blocking, and latency hiding techniques such as software prefetching. Large parallel systems with long remote memory access times both need and can afford more aggressive software optimizations. Locality optimizations are very specific to the high-level structure of the data. While automatic compilation techniques may suffice for dense matrix computations in the future, research into language designs that cleanly expose the high-level design of data structures is necessary.

1 Different Levels of Memory Hierarchy

This paper addresses the problem of how to improve the performance of the memory subsystem for both uniprocessors and parallel systems. Traditionally, compilers are responsible for managing registers, hardware manages the caches, and finally operating systems, with the assistance of hardware, manage the primary memory. This partitioning of responsibility needs to be re-examined as the ratio of memory access and instruction execution time continues to increase. There is less consensus on how to delegate the management of remote memory in parallel systems: Cache-Only Memory Architectures (COMA), such as the KSR machine, cache data at the processor memory level in hardware; the Stanford DASH multiprocessor has a shared address space but does not support caching at the local memory level; finally machines with distributed address spaces rely completely on software to manage the local address space explicitly. While a network of computers may not be the best configuration for parallel computation, they are readily available and represent a resource that should be considered. Naturally, as the scale of a system increases, the scope of applications for which the system is effective will decrease. We envision that future systems will employ all levels of memory hierarchy: registers, caches, local memory, remote memory in a shared address space, and also remote memory in distributed address spaces. Thus, we need to develop a range of techniques to handle all the different levels of the memory hierarchy.

Memory hierarchies can be managed by hardware and/or software. Hardware and software techniques each have their advantages and disadvantages. Hardware cache designs tend to use simple and fast algorithms. For example, caches have a small set associativity and data are manipulated in fixed size units. Cache conflicts and false sharing can cause unnecessary data traffic. In particular, matrix computations can have a very high cache miss rate; the working set of these computations is large, and once a cache conflict occurs, it is often repeated many times because of the regular addressing pattern in these codes. Unless the cache miss penalty is very low, caching alone will not be effective on matrix computations. On the other hand, software can tailor the communication and memory allocation algorithms to a particular application at the cost of higher overhead. However, ensuring data coherence at the software level is difficult for programmers, and so far, only matrix computations are amenable to automatic techniques.

We should choose the best combination of hardware and software techniques for each level of the memory hierarchy. For example, we advocate caches be used for uniprocessors and shared memory multiprocessors. Even though caches may not perform well for matrix code, they are effective for many other applications, many of which cannot be analyzed and optimized by compilers. We can complement the hardware cache algorithm with compiler optimizations. Many of the transformations previously developed for vectorizing and parallelizing compilers can be used to improve the cache performance of scientific code. This combination of software and hardware will help deliver a robust performance across a large suite of applications. For systems with a very long remote memory access time, e.g. workstations farms, explicit software control over the communication and management of the local memories will play a bigger role in these systems. In general, we expect the need for software intervention to increase as the memory hierarchy latency increases.

2 A Research Approach

Much of the need and opportunities to improve the memory performance lies in the manipulation of aggregate data structures. There has been a lot of research on understanding how to parallelize and optimize the memory hierarchy performance on matrix computations. The results indicate that high-level information on the structure of the data and computation is necessary for optimization. For example, if we access a row in a column-major matrix, only one word in each cache line transferred is used; changing the organization of the matrix in this case can significantly improve the program's locality. Trees are another important data/computation structure that has received a lot of attention lately. It is important that we continue to accumulate knowledge on how to design and manipulate data structures in a way that uses the memory hierarchy effectively.

The next challenge is to encapsulate this knowledge in a reusable manner. An effective approach is to embed these optimization algorithms in the implementations of high-level programming languages. The programming language thus becomes an interface with which a novice user can gain access to the optimization algorithms. The array data structures are explicit in many conventional programming languages, and various memory hierarchy optimizations have been implemented in compilers for these machines. Unfortunately, it is very difficult to extract the high-level structure of user-defined data types in programming languages we have today. Future language research should aim to capture the high-level design of the data structures to make memory hierarchy optimization possible.

The rest of the paper will focus on the domain of matrix computations. It is an important domain that does not run well on caches, but it is also a domain for which effective software optimizations exist. We will discuss the range of hardware and software techniques useful for different levels of the memory hierarchy.

There are two facets to optimizing the memory hierarchy performance: the first is to improve a program's data locality, and the second is to optimize the remaining data transfers. Important techniques for improving data locality are data and loop transforms such as loop interchange and blocking. They are useful for all levels of the memory hierarchy, from improving register efficiency on uniprocessors to minimizing communication on message passing machines. These techniques not only reduce the effective memory access latency, but also reduce the bandwidth required between adjacent levels in the hierarchy. To optimize the data transfers, systems of different latencies require different techniques. Software prefetching is particularly promising for optimizing the memory hierarchy on uniprocessors and multiprocessors. On machines with long communication latencies, more explicit control is necessary to obtain an acceptable performance. We have developed algorithms that can compile dense matrix code into optimized communication code on distributed address space machines. As can be expected, our software prefetching algorithm is applicable to many more programs than our distributed memory compiler algorithm.

3 Increasing Locality through Loop and Data Transformations

The goal of loop and data transformations is to increase the number of times a data item at each level of the hierarchy is used before it is displaced. A useful optimization in dense matrix code is to change the organization of a matrix, or the order in which the iterations in a loop are executed. Unimodular transformations (such as loop interchange) try to schedule the computation so that iterations in the innermost loop nest reuse the same data or the same cache line. Blocking (also known as tiling, unroll-and-jam, and stripmine-and-interchange) is a particularly important transform for computation that operates on matrices. Programs that process large matrices a row, or a column, at a time tend to suffer from poor locality between the row, or column, operations. Blocking rewrites the loops so that the program processes the data a sub-block at a time, thus increasing the "dimensionality" of the locality. Once brought into a level of memory hierarchy, a data item is reused multiple times before it is displaced. Blocking thus not only reduces the data fetch latency, it also reduces the required data bandwidth. The technique is applicable to all memory hierarchy levels: registers, caches, local memories on multiprocessors, translation lookaside buffers, secondary storage and remote memories on parallel systems. Blocking for registers has the additional benefit that it reduces the total number of instructions to be executed.

We have developed an algorithm that combines unimodular loop transforms (interchanges, skews and reversals) with blocking to optimize for locality and parallelization in a loop nest[WL91]. We have implemented the algorithm in the SUIF (Stanford University Intermediate Format) compiler. The applicability of data and loop transformations is limited to linear algebra kernels that operate on arrays with at least two dimensions. When loop transformations are applicable, their effect can be very dramatic. Our algorithm is successful in blocking a variety of numerical kernels such as matrix multiplication, LU decomposition (without pivoting), Givens QR decomposition, and successive over-relaxation algorithms. Some performance data are shown in Figure 1. We have recently developed an algorithm that analyzes multiple loop nests and distributes the data and computation across the processors with the objective of minimizing communication while supporting parallelism[AnL93].

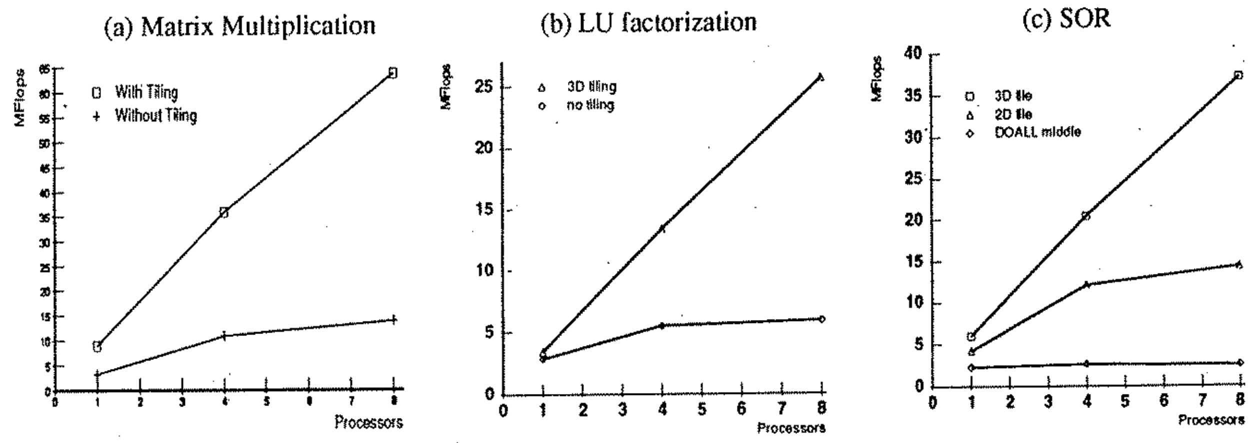


Figure 1: Performance of blocking on an SGI 4D/380 machine: (a) 500 x 500 double precision matrix multiplication, (b) 500 x 500 double precision LU factorization without pivoting, (c) 30 iterations of a 500 x 500 double precision SOR (successive over-relaxation) step.

Most microprocessors have direct-mapped caches or caches with a small degree of associativity. Conflict misses can severely degrade the performance of a cache in scientific computation[LRW90]. Increasing the associativity by a small degree, or using a victim cache, can aid those cases where multiple vector operands happen to map to the same locations. These techniques are inadequate for blocked algorithms where we wish to hold a reasonably large set of data that are separated by a regular stride. Conflict misses in this situation are highly sensitive to the stride. A solution is to copy the submatrix we wish to reuse into a contiguous space. The copying cost is acceptable as the copied data are often reused many times. On parallel systems, it is often useful for the compiler to rearrange the data organization and to map the data used by a processor to its local memory. This type of optimization is particularly important on systems with a high communication cost.

4 Reducing and Hiding the Communication Latency

After reducing the bandwidth requirement between the different levels of the memory hierarchy, we next try to optimize the remaining data accesses. One important technique is to overlap the data accesses with computation on other data. Previous architectural proposals include using vector instructions, data streaming support, decoupling computation from memory accesses, hardware prefetching, software-directed prefetching, and pipelined load/store instructions. We have experimented with software prefetching and found the technique to be very promising. When the data access latency is very long, more aggressive optimizations, such as minimizing extraneous traffic, aggregating messages, and reducing the round-trip data access delays, are necessary. In the following, we first discuss some of our results on software prefetching, then some compiler optimizations for handling long latencies.

4.1 Software prefetching

We advocate that future microprocessors have lock-up free caches, caches that allow multiple outstanding cache misses. This is the basic functionality that must be provided for all latency hiding optimizations. We advocate that the instruction set architecture include explicit software prefetch instructions. The compiler inserts explicit instructions to prefetch a cache line and the cache serves the function of a large data buffer. However, unlike registers or local memories, caches allow their data be accessed by their original physical addresses. The compiler need not determine if the prefetched data are aliased with other data written during the prefetch interval. This is significant because it is often nontrivial or impossible to prove that two accesses are not aliased.

We have developed a prefetch algorithm for both dense and sparse matrix code, implemented the algorithm in our SUIF compiler, and simulated the performance of prefetching across a number of applications[MLG92]. Our algorithm analyzes the data usage in loop nests (using the same analysis we used for blocking) and issues prefetches only for those accesses that are likely to miss. This is important because prefetching incurs both an instruction and a cache access bandwidth overhead; this overhead can negate the benefit of prefetching, if we prefetch indiscriminately. To hide the long memory latency, the algorithm "software pipelines" the prefetches with the computation. That is, the compiler generates a prolog, some steady state code and an epilog. Within the steady state, each iteration of the loop would prefetch the data needed in some future iteration.

On the whole, our evaluation of prefetching indicates that the algorithm is successful in reducing the memory stall time significantly over a wide range of scientific code. Figure 2 shows the results of simulating a collection of scientific programs on a system patterned after the MIPS R4000. The processor runs at a 100 Mhz internal clock. The processor has an 8 Kbytes first level cache and a 256 Kbytes secondary cache. Both caches are direct-mapped and use 32 byte lines. The penalty of missing in the primary cache is 12 cycles, and the penalty of missing in both caches is 75 cycles. The collection of programs includes the kernels of Nasa7 and the Tomcatv program from the SPEC benchmarks, Ocean from the Splash benchmark, and the rest are from the NAS Parallel Benchmark. Note that we have manually changed the alignment of some of the matrices in the Nasa7 and Tomcatv programs to reduce the number of cache conflicts. As discussed above, a small set associativity or data copying to contiguous locations is useful in eliminating conflict misses in the program. Results from experimenting with both loop transforms and prefetching indicate that they are complementary techniques. For example, a loop interchange may enhance the spatial locality of the program, and prefetching can further improve the performance by hiding the latency of the remaining cache misses.

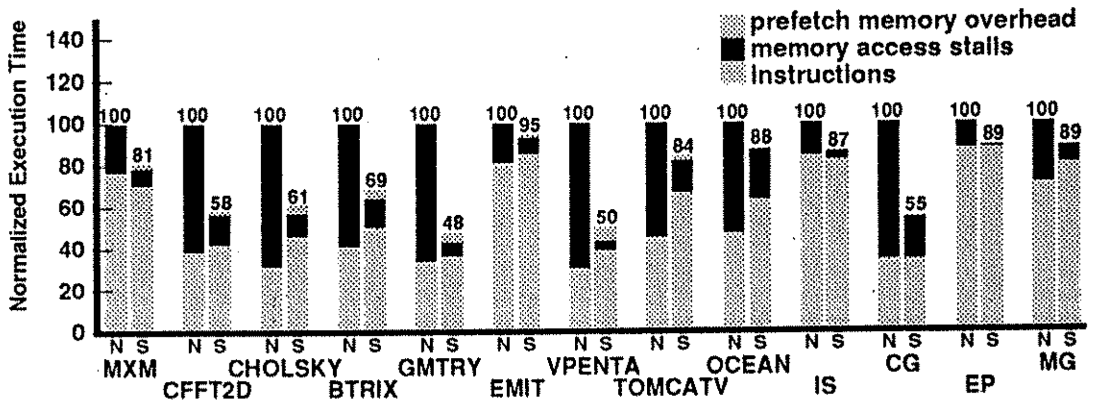


Figure 2: Performance of our software prefetch algorithm (N= no prefetching and S = selective prefetching).

Our results suggest that software prefetching is effective and that complicated hardware support is unwarranted. First, software can exploit the higher level program semantics in calculating the addresses of complicated data access patterns. The data address calculations need not be completely data independent before software prefetching is applicable. Even if the prefetch address depends on an earlier memory access, the prefetch latency can be hidden as long as there are other operations that can execute in parallel. This is much more powerful than alternatives such as vector instructions and hardware prefetching. Second, the overhead of prefetching is tolerable; a compiler can selectively prefetch only data that are likely to miss, and it can use the same set of registers to calculate the addresses for both prefetches and the actual uses. Since software prefetching uses general processor cycles, the effect of the instruction overhead will decrease with faster processor implementations. This is superior to using dedicated hardware with limited generality.

4.2 Hiding Long Latencies

Implicit in the design of software prefetching is the assumption that the data transfer latency is relatively short compared to the amount of computation in a loop. Furthermore, since prefetching initiates a message transfer every cache line, the overhead in initiating a data transfer must necessarily be small for this method to succeed. With long communication latencies, prefetching would degenerate to fetching all input data for the entire loop before any computation can start. Under such circumstances, we need to employ more expensive techniques to solve the problem.

The shared virtual memory approach of caching at the page level is unlikely to perform well on, for example, matrix computations. The scenario of accessing a row of a column-major matrix, as discussed earlier, may cause a page to be brought in for every data accessed. For matrix computations to execute efficiently, software techniques to optimize the data transfers are essential. One important optimization useful for many matrix computations is to statically determine the necessary communication. Instead of having the process that needs the data request the data transfer, the sender initiates the data transfer as soon as the data are available. This technique cuts down the round-trip communication latency by a factor of two. Another important optimization is to amortize the communication overhead by aggregating the data transfer into larger messages.

The algorithm for generating efficient distributed memory code relies on a deeper analysis than that used in either the loop transform algorithm or the software prefetching algorithm. Within the domain of dense matrix computation where the data addresses are affine functions of loop indices, we now have accurate data flow analysis algorithms that relate all the dynamic instances of data accesses to the very instance that produces the value[F91][MAL93]. This analysis provides the information necessary for generating optimized communication code for distributed memory machines[AmL93]. We expect that automatic techniques will be able to handle this, albeit narrow, domain adequately in the future. Cooperation between the programmer and the compiler will expand the scope of the techniques.

5 Concluding Remarks

To optimize the memory hierarchy, we must design hardware and software solutions that cooperate to provide a robust performance for all programs. Hardware caches provide a basic mechanism that can exploit locality of reference in all programs, without any software intervention. While caches do not perform well on computations that operate on large aggregate data structures, such computations are, fortunately, more amenable to compiler analysis and optimizations. We show that matrix computations can run well on uniprocessors and shared memory machines by using the techniques of loop transformations and software prefetching. Systems with long latencies, such as workstation farms, require more sophisticated software techniques. Again, fortunately, the domain of applications that can exploit such parallelism is also narrower and is more susceptible to software optimizations. We believe that this approach of developing combinations of simple hardware support and intelligent software algorithms is the key to solving the memory hierarchy problem.

Acknowledgments

The research results described here were produced jointly with Saman Amarasinghe, Jennifer Anderson, Anoop Gupta, Dror Maydan, Todd Mowry, Ed Rothberg and Michael Wolf. The research has been supported in part by DARPA contracts N00039-91-C-0138 and DABT63-91-K-0003 and an NSF Young Investigator Award.

References

[AmL93]
S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Languages Design and Implementation*, June 1993 (to appear).

[AnL93]
J. A. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Languages Design and Implementation*, June 1993 (to appear).

[LRW90]
M. S. Lam, E. E. Rothberg and M. E. Wolf. 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)*, pp. 63-74, April 1991.

[MAL93]
D. E. Maydan, S. P. Amarasinghe and M. S. Lam. Array Data Flow Analysis and its Use in Array Privatization. In *Proceedings of ACM Conference on Principles of Programming Languages*, pp. 2-15, January 1993.

[F91]
P. Feautrier. Dataflow analysis of array and scalar references. In *International Journal of Parallel Programming*, 20(1):23-52, February 1991.

[MLG92]
T. C. Mowry, M. S. Lam, A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62-73, October 1992

[WL91]
M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Languages Design and Implementation*, pp. 30-44, June 1991.

HIGH-PERFORMANCE MEMORY SYSTEMS

Howard G. Sachs
Executive Vice President, Hardware Systems
Intergraph Corporation

March 1, 1993

ABSTRACT

It has been proposed that the performance of microprocessors has increased by a factor of 50% to 100% per year, and during that time, DRAM has only increased at 10% per year and that this disparity will increase with time causing a bottleneck for further performance improvements in the next decade.

This paper examines the proposed disparity and suggests that large Level 1 Caches and large Level 2 Caches buffer this disparity quite effectively for general-purpose workstation-class uniprocessors, which will offer a 9x performance improvement in the year 2000.

However, these large L1 and L2 Caches allow only a small number of processors to operate on a common or distributed memory system. In order to build systems in the year 2000 that are 100 to 1000 times faster than today's machines, new techniques are required to solve the memory bottleneck. One hardware approach is proposed that seems to offer some promise.

BACKGROUND

Computer architecture has evolved each decade by overcoming bottlenecks (such as packaging, logic density), and now it appears that memories may be limiting performance.

Today (1993), a typical microprocessor architecture could be characterized as follows:

Clock Rate	=	100 MHz
Instructions/Clk (IPC)	=	1
L1 Cache	=	16 KBI/16KBD 2.4% MR/100 instructions
L2 Cache	=	1MB 0.3% MR/100 instructions
L2 Access Time	=	4 clocks
MM Access Time	=	25 clocks

If this machine had no miss in the L1 Cache, the performance would be limited only by the internal logic of the CPU. In reality, the miss rate using SPEC92 integer and floating-point codes runs approximately 2.4% per 100 instructions. So the performance degradation attributed to the L1 Cache is 2.4% * 4 clocks = 0.10 clocks, and the secondary miss rate is 0.3% yielding 0.3% * 25 = .075 clocks. Therefore, the total time attributed to cache misses is 0.10 + .075 = 0.175 clocks, which is a 17.5% performance reduction over our ideal machine with no cache misses. This is not a great penalty, and most architects do not spend much effort to drive this penalty lower.

In order to achieve higher performance than is available in today's uniprocessor systems, many types of multiprocessors have been constructed. For general-purpose applications, I will only consider symmetric multiprocessing (SMP) with a common main memory. There are two general bottlenecks: one in the DRAM memory system and the other being the bus interconnecting the processor and memory system. With today's technology, we could build a bus system that could transfer up to 64 bytes in one 100 ns clock cycle yielding a maximum bandwidth of 640 MB/sec. A one-bank memory system can be built which will achieve 640 MB/sec, and certainly multiple banks can easily deliver the required bandwidth.

Each processor requires:

$$BW = 100\text{ MHz} * 1\text{ IPC} * 0.3\%\text{ MR} * 64\text{ Byte Line} \approx 20\text{ MB/sec per Processor}$$

So, with a 640 MB/sec Bus, we can easily build a system with

$$\frac{640\text{ MB/sec}}{20\text{ MB/sec}} = 32\text{ Processors}$$

However, for supercomputer applications, in the order of 300 processors would be required, not 32. This presents a clear problem for a single bus with 300 processors connected to main memory. The clock rate for 300 processors would be:

$$\frac{300 * 20\text{ MB/sec}}{64\text{ B bus}} \cong 100\text{ MHz}$$

Four- to-five feet of system bus with over 300 discontinuities (caused by the attached microprocessors) would raise significant signal integrity issues if a 10 ns clock were required. The memory system would also require up to 64 banks for a total of 4 GB. Using 4 Mb chips would yield 64 MB or 128 chips per bank for a total of 8192 chips.

This system is not feasible to build today because of the interconnect problems which are exacerbated by the large number of banks required.

PERFORMANCE IN THE YEAR 2000

Process technology, as well as architecture, dramatically affects processor performance. DRAM-driven process technology has doubled the gate density and increased the effective gate speed by 50% every two years over the past decade. In the next ten years, the industry will not be able to keep up this pace because of device physics limitations and huge capital equipment costs. So it is expected that new generations will appear every three years, and we can expect 2x in density and a 50% transistor speed improvement every three years. It should be pointed out that the bandwidth of the interconnect system will be dramatically reduced because the resistance of the interconnect will go up faster than the capacitance goes down. Assuming that architectures will work around the interconnect problems, we can expect clock rates in the range of 225 MHz.

Clearly, higher clock rates can be achieved with techniques such as superpipelining; however, the IPC would be reduced below 1.

Architecture improvements will be as dramatic as process improvements running at 2x every three years, or 4 IPC in the year 2000. Wide-word architectures with up to ten independent functional units will be cost-effective even if their utilization is low. In addition, new compiler techniques, such as predicated execution that reduce branch latencies, will be employed requiring many functional units in order to be efficient. In order to achieve 4 IPC, however, it will be necessary to execute two loads and a store in the same clock cycle, which causes the L1 Cache to be dual-ported and, in essence, 1.5 the size of a single-ported SRAM. Non-blocking loads will become more important, and significant logic will be required, in addition to high bandwidth L2 Caches with synchronous SRAMs.

CACHES

Currently, cache 6T (transistor) cells are about $100\mu^2$ in most logic processes. Special SRAM processes are, of course, much denser. By the year 2000, a cell of $15\mu^2$ in this logic process should be achievable.

If we take the $15\mu^2$ cell and assume a 40% array efficiency, the total memory size per cell would be $37.5\mu^2$. For a 256 KB dual-ported RAM, the area would be

$$(256 \text{ KB} * 37.5\mu^2 * 1.5) = 115 \text{ mm}^2$$

Since the maximum die area is 20mm x 20mm = 400 mm, the 256 KB D-Cache will occupy 115/400 = 29% of the die area. This 256 KB L1 Cache would have a miss rate of 0.6% or less per 100 instructions.

L1-L2 BANDWIDTH CONSIDERATIONS

The average bandwidth the secondary cache sees is calculated as follows:

$$BW = 225 \text{ MHz} * 4 \text{ instructions/clock} = 900\text{M Inst/sec}$$

With a 256 KB L1 Cache, a 0.6% miss rate is to be expected on average per 100 instructions. The bandwidth that is required by the L2 Cache is:

$$BW = 900 \text{ M Inst/sec} * 0.6\% * 64 \text{ Byte line} = 346 \text{ MB/sec}$$

High performance requires a 4- clock latency with a 1-clock synchronous RAM and 256 data lines. As a result of this very low latency, we have a very high bandwidth.

$$BW\ L2 = 256/8 * 225 * 10^6 = 7.2 \text{ GB/sec}$$

This bandwidth is actually 20 times the required 346 MB/sec and, therefore, not a consideration.

WORKSTATION SYSTEM 2000

If we look at our workstation system in the year 2000, it will have the following characteristics:

<u>Micro 2000</u>

Clock Rate	=	225 MHz
IPC	=	4
L1 Cache	=	Dual-ported 256 KB 0.6%/100 units
L2 Cache	=	16 MB .05% miss rate
L2 Access Time	=	4 clocks
MM Access Time	=	50 clocks - 512 MB (72 chips)

Our machine is nine times faster than the 1993 machine, not considering cache misses.

$$\frac{225 \text{ MHz}}{100 \text{ MHz}} * \frac{4\text{IPC}}{1\text{IPC}} = 9$$

Calculating the effects of cache misses on overall performance yields a 20% degradation similar to our 1993 system. The calculations are as follows:

$$0.6\% \text{ L1 MR} * 4 \text{ Clks} = 0.05\% \text{ L2 MR} * 50 \text{ Clks} = .05 \text{ Clks}$$

With an IPC of 4, the CPI = 0.25, so:

$$\frac{.25 + .05}{.25} = 1.20$$

If we now take a look at our Micro 2000 from an SMP perspective, our problems that were unsolvable in 1993 have just gotten worse by 50%. While each Micro 2000 processor is 9x faster than the current model, the large L2 Cache has buffered some of the bandwidth required.

1993 Micro	BW = 100 MHz * 1 IPC * 0.3% MR * 64 ~ 20 MB/sec
Micro 2000	BW = 225 MHz * 4 IPC * 0.05% MR * 64 ~ 30 MB/sec

So the bandwidth required from our memory system in the year 2000 is now 50% greater. Therefore, our interconnect system will require a 6.6 ns bus, definitely requiring a different approach.

POSSIBLE SOLUTIONS

The memory interconnect system using a bus is not a good solution because it does not lend itself to modern chip technology. It requires lots of power and space and many chip I/O pins which are also inefficient. Solutions similar to the RAMBUS™ memory approach probably offer the most promise.

If each Micro 2000 had two wires (one for data and one for address) and all information was transmitted bit serial at high rates ~ 30 MB/sec, each processor's bandwidth would be satisfied easily, and all 300 processors could have their addresses concentrated into one chip to then access a 32-bank memory system. Each bank would be one-to-four 64 Mb DRAM chips. Each DRAM chip could then drive another concentrator chip which would route the data from a DRAM to the appropriate processor.

In the year 2000, chips will be running at 2.5 volts, and if these off-chip signals are low-voltage at ~ 1 volt, we could run dual-rail or perhaps single-ended point-to-point at the 500 MB/sec with little difficulty. The power consumed at these low voltages is minimal and the interconnect system very simple. The concentrator chips will be large high-pin-count devices probably requiring area bumping.

SUMMARY

While initially it appeared that the computer industry would have a problem with the lack of DRAM performance on architectures in the year 2000, it is now clear, with very large 256 KB on-chip L1 Caches and 16 MB L2 Caches, the effects of main memory in the system are negligible. We will, therefore, see factors of 9x in performance at the same prices of today's microprocessor workstations.

There is a serious problem, however, in multiprocessor systems, interfacing a large number of processors to the memory subsystems. The problem is very difficult today and will continue to get worse as processors get faster. The main problem is the interconnect between DRAM and processors.

One approach has been discussed which will solve this problem, but it requires significant research and design effort to perfect.

White Paper for NSF workshop on High Performance Memory Systems
April 12-13, 1993

Steve Scott, Cray Research

There is no getting around the facts that (1) large memory access times are increasing relative to processor speeds and (2), as parallelism increases, communication times are increasing relative to processor speeds. We should do the best we can to slow the widening of these speed gaps, but basic physics imply that they will continue to grow.

So, we have two choices for dealing with this problem. The first is to build memory hierarchies that exploit locality of reference (both spatial and temporal) and massage programs to increase their locality. Programs will perform better with low-latency access to memory, and can depend upon this if they are well structured, because data will typically be found close to the processor in the memory hierarchy. In addition, since the inner layers of the hierarchy (registers, first level cache) service most of the requests, the memory bandwidth provided by the hardware can diminish substantially in the outer layers.

This first approach works very well (at least most of the time) for serial codes, and even works pretty well for some parallel codes. We should continue to push this technique, inserting additional layers (eg. second- or third-level caches) as needed to minimize average memory latency subject to cost constraints. Unfortunately, this approach does *not* work well for many parallel codes, and parallel codes are the only game in town when it comes to future high performance computing. Moreover, even where this technique works, we are slowly losing ground as the speed gaps widen.

The second approach for dealing with the speed gaps is to provide *lots* of bandwidth in hardware, and massage programs to tolerate latency. This can be difficult to do, but bandwidth is what really matters when it comes to sustained performance on many real production codes. In particular, scientific codes often display reference patterns that are not amenable to memory hierarchies due to large strides through memory or very large working sets. Many parallel algorithms have inherent interprocessor communication requirements that also limit the effectiveness of memory hierarchies.

At Cray Research, we have always designed systems with an emphasis on the second approach (using the first approach where appropriate). This is clearly the case with our line of parallel vector supercomputers, and is now the case with our MPP efforts as well. The Cray T3D uses a three dimensional torus interconnect to provide very high interprocessor bandwidth, and places a shell around a high performance microprocessor to help it cope with relatively longer delays to global memory. These efforts are being further extended in the second generation MPP now under design.

In light of these approaches, I see several avenues of research that should play a critical role in the development of future computing platforms that cope well with the widening speed gaps.

First, compilers: keep up the good work. In the past decade, we have seen significant advances in the capabilities of parallel compilers, relieving programmers from much of the burden of shaping code to fit the form and function of the machines on which it runs. We will be particularly helped by further advances in the following areas:

* Automatic detection of parallelism (not really the scope of this workshop).
* Loop transformations to reduce the number of memory references per floating point operation. This includes cache-conscious design, and techniques such as unroll-and-jam that allow greater register reuse of operands.

* Prefetch, prefetch, prefetch!

Second, microprocessors: we're not *all* building workstations. While the rapid increase in microprocessor power is impressive, processors are being optimized for use in workstations and bus-based servers, not MPP's. This is certainly understandable, given that the workstation market dwarfs the MPP market, but nevertheless, it is a basic stumbling block to the realization of efficient, powerful, highly parallel computers. Research (and plain engineering in some cases) that would lead to the following would be particularly helpful:

* More address bits, both virtual and physical, to support the very large address spaces required in high-end MPP's.
* More bandwidth! We need pipelined interfaces capable of sustaining bandwidth on the order of a word per flop.
* More control of on-chip caches, including prefetching, invalidation, and external control signals.
* More outstanding loads. Caches should be lock-up free, and loads should be limited only by the number of available target registers (and perhaps not even by that).

Third, system design: don't forget the bandwidth. Recent trends in DRAM design (Rambus, RAM Link) are encouraging. It appears that the need for additional DRAM bandwidth is finally overcoming the inertia of the past. Future research should focus on ways to improve memory bandwidth while attempting to keep latency low. The interconnection network is *critical*. Again, the focus of research must be to increase bandwidth, while at the same time keeping latency low. Interesting topics include the introduction of adaptability and fault tolerance, while not detracting from basic network performance. Another research topic that could help fight the speed gaps is how to provide (and use) cheap, efficient synchronization mechanisms. Finally, there is the issue of cache coherence. We know that coherence can be provided, it's just that it costs us something (design time and complexity, additional hardware, and run time). Recent research has focussed on how we can provide coherence. I would suggest that an equally (if not more) useful topic would be: do we *need* it?, and if so, for what types of applications?

White Paper -

Workshop on High Performance Memory Systems

Alan Jay Smith
Computer Science Division
EECS Department
University of California
Berkeley, CA 94720
smith@cs.berkeley.edu

Introduction

Microprocessor performance is doubling every year or two. The various parts of the computer memory system are not increasing in performance at the same rate. Improvements in design and performance are needed for various aspects of the computer memory system. These aspects are listed below, grouped by category.

Uniprocessor Cache Memories

Current and coming generation uniprocessors will have on-chip caches of 8Kbytes to 64Kbytes. These microprocessors will be highly parallel, will have cycle times of in the neighborhood of 5ns, and will place enormous demands on the first level cache. There are a number of research problems of interest: (a) How to get enough bandwidth out of the cache. (b) How to predict cache performance as a function of cache parameter selection. (Much as has already been done to quantify the effect of line size and associativity.) (c) How to properly implement prefetching so as to minimize the frequency of demand misses. (d) How to best maintain consistency between the first and second level caches. (e) What write policy (write through, copy back, etc.) to use for the first level cache. (f) See if there is any way to improve cache performance for supervisor workloads. (g) Quantify the performance of TLBs as a function of their design (associativity, page size, TLB size). (h) How to generate code that separates as far as possible the request for data (e.g. a load) and the use of the data, so that dependence on cache performance is minimized.

Multiprocessor Cache Memories

There is a steady and persistent shift towards obtaining high performance by building shared memory multiprocessors. The problem with such systems is to maintain cache consistency (coherence). Studies by the author and his students have suggested that for shared bus consistency protocols, little improvement is possible beyond that available from known algorithms. Those studies have also shown that parallel programs that have not been optimized for shared memory caching have very high levels of memory consistency traffic; such programs obtain very poor speedups in multiprocessors because of memory system bottlenecks. We believe that there are some important research problems in this area: (a) Study the effect of recoding applications on the MP cache system performance. (b) Develop guidelines for the coding of MP applications so that they will run on shared memory systems with caches. (c) Design and evaluate practical interconnection schemes and cache consistency algorithms for multiprocessors with large numbers of processors.

File and I/O Systems

The performance of file and I/O systems has been increasing only very slowly, although the capacity of storage devices has been doubling every two to three years. The traditional "access gap" thus gets wider every day. Further, the shift to single user workstations makes it more and more difficult to overlap I/O activity with multiprogramming. These observations suggest a number of significant research problems: (a) Design improved algorithms for disk caching, so as to minimize the probability of I/O wait. (b) Develop algorithms for disk caching in a multiprocessor distributed environment, in which cached copies must be kept consistent. Such systems must also be recoverable after failure, which limits the ability of the cache to indefinitely hold dirty copies of the disk contents. (c) Evaluate the performance of new file system techniques (e.g. RAID, LSFS) in production environments. (d) Develop algorithms for the optimal management of file migration in systems with large optical jukeboxes and/or huge tape libraries. (e) Develop and evaluate algorithms for the management of databases shared among processors in distributed systems.

A Case for Memory Performance Models

David A. Wood
University of Wisconsin

The disparity between processor and memory speeds is a problem that will only be exacerbated by ongoing technology trends. To illustrate the scope of the problem, Table 1 below presents execution time and speedup results for several of the SPEC92 benchmarks[3] for two models of the DECStation 5000, the 5000/125 and 5000/200. These two models have the same speed processor (25Mhz Mips R3000) and same size caches (64 Kbyte split I/D) but have different main memory access times: 400ns on the 5000/200 and 800ns on the 5000/125[1]. Thus comparing the slower Model 125 with the faster Model 200 will illustrate the effects of the anticipated technology changes. Table 1 shows that the slower memory makes a significant difference: speedups range from 0.72 to 0.87 for these well-known benchmarks, with a geometric mean of 0.78 (the speedups are less than one, since the Model 125 is slower than the Model 200).

If processor speed increases at 50% per year and DRAM speed increases at only 10% per year, then the ratio of processor speed to memory speed will *double* every 2.25 years. After 6 years, the performance of these benchmarks will only be *half* what it would have been had memory speed increased proportionally to the processor—all else being equal. Of course computer architects won't sit idle and some of this difference will be mitigated through standard techniques such as multi-level caches and prefetching, as well as more aggressive techniques such as lock-up free caches. However, I believe that hardware remedies are insufficient; software assistance is necessary to fully deal with the problem.

To support this view, Table 2 presents results comparing restructured versions of the benchmarks above with the originals. This subset of the SPEC92 benchmarks were modified to improve their cache performance using CPROF[2], a cache profiler we have developed that annotates source lines and data structures with the corresponding number of cache misses. These results show that simple cache-conscious changes to the application programs can result in major performance improvements: `vpenta`, one of the kernels in the `dnasa7` benchmark, achieves a speedup of 3.44 on a DECstation 5000/240. Over all applications, the speedups ranged from 1.03 to 3.44 on a DECstation 5000/240, using program modifications that included array merging, padding and aligning structures, structure and array packing, loop merging, and blocking.

As the ratio of processor speed to memory speed increases, the importance of cache-conscious programming also increases. Table 1 clearly shows that our modifications have greater effect on the 5000/125 and 5000/240 than on the 5000/200, due to the greater miss penalties of these machines. However, while cache profiling can help improve cache performance and reduce execution time, it only provides information about the memory system performance of a particular machine. Optimizations made for one implementation may in fact hurt performance on another.

These results indicate that the problems caused by the speed gap should be addressed by software and hardware together. I believe the solution lies in defining *memory performance models* that abstract away the details of particular implementations, yet capture essential features such as temporal and spatial locality. In addition, the models should provide annotations to allow programmers and compilers to identify access patterns that the hardware can exploit. By defining a performance contract between software and hardware, programmers and compilers can predict when `loads` and `stores` will be fast or slow, allowing them to optimize their code accordingly. Conversely, the models identify the cases that computer architects and hardware designers should spend time optimizing.

As a concrete example, consider the Check_in/Check_out (CICO) model proposed for Cooperative Shared Memory[1]. The CICO model tries to identify data sharing patterns and exploit temporal locality for shared-memory multiprocessors. Programmers bracket data usage with **check_out** and **check_in** annotations, to

[1]The 5000/200 also has a deeper writebuffer than the 5000/125

	Machine		
Program	5000/200	5000/125	
	Seconds	Seconds	Speedup
compress	20.42	25.80	0.79
dnasa7	904.74	1213.72	0.75
eqntott	58.70	67.56	0.87
spice	1762.34	2242.10	0.79
tomcatv	160.24	221.04	0.72
xlisp	286.56	385.24	0.74

Table 1: Execution time speedup for SPEC benchmarks

identify when they expect to begin using a datum and when they believe they are done with it. By using CICO annotations, programmers can reason about temporal locality and sharing patterns, thereby reducing communication. In addition, hardware can use these annotations as directives to improve performance. For example, the `check_in` directive causes the hardware to flush a block from the cache, so that another processor finds it in main memory.

I believe that memory performance models must be generalized to capture not only temporal locality, but lack of temporal locality, spatial locality, and structured access patterns. For example, many scientific codes reference large matrices using highly structured access patterns. A memory performance model should not only encourage programmers and compilers to increase temporal and spatial locality (e.g., using blocking), but permit annotations to identify data that will not exhibit locality, allowing the hardware to prevent cache pollution. Similarly, the model should support annotations to identify structured access patterns, such as sequential, strided, and block-strided patterns, so that aggressive hardware may prefetch it.

Many machines have provided directives and other mechanisms for improving memory system performance. Many programmers and compiler writers have worked on optimizing memory system performance. What is needed is a common memory performance model, so that programmers, compiler writers, and computer architects have a common interface.

References

[1] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, July 1992. Also appears as University of Wisconsin, Computer Sciences Department Technical Report #1096.

[2] Alvin Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study, 1993. Submitted to IEEE Computer.

[3] Joseph Uniejewski. SPEC Benchmark Suite: Designed for Today's Advanced Systems. *SPEC Newsletter*, 1(1), Fall 1989.

Execution Time Speedup							
	Machine						
Program	5000/125		5000/200		5000/240		Modification
	Seconds	Speedup	Seconds	Speedup	Seconds	Speedup	
btrix	144.06		114.14		82.52		original
	109.50	1.32	90.20	1.27	55.94	1.48	loop merging
cholesky	188.90		140.98		97.14		original
	162.16	1.16	124.88	1.13	73.66	1.32	transpose array
compress	25.80		20.42		20.38		original
	23.48	1.10	18.98	1.08	17.66	1.15	merged key and value arrays
dnasa7	1213.72		904.74		815.22		original
eqntott	67.56		58.70		39.96		original
	60.98	1.11	55.40	1.06	38.92	1.03	changed short to char
gmtry	177.06		142.04		128.42		original
	119.78	1.48	95.88	1.48	50.92	2.52	swap indicies
matrix multiply	248.44		184.62		91.36		naive
	122.06	2.04	106.16	1.74	66.08	1.38	SPEC column blocked
spice	2242.10		1762.34		1557.90		original
	1781.72	1.26	1406.04	1.25	1163.42	1.34	merged pointer and number
tomcatv	221.04		160.24		140.08		original
	166.16	1.33	132.56	1.21	93.60	1.50	merged arrays X and Y
	150.96	1.46	125.34	1.28	88.76	1.58	+loop merging
vpenta	264.78		169.76		205.96		original
	126.38	2.10	91.74	1.85	70.48	2.92	merged arrays and swap indices
	104.54	2.53	79.44	2.14	59.80	3.44	+loop merging
xlisp	385.24		286.56		205.72		original
	361.96	1.06	277.18	1.03	190.30	1.08	padded node to 16 bytes

Table 2: Execution time speedup after cache profiling

Not Only Is DRAM *Slow*, It Isn't Even RAM

Wm. A. Wulf
University of Virginia

It is widely recognized that there is a growing disparity between the speed of microprocessors and that of affordable (and cool) DRAM.

Less widely discussed is the fact that DRAM isn't really "random" anymore; the cost to access elements is not uniform. The potential concurrency in multibank memory systems is a familiar example of non-uniform access costs, but newer memory *components* also exhibit nonuniform cost. Modern DRAMs have increasingly sophisticated "page mode", "burst mode", and other features that make the cost sensitive to the prior history of accesses.

It would appear that the trend is toward greater sophistication, and hence even greater sensitivity to the order of requests. In the new JEDEC standard for synchronous DRAM, for example, the cost differs by a factor of ten. RAMBUS has, in effect, two cache lines on chip and is best used in burst mode. Other, more exotic proposals are in the wings.

To take full advantage of these components, one must adapt the pattern of accesses to account for the nature of the non-uniorm costs (mostly this means maximizing the locality). There are a number of obvious ways to do this — for example simply increasing the line size of a cache increases the number of references to the same page in a page-mode DRAM and thus amortizes the initial page-miss cost. In the past the relation between line size and total cache size has been studied, but the effect of nonuniform access needs to be factored into these studies.

Of course, not all programs exhibit the sort of locality that benefits from a cache — scientific vector and string processing being the prime examples. Fortunately, these are also the style of computations in which it is possible at compile time to predict the pattern of accesses. It is possible to use this information to improve performance substantially.

For example, one can simply unroll a vector loop and group access to the same vector; this will increase the probability of consecutive accesses to the same DRAM page, and thus amortize the page miss cost. The thesis of one of my student's[1] provides an encyclopedic analysis of what can be done at compile time to exploit non-uniformity.

Compile time exploitation of current components is, unfortunately, limited in a number of ways:

— the lack of run-time information such as precise alignment, and
— a limited number of registers in which to buffer data

We at Virginia are therefore looking into mixed compile-time/run-time approaches. At compile time we detect the pattern of vector accesses; this information is then passed to a run-time mechanism that reorders the accesses to optimize memory performance while buffering data so that the processor can continue to access operands in an order that is natural to the algorithm. Simulation results to date indicate that this scheme can consistently deliver over 90% of the effective peak bandwidth of a memory system.

[1] Steve Moyer, "Access Ordering and Effective Memory Bandwidth", 1993

Potential Research Areas in Memory-Conscious System Organization and Compiler Support for Multiprocessors

Pen-Chung Yew
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

The following is a short list of potential research areas in high-performance memory systems for multiprocessors.

(I) Clustered Memory Systems

In addition to the challenge posed by ever-increasing speed differentials between processors and memories, researchers are facing the physical constraints imposed by the packaging technologies. Those technologies usually impose an inherent hierarchical structure on a large system. For example, an MCM module may accommodate several processors that form a small cluster, a multi-layer mother board may hold several such clusters, and several such boards may be contained in a chassis. Signal latency across packaging boundaries is significantly longer than the latency within a boundary. Also, because of the pin and wire constraint, the bandwidth that could be provided within a package is much larger than that between packages. In view of such packaging constraints, it is often more practical to use different interconnection schemes at different levels than to choose a single scheme for the entire system. For example, within a cluster we may use a crossbar switch or a fast bus, because scalability is not an issue there. Among clusters, we can then use more sophisticated interconnection networks such as high-dimensional meshes or multistage networks.

Interestingly, in most application programs, we can observe a similar hierarchical structure. They can be structured into micro-tasks, macro-tasks, and heavy-weight processes. The communication bandwidth required within a task is much larger than that required between tasks or processes. This program structure maps very well on the physical hierarchy mentioned above; that is, we can schedule a micro-task on a processor or a macro-task on a cluster.

The following describes a few of the many interesting research problems related to such a hierarchy.

(A) How do we design a memory system that reflects such a hierarchy in physical layout and program structure? We may include a cluster memory shared by processors within a cluster, followed by a global memory shared by processor clusters at the next level. Different cache/memory coherent protocols can be used at different memory levels, depending on the interconnection schemes chosen. For example, a snooping cache protocol or a shared cache can be used in a cluster if we use a fast bus or a crossbar switch in the cluster. A directory scheme or a software coherence scheme can be used for the global memory if an interconnection network is used. Some research prototypes like Dash and Cedar have already addressed some of these issues. Several recent commercial machines such as KSR and Convex MPP have also been built with hierarchical structures, even though they are addressing different issues in their designs.

(B) How can the compiler and language support such a hierarchy? Should we make this memory hierarchy visible through the language to compiler and application programmers, so they can take advantage of such a hierarchy? HPF allows data alignment and distribution to be specified, but it does not provide the notion of memory hierarchy. It is not easy to specify a clusterly-shared variable vs. a globally-shared variable using HPF.

(C) Can algorithms and applications take advantage of such a hierarchical structure? Some preliminary studies on Cedar seem to indicate this possibility. However, more studies are needed in this area.

(II) Parallel Program Memory Behavioral Study

We still know very little about the memory access behavior of parallel programs in general. We have seen some parallel benchmarks like the Perfect Benchmarks, Splash, and NAS benchmarks emerging in the last few years. However, it requires a tremendous effort to port these codes to a variety of machines because many of them were originally targeted for one particular machine organization. It would be very useful if we could pool different versions of the same benchmark suite when they are ported to machines with different organizations. We can then compare the program behavior change caused by different machine and memory organizations.

We still know very little about the amount of data sharing between tasks, the data sharing patterns, the synchronization behavior, and the amount of parallelism that exists at different levels of task granularity in parallel programs. Defining these characteristics will be very useful in helping us to design cache coherence protocols, synchronization primitives, data prefetching techniques, and other latency reduction and locality enhancement strategies.

The main difficulty here is to obtain a large suite of parallel programs that more or less represent "typical" application codes being run on parallel machines. We then have to port these codes to different machines and tune them for good performance. This effort requires coordination and cooperation within the research community. An effort like SPEC or the Perfect Benchmarks within the research community will be very useful. However, the focus here is not on benchmarking, but rather to provide a common application suite for system studies.

(III) Compiler and Language Support

There are generally two approaches to reduce the penalty caused by the memory latency: (a) exploit locality and use fast cache/local memories; (b) hide latency using prefetching, multithreading, and other techniques. The effectiveness of each approach needs to be studied further. The compiler techniques to enhance program locality for cache and local memories have been studied quite extensively. Data distribution through language directives also has been proposed and studied. However, the compiler techniques for latency hiding schemes are just beginning to be investigated. Many of these techniques are suggested without real implementation and without performance data from real application codes. More such experimental studies are needed in this area.

(IV) Development of Methodology for Performance Measurement on High Performance Memory Systems

The methodology of performance measurement on parallel machines is still being developed. Because of the potentially large amount of traces and performance data that can be obtained from such empirical studies, more sophisticated measurement tools and techniques need to be developed.

To date, many software and hardware performance monitoring tools have been developed on some research prototypes such as Cedar and Dash. However, many of those tools have been developed by the "tool builders," not necessarily by the "tool users." It is very important to evaluate various performance measurement techniques and to develop more efficient and effective methodologies.