# A Retrospective on
# The MIT Alewife Machine: Architecture and Performance

Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology

The MIT Alewife project evolved out of exploratory work at Stanford on directory schemes for cache coherence [1] (also included in this issue). Using data from small bus-based multiprocessors, this early work demonstrated that directory schemes were as efficient as bus-based snooping protocols, and that by distributing directories along with main memory, they could provide the foundations for a cache-coherent shared-memory multiprocessor based on an interconnection network. This paper further recognized the scaling limits of bit-vector directories – they consumed memory proportional to the square of the number of processors – and speculated that variants such as limited pointer directories or limited broadcast directories[1] might be attractive scalable alternatives. The paper, however, stopped short of demonstrating the feasibility of limited directories, largely because of the lack of either address traces or parallel programs written for a scalable coherent shared-memory system. This lack of data was not surprising given that such a machine had not been invented yet!

**Exploration** The Alewife project was born out of a desire to build a *shared-memory* multiprocessor that was truly *scalable* (see the section "Perspectives and Summary" in the Alewife paper in the Proceedings of the Workshop on Scalable Shared Memory Multiprocessors, Kluwer Academic Publishers, 1991, to get a sense of our early thinking). Although scalable message-passing multicomputers had been around for years, they were known to be notoriously hard to program. We believed that shared memory was easier to program, and accordingly, we chose early on to offer no compromise on the shared memory programming model.[2] Notice that our early Alewife thinking offered no plans to expose message passing to the software system.

For scalability, we chose to borrow heavily from message passing machines conceived by researchers such as Seitz and Dally. Message passing machines achieved their scalability by distributing constant per-processor resources over a point-to-point interconnect and exposing this distribution to the programmer. Accordingly, we decided early on to distribute memory and processors over a point-to-point mesh network (as opposed to a uniform-access multistage network) and strove to keep per-node costs more or less constant. We believed that scaling to even tens of processors required support for locality management from schedulers and compilers. As we discovered years later, (for example, see Nussbaum's PhD thesis), software management of interconnect locality in a cache-based system became important only for systems that exceeded many hundreds of processors.[3] We also learned later that the real benefit of mesh networks for few tens of processors was their low cost, modularity, and ease of packaging.

When the project started, we believed limited directories offered the solution to scalable memory requirements, since their per-processor costs scaled as the log of the number of processors. The discussion below tells why our initial optimism with regard to limited directories was completely misplaced, and how Alewife avoids their deficiencies.

Adhering to our shared-memory programming discipline, we wanted to avoid exposing locality to the programmer at all costs. Alewife does expose locality to the software system, and we believed we could develop limited-directory-based hardware and an accompanying software system that could exploit the underlying locality for scalable performance, while providing an uncompromising shared memory abstraction to the programmer. Our challenge was to build such a system and to demonstrate that the twin goals of programmability and scalability could be met.

Notice that in the very early Alewife days (Spring, 1988), the following features were chosen: a point-to-point mesh interconnect that exposed locality to the software, a limited directory distributed with memory among the processing nodes, and a shared-memory programming model. Features such as context switching, fine-grain synchronization, message passing, and in fact, the name Alewife itself, came later – each with an interesting history of its own. Our early research addressed two major questions, both related to scalability. Could shared memory systems tolerate the latencies of mesh interconnects? Could limited directories scale?

Although architects today do not think twice about using networks with non-uniform communication latency for shared memory, interestingly, we spent a lot of time worrying about their "programmability," that is, whether their non-uniform latency and bandwidth were mismatched with the demands of the uniform-access shared memory abstraction. Anecdotally, we were able to find some email from Agarwal to Hennessy in April of 1988 that talks of the tradeoffs in using a mesh network: "There are certainly a lot of problems [with mesh networks], and perhaps the main one is the issue of programmability. But if we are to get anywhere building large machines, this issue of locality (proximity) must be made visible to the compiler/scheduler (or to the programmer as in message passing machines or connection machines) in some graceful way."

Even more interestingly, the multithreading solution to the latency problem adopted by Alewife was inspired by the following response from Hennessy two days later: "When thinking about how to scale a shared memory machine above a few hundred processors, the difficulty becomes tolerating the latency. I tried to

[1] A limited pointer directory maintains pointers to a fixed number of cached copies of data. A limited broadcast directory divides the processors into sets, and maintains a pointer to each set of processors, sending broadcast invalidations to the entire set when needed.

[2] During the early Alewife days, the notion of shared memory with weaker memory semantics had not yet been formally defined. Therefore, we took sequentially consistent shared memory as a given. As discussed later, we chose to use context switching as a way of tolerating latency. When the weaker models began to appear in a sequence of path-breaking papers from USC, Wisconsin, and Stanford – all three are included in this issue – we were faced with the choice of adopting a weaker model. At this point, we decided to support the sequentially consistent memory model since it did not require compromising the shared memory programming abstraction, and since our investigations revealed that the performance of weak consistency was comparable to other forms of latency tolerance.

[3] A key reason is that even with efficiently engineered interfaces, a significant component of the remote access latency is attributable to overheads at the source and destination.

think how the message passing folks deal with their horrible latencies and it occurred to me - they context switch. Suppose you could build a machine that could context switch quickly (easy for a MIPS-style RISC machine, just use multiple register sets). Suppose you knew when a memory request would take a long time - simply context switch."

We chose to use context switching on cache misses as a mechanism to tolerate the latencies of mesh networks. We also began talking to Bert Halstead at MIT, whose group was exploring the design of a multithreaded processor called March. Like HEP, March used fine-grain multithreading (context switching on each cycle) to tolerate memory latency. March also included tag support for Multilisp Futures and fine-grain synchronization. Alewife's processor, Sparcle, (initially named April, for it followed March!), inherited many of the features of March, and improved upon it in many ways.[4] We believed single thread performance was key to the competitiveness of multithreaded processors. Accordingly, Sparcle context switched only on cache misses to remote memory and synchronization failures (both large latency events). Infrequent context switches allowed the architecture to exploit traditional pipeline optimizations for good single thread performance. It also enabled a simple implementation of Sparcle, since infrequent context switches are tolerant to relatively long context switch times (about 10 cycles). Taking a minimalist approach, we also simplified the tag support architecture for fine-grain synchronization.

Realizing that building a multiprocessor system was a massive effort, we began to explore potential collaborations that could reduce our own effort. The first of such collaborations was with LSI Logic. We realized in the Summer of 1989 that the Sun Microsystems' SPARC architecture could yield a simple path to implementing the Sparcle processor (Sparcle was actually named following our decision to use SPARCs). We met with Gene Hill of LSI Logic, then the head of the SPARC division, and he agreed to help us implement Sparcle by modifying LSI's SPARC implementation. A fruitful collaboration with LSI and Sun followed this initial discussion.

We explored a collaboration with Tom Knight on interconnects. Knight was interested in developing a high-speed circuit-switched multistage interconnect, including a packaging technology using "fuzz button" pressure connectors in a liquid cooled system. Given our focus on communication locality, Knight offered to provide short-circuit feedback paths in the multistage interconnect to support fast near-neighbor communication in the multistage network.[5] We ultimately decided to stick with the mesh interconnect for many reasons. We felt the interconnect technology introduced too many additional failure modes into what was already an ambitious and risky project. The mesh network was a better match to Alewife's pedagogy. Early simulation results indicated that packet switching provided better performance than circuit switching for our system parameters. Finally, and perhaps most importantly, we were also able to obtain working Mesh Routing Chips (MRCs) from Chuck Seitz at Caltech, thereby eliminating (or so we thought) a major risk factor.

It turned out that the self-timed protocol of the MRC was both a blessing and a problem. It helped us in that we did not have to worry about clock synchronization across the entire machine. It also allowed us to conduct sensitivity experiments on the Alewife machine by varying the processor clock for the same network speed. These sensitivity experiments were critical in determining the ratios of processor to network clock speed under which either shared memory or message passing was optimal. Their asynchronous nature resulted in some nightmarish testing and debugging problems, turning many Alewife researchers into transmission line hackers. The asynchrony also required some creative test methodologies. Overall, we believe we came out well ahead by using the MRCs, and we are beholden to Chuck Seitz for making them available to us. On the other hand, the Alewife implementors will be very wary of asynchronous logic in the future.

By the Fall of 1989, the Alewife architecture had evolved to the following: Its fast context-switching Sparcle processors would be based on SPARCs and its mesh network would use Caltech MRCs. The Sparcle processor would support fine-grain full/empty bit synchronization. We, however, were beginning to weaken on the limited directory, and message passing had not shown up yet.

## Design

Extensive simulations against address traces for large numbers of processors obtained by running several parallel programs from IBM, MIT, and Stanford during 1989 and 1990 began to lead us to the conclusion that limited directories were simply not robust. Although all programs exhibited predominately limited data sharing, disquietingly, almost every program included at least a small number of widely-shared (but mostly read-only) variables. Initially, we hypothesized compiler and software system passes that would automatically detect such widely shared objects and fix the problem, and thereby obtained fairly positive results out of our simulators by subtracting the effects of these errant references. For example, we believed we could eliminate widely shared variables in barriers by using scalable software combining trees.

Simulator hacks can only take you so far when you have undertaken to build a real working system, so we began to develop the hypothesized software passes needed for widely shared references. Unfortunately, each new program encountered a new type of optimization that had to be performed. The growing list of optimizations made the system extremely fragile, and gradually, our resolve weakened as evidence mounted on the fallibilities of limited directories. As a result of discussions with David James and Gurindar Sohi, we began to explore alternatives such as pointer chains that could still yield a constant cost per node. (Pointer chaining techniques, which were adopted for the IEEE Scalable Coherent Interface, linked cached copies using pointer chains rooted at the home memory node). We were concerned about the long latencies of single pointer chains and the complexities of the doubly linked alternatives. We also looked at purely software-based approaches using traps and software allocation of pointers in garbage-collected heap storage, and discarded them as being too expensive, at least for the processor-memory speed ratios at that time.

During this time of uncertainty in the project, Kubiatowicz had begun to design a system-level message interface so Alewife could perform I/O operations efficiently. It then occurred to us that we could take advantage of software-injected messages and a trap-based processor interface to extend the limited-directory mechanism into software in the rare case that a widely shared item caused a limited directory overflow. By gracefully extending the directory into garbage-collected heap storage and maintaining it as a software hash table with linked lists, we could allow widely shared objects to revert to the software structures. We named this scheme LimitLESS – limited directories locally extended with software support – and designed a unified message abstraction across both the software and the hardware. The LimitLESS scheme was particularly appealing because it enabled building an experimental system that allowed us to vary the number of hardware pointers from five to zero, zero being the all-software case in which all remote memory operations were being handled in software.

We believed that experimenting with the zero pointer case was important because it afforded a system with minimal hardware support for shared memory. As demonstrated by Chaiken in his PhD

[4] Kranz and Nussbaum worked on the March project and later joined Alewife.

[5] The name Alewife itself came up in 1988 during a discussion with Knight. Alewife was a recently constructed station on the red line in Boston's subway system. Knight's interconnect project continued under the name Transit.

thesis the all-software case was only about a factor of two or three off from the hardware case. Kirk Johnson's PhD thesis took the all-software approach one step further and explored the feasibility of a coherent shared memory system called CRL built on top of an efficient message passing substrate. Interestingly, this thesis articulates a key benefit of interrupt-driven delivery of messages: interrupts are better than polling when asynchronous messages invoke handlers that are unrelated to the computations being performed on the receiving processors.

Although the software-based LimitLESS approach had been conceived to solve the scalability problem of directories, we learned soon enough that it had other appealing properties such as flexibility and adaptability. Chaiken's thesis discusses several such adaptive protocols – for example, those that switch between individual invalidates and software broadcasts – and presents experimental data on their performance. Our instinct about the all-software case proved to be abundantly true as the flexibility and scalability of software approaches have all but shut out hardware directory approaches in more recent research projects.

The LimitLESS case study highlights perhaps the two most important reason for building real systems in research environments. First, unlike simulators, real working systems seldom hide serious flaws. And second, when a research group has undertaken to build a novel system, they will invent the necessary mechanism and do whatever it takes to make it work. Such an environment of necessity that breeds invention is impossible to simulate.

The messaging interface changed the face of Alewife and rapidly established the value of integrating both messaging and shared memory. Recall, during the early Alewife days, shared memory and cheap messages were provided largely in exclusion in previous systems. Even in Alewife, messages were first introduced as a means of performing efficient I/O. They were then extended to provide the foundations for a software-based directory architecture. The message interface also allowed cost-effective solutions to deadlock problems caused by limited buffering in the network hardware. As the message mechanism was exposed to system software it rapidly pervaded the runtime system, since many operations such as scheduling and synchronization were best performed with messages.

Since the initial message interface was available only at system level, user-level software incurred a heavy overhead in using messages. The software folks campaigned for the same functionality to be available at user level, and convinced the architects to provide a user-level message send. Although it may seem that implementing both takes a kitchen-sink approach to the architecture, it turns out that shared memory systems require much of the underlying hardware functionality anyway. The additional requirements are to expose this functionality to the software. Kubiatowicz's thesis has a solid analysis of the extent to which resources can be shared between shared memory and message passing.

**Implementation and Evaluation** We decided to use an application-specific integrated circuit (ASIC) for the Alewife cache and memory management unit (CMMU). This chip provided most of the hardware support for messaging and coherence. At this point we discovered that ASIC vendors were hardly tripping over each other to obtain our ASIC business. In fact, most ASIC vendors will not support a research chip project even with a high NRE (non-recurring expense – a one-time charge paid to the vendor) because the production chip volumes are usually quite low compared to the numbers they are used to. Continuing our relationship with LSI, we established contact with Brian Halla, then the general manager of the ASIC division at LSI (currently CEO of National), who graciously agreed to support our chip building efforts.

The chip design involved writing more simulators. The early phases of Alewife involved trace-driven simulators. These were replaced by ASIM, an instruction-level simulator. As the CMMU chip design started, ASIM was itself replaced by NWO (which stood for New World Order), which was faithful to the real design. NWO, in fact, incorporated some of the control logic directly from the real design. Testing and validation of the CMMU was done using LSI's simulators, augmented with a TCP interface to NWO. NWO also ran on Thinking Machine's CM-5, a configuration that facilitated software validations and architectural studies.

Looking back, the implementation effort was a process of incremental discovery in itself, and we happened upon several interesting discoveries along the way. As discussed earlier, the integration of message passing and shared memory was one of the most important ones. Another was the discovery of the "window of vulnerability" problem. Although we had known that several livelock scenarios and some deadlock scenarios existed in the presence of multi-phase memory transactions, we had never encountered them in our initial simulations. Soon enough, however, we ran into the first of these, namely a livelock scenario that arises when both an instruction and its associated data item map to the same cache line. Our initial solution involved additional state in the cache tags to recognize and correct this problem by temporarily locking down cache lines. However, as the implementation progressed, it became clear that locking down cache lines introduced deadlocks in the presence of message-passing and LimitLESS traps. Naturally, we had to make the system work, and shortly thereafter we developed an algorithm called "associative thrashlock" and a unified hardware framework called the transaction buffer for solving these problems in general.[6]

The transaction buffer mechanism helped solve yet another problem that we encountered, or more truthfully, took upon ourselves. Late in the design phase, we chose to modify our coherence protocols to support misordering of messages in the network. Our reason for doing this was to develop a more generally applicable solution than that required for our own network (which did deliver messages in order). It turned out that minor modifications to the protocols in concert with the transaction buffer mechanism enabled us to make this significant improvement to our protocols. We also took advantage of the reordering protocols to create a software-based network overflow solution, thereby eliminating the need for multiple networks or virtual channels.

As the implementation progressed, a large body of software was written to make the machine usable. New synchronization and scheduling algorithms were among these (see Lim's and Nussbaum's PhD theses). Interestingly, some discoveries came about through limitations in the prototype. Alewife did not support virtual memory, since we believed we could answer our research questions without it. Barua and Kranz developed a software method called software address translation in which the compiler inlined customizable translations into the code. This method did not see much use when first developed, but we believe that its flexibility combined with the emergence of user-customizable operating systems like Exokernel and SPIN will make it appealing in the future as a replacement or an adjunct to hardware TLBs.

Our collaboration with LSI on the Sparcle processor worked out very well. There was a scary period, though, when Gene Hill left LSI, and the question of why Sparcle was being supported arose. With help from our technical counterparts at LSI and Sun, Godfrey D'Souza at LSI and Mike Parkin at Sun, we were able to obtain the support of Amnon Fisher at LSI, and our Sparcle efforts continued smoothly. As depicted in the timeline in the figure, working Spar-

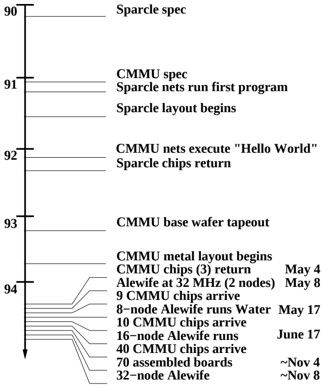
[6] Chaiken's Masters thesis discusses the thrashing problems and one of our early solutions. Kubiatowicz's PhD thesis elaborates on the window of vulnerability and the transaction buffer solution. Kubiatowicz's thesis also contains a nice discussion of the chip design, implementation and test efforts, and the futility of chip refabrication in a university.

cle chips arrived from LSI in early 1992.

We also initiated a collaboration on packaging with Bob Parker, Jeff LaCoss, and Diane Delute at the advanced packaging technology (APT) group at ISI in California. Inspired by Tom Knight's work on pressure connectors, we came up with a prototype design with Jeff LaCoss. This "cool" design had two key features: it did not involve backplane boards, and it was highly compact. Its biggest problem was that the replacement of a single board involved decompressing all the connectors, thereby violating the basic tenet of successful system building – if it ain't broke, don't touch (sic) it. Fortunately, we jettisoned this packaging technology for much less risky technology: passive backplanes, node boards, and traditional connectors.

We first designed and built a node board at MIT. This design was built with a large number of probe points and optimized for table-top debugging. Then, APT turned this prototype board design into a more compact production quality design and also designed the backplane boards. They also came up with the power distribution and cooling design. MIT added the designs for clock distribution and JTAG support.

The following figure shows the progress of our implementation effort. Perhaps the most significant feature of our schedule was our gross underestimation of the time it would take to test the CMMU chip. Fortunately, we got working parts back, and our aggressive software effort that ran concurrently with the chip design paid off handsomely. We had 2 nodes working together in a week, 8 nodes in two weeks, and 16 nodes in a month and a half.



As the machine came online, our evaluation process began. The effort got a massive infusion of enthusiasm and energy when Ricardo Bianchini came to work with us over the summer. Following Ricardo's energetic efforts, we had a large number of the Splash applications ported to Alewife within months. The results from our evaluations are reported in the Alewife paper included in this issue. As we conclude in the Alewife paper, the basic shared memory programming abstraction augmented with mechanisms such as explicit messaging, fine-grain synchronization and context switching, provided both use-performance (ease of use and reasonably good performance) and means for further tuning.

**Looking Back** Although it is a little early to look back and assess a project that came to fruition four years ago, it is instructive to do so nonetheless. In particular, it is useful to discuss features that did not deliver on their promise. As might be expected, we grew to love all of Alewife mechanisms, despite all their warts, so it is always difficult to knock any one of them. However, the reader can take a less than enthusiastic response about a feature as a sign of a negative result. Furthermore, as discussed previously and further on in this writeup, many of the features turned out to be useful in unanticipated ways, so the negative results really are often in the context of the anticipated uses.

Perhaps, most importantly, the answers to the two key questions we set out to answer at the initiation of the project, namely, how to exploit locality in a mesh network and how to build a truly scalable directory system, turned out to be of lesser importance than some of the other contributions of the project. The software approach to directories and the integration of message passing and shared memory turned out to have the bigger impact. We further observe that although software-based directories and integrated messaging seem to have impacted other research projects, this impact has not been felt in industry at this time.

It is interesting to speculate on why the issues of scalability we thought so important ten years ago turned out not to be so significant. With the enabling technologies of shared memory programming and cost-effective mesh networks open to scalable machines, we believe their usability and bill-of-materials cost are no longer the issue. We speculate that the reason lies in the nonexistence (at least at this time) of a large class of problems that demand scalable machine performance. Consequently, only a relatively small percentage of the computing world really cares about large-scale multiprocessors. As in the past, a select cadre of users – who are willing to hand-tune their applications – expend extraordinary effort to meet their computational needs. The business case for addressing the needs of this relatively small market remains as elusive today as it was a decade ago.

There are other practical factors that relate to the scalability and applicability of LimitLESS directories in either their hardware-software form or in a purely software form. Although our results demonstrated the compelling cost-performance of two-pointer directories by balancing the hardware and software components of a multiprocessor, we doubt there will ever be commercial or research machines that combine hardware and software like LimitLESS. Since Alewife was an experimental machine, it made sense to implement a few pointers in hardware since we could explore the degradation suffered in going from several pointers to zero pointers. As our results indicated, the five pointer case was competitive with an all-hardware system, and the all-software case was between a factor of two or three worse.

In the commercial environment present at the time of this writing, it makes sense to build all-hardware systems since the hardware overhead is not significant for systems with few tens of processors, and since this approach does not involve modifying existing processor interfaces. Furthermore, multigrain systems allow modest-sized systems to be built by composing smaller machines using software page-based coherence between the components. For these relatively small systems, we will likely see a transition from the hardware approach to a software approach (probably using a separate protocol processor in the short term and a unified processor in the longer term) as the increasing latency gap between the processor clock and main memory makes software attractive.

If Moore's Law ever breaks down, however, scalability will be applicable to mainstream computing (as opposed to marketing hype), and it is very possible that the same market forces might dictate a different tradeoff between software and hardware. Assuming that a new computing paradigm does not emerge, LimitLESS-style coherence could well become appealing for its scaling properties.

One of the surprises after Alewife was built was that many applications written using shared-memory were found to be competitive with the same applications written using message passing. While this was a significant and unanticipated result in favor of shared memory, it was counter to our intuition. A sensitivity analysis revealed later that the state of technology plays a major role in determining which is better. It turns out that shared memory is competitive or better than message passing when the processors are slow compared to the interconnect, and the opposite is true when the processor clocks increase relative to network speeds. We further observed that asynchronous message notification is inherently better suited to operating system like applications such as CRL.

One of the questions we asked ourselves was whether building was necessary in the face of our sophisticated simulation technology. Not surprisingly, for the applications that ran on our simulators, our results from the real prototype were not qualitatively different from our detailed simulators. However, the availability of the real prototype allowed us to develop a large number of applications and obtain results for realistic problem sizes rapidly. Since there were no major surprises, running these applications and large problem sizes served to validate our conclusions.

And of course, as discussed earlier, many of the key Alewife mechanisms would not have been invented otherwise. Taking the example of integrated messaging, we doubt Kubiatowicz would have even contemplated the introduction of a real message-based I/O mechanism for a simulator. Finally, the simulators could not have reached their level of sophistication had we not been on an implementation path. As a case in point, NWO leveraged many of the same control state machine specifications used in the real hardware. As one of us is fond of saying, if there had been a way to hypnotize ourselves into believing that we were working on a real machine, we could have saved a year spent in design verification whose major value could be measured not in terms of the contributions to science but in value to the soul.

What of fine-grain synchronization and context switching? The insignificant value of hardware support for fine-grain synchronization was one of the salient negative results from the project. As reported by Donald Yeung in his Master's thesis, the means for expressing fine-grain parallelism in the source language is of considerable importance, while the special hardware support for full-empty bits is of marginal value.

The jury on context switching is still out. Context switching is intended to improve the performance of applications with a lot of parallelism that suffer low processor utilization due to their poor cache behavior. Context switching is most useful when the network has a large latency but can deliver high bandwidth. Context switching delivered on its promise for applications with poor memory performance such as MP3D. A dedicated context for handling asynchronous message interrupts without disrupting the computational state on the processor was also valuable. However our applications exhibited reasonable cache behavior, and therefore a reasonable processor utilization. Clearly, MP3D is an exception.[7]

The open question, then, is whether there will be sufficient applications that exhibit poor cache behavior when written in a natural manner under shared memory by average programmers. Looking back, although Bianchini and Lim have done some follow up evaluation of context switching on Alewife,[8] we have been remiss in not expending the effort to find more applications and fully evaluating context switching.

Alewife leveraged many of the advances of previous research such as wormhole-routed low-dimension interconnects, directory based coherence, and efficient message interfaces. In turn, in advancing the notions of software-based directories and integrating messaging and shared memory, we hope it contributed in modest measure to this cycle of research.

[7] One might be tempted to speculate that many of these applications were developed for DASH or Alewife, both cache-based machines, and therefore coded in a cache-friendly style.

[8] Bianchini and Lim evaluated context switching on Alewife and published their findings in the August 1996 issue of JPDC. They conclude that "prefetching is preferable over multithreading for machines with low remote access latencies and/or applications with poor locality and consequently short run-lengths. The performance of both techniques is comparable for applications with high remote access latencies and/or good locality." They also argue that context switching has added value in microkernel environments.

## References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280–289, Honolulu, HI, June 1988. IEEE.