Reducing Network Latency Using Subpages in a Global Memory Environment

Hervé A. Jamrozik, Michael J. Feeley, Geoffrey M. Voelker, James Evans II
Anna R. Karlin, Henry M. Levy, and Mary K. Vernon

Department of Computer Science and Engineering
University of Washington

First published in the "Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems", October 1996.
Also available as Technical Report UW-CSE-96-07-03.

Abstract

New high-speed networks greatly encourage the use of network memory as a cache for virtual memory and file pages, thereby reducing the need for disk access. Because pages are the fundamental transfer and access units in remote memory systems, page size is a key performance factor. Recently, page sizes of modern processors have been increasing in order to provide more TLB coverage and amortize disk access costs. Unfortunately, for high-speed networks, *small* transfers are needed to provide low latency. This trend in page size is thus at odds with the use of network memory on high-speed networks.

This paper studies the use of *subpages* as a means of reducing transfer size and latency in a remote-memory environment. Using trace-driven simulation, we show how and why subpages reduce latency and improve performance of programs using network memory. Our results show that memory-intensive applications execute up to 1.8 times faster when executing with 1K-byte subpages, when compared to the same applications using full 8K-byte pages in the global memory system. Those same applications using 1K-byte subpages execute up to 4 times faster than they would using the disk for backing store. Using a prototype implementation on the DEC Alpha and AN2 network, we demonstrate how subpages can reduce remote-memory fault time; e.g., our prototype is able to satisfy a fault on a 1K subpage stored in remote memory in 0.5 milliseconds, one third the time of a full page.

1 Introduction

New-generation networks, such as ATM, have now surpassed disks in their ability to transfer data rapidly into processor memory. As research has shown, such performance greatly encourages remote paging [5, 9, 12] or global management and use of network-wide memory [10, 6, 7]. That is, network nodes with memory-intensive applications can use the primary memory of lightly-loaded nodes as temporary backing store. In effect, the use of remote memory to reduce the latency of VM, file access, or database operations, introduces a new level of the memory hierarchy: namely, a global memory cache that lies (logically) between local memory and disk. This new level has the potential for much lower latency than disk.

Because pages are the fundamental transfer and access units in remote memory systems, page size is a key factor in remote memory performance. On current processors, page size has been driven upwards by two factors. First, magnetic disks have high access latency, therefore large pages and multi-page transfers are needed to amortize the cost. Second, the combination of relatively small TLBs and large physical memories on current machines causes performance degradation due to insufficient TLB coverage. TLB coverage is increased by large page sizes or superpage mechanisms [20, 19, 16]; e.g., the DEC Alpha supports page sizes from 8KB to 1MB, the SUN UltraSPARC supports page sizes from 8KB to 4MB, and the MIPS R10000 supports page sizes from 4KB to 16MB.

Unfortunately, the latency characteristics of high-speed networks are at odds with this trend. Recent research has succeeded in substantially minimizing the latency caused by operating system software for network transfers [23, 21], and newer controllers reduce latency even further [8]. Therefore, the total latency for remote memory transfers is dictated to an increasing extent by the *size* of the transfer, rather than by the controller and software overhead. For remote memory access, then, *smaller* transfers may ultimately be needed to reduce latency and improve program performance.

This paper examines the use of *subpages* as the transfer units in remote memory systems. Subpages are hardware- or software-supported power-of-two subunits of full pages; for example, an 8K page might be managed as 8 1K subpages, 16 512-byte subpages, etc. We present several alternative techniques for using subpages for remote memory access. Using trace-driven simulation, we show how and why subpages reduce remote memory latency and improve performance of programs executing in a global memory environment. To ensure that our simulation parameters are realistic, we implemented a prototype software-supported subpage mechanism on DEC Alpha workstations connected by a DEC AN2 ATM network. We measured remote subpage transfers using this mechanism in a globally-managed memory for the Digital Unix operating system. Our implementation is able to satisfy a page fault on a 1K subpage stored in remote memory in .52 ms, compared with 1.4 ms for a full page. (By comparison, an average local disk access takes 4 to 14 ms on the same system, depending on the nature of the access – sequential or random.) Our results demonstrate the value of using subpages to reduce transfer size and latency in a high-speed

This research was supported in part by grants from the National Science Foundation (CCR-9200832, CDA-9123308, CCR-9632769, CCR-9024144, GER-9550429, and GER-9450075) and from Digital Equipment Corporation.

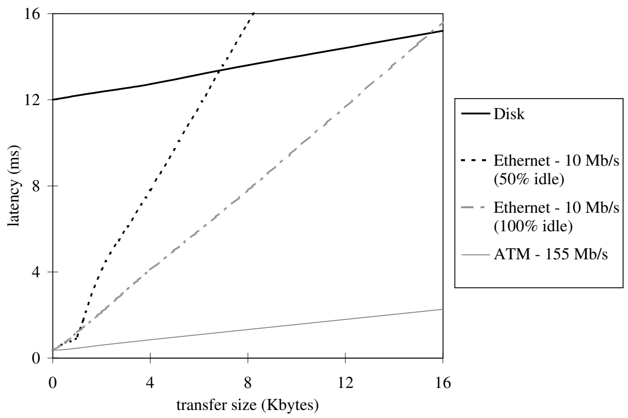

Figure 1: Latency vs. Page Size for Disks and Networks

network. For example, we show up to 1.8-fold speedup of memory-intensive applications executing with 1K subpages, when compared to the same applications using full 8K pages in the global memory system. Those same applications using 1K subpages execute up to 4 times faster than they would using the disk for backing store.

The remainder of the paper is organized as follows. Section 2 presents further motivation for the use of subpages in global memory systems and introduces several alternative techniques for subpage management. Section 3 describes our simulation methodology and our subpage prototype implementation. In Section 4 we present performance results for several subpage management schemes. We also examine application memory access behavior and show why that behavior lends itself to the subpage models defined in Section 2. Finally, Section 5 concludes and summarizes our results.

2 Motivation

As mentioned above, the characteristics of modern network performance motivate a change in the transfer size for distributed memory systems. We see this graphically in Figure 1, which plots the transfer latency of a disk subsystem, a heavily-loaded 10 Mb/s Ethernet, a lightly-loaded Ethernet, and an ATM network as a function of page size on a DEC Alpha workstation. The graph demonstrates four key points. First, the disk subsystem exhibits high latency even for a "zero-length" page, as expected. Second, the networks have much lower initial overhead; therefore the linear increase in the transfer time with page size accounts for a more substantial fraction of the total latency. Third, even for an ATM-speed network, we can reduce latency substantially with smaller packets; this is difficult to see in the figure, because the scale is distorted by the high disk access time. The final interesting point shown by the graph is that even Ethernet, while much worse than disk for transferring large pages, would still have better latency than disk for very small pages.

How important are these measurements to a system using global page caching? In previous work, we showed how network memory could be managed globally by implementing a global memory management system on the Digital Unix operating system on DEC Alphas [7]. For our system, the complete latency for faulting an 8K Alpha page into memory from a remote node, including software overhead and management messages, was about 1.6 ms. Of that time, 1.03 ms was due to network and controller time for sending the page from the source node to the destination. If the latency for a remote page fault is to be greatly reduced, then reducing this on-the-wire time is crucial. Moreover, in our experiments, the speedups using global memory management were close to the maximum achievable, given the ratio of disk access to remote memory access time. Therefore, reducing remote memory latency is important for obtaining further performance gains.

2.1 Using Subpages for Global Memory

Global memory systems rely on the idle memory of lightly-loaded network nodes to hold pages for heavily active nodes. A fault on node A may be satisfied by node B, either because B has stored A's page in its "global memory," or because A has faulted a page actively in use by B (e.g., a shared code page). In any case, pages are the fundamental units of access detection and transfer; because the virtual memory system is the only hardware mechanism available to trap access to non-resident storage, we must move data in integral page-sized units.

A *subpage* is a power-of-two subunit of a full page. Conceptually, a system supporting subpages would permit any (incomplete) subset of the subpages to be present within a full page, and would detect and trap attempted access to any non-resident subpage. In this way, when a fault occurs, the operating system could (a) transfer only the subpage containing the word accessed, and (b) restart the program immediately on receipt of that subpage. Because subpages are much smaller than full pages, the subpage transfer completes quickly and the program suffers a smaller latency. Whether this benefits the program in the long run depends on a number of issues, which we discuss shortly.

We have defined several possible techniques for handling subpage transfers:

1. *Lazy Subpage Fetch*. Transfer the faulted subpage. Bring the remaining subpages only on demand. This is equivalent in many respects to simply reducing the page size.

2. *Eager Fullpage Fetch*. Transfer the faulted subpage and restart the program. Asynchronously, transfer the remainder of the *full* page as a large transfer.

3. *Subpage Pipelining*. Transfer the faulted subpage and restart the program. Asynchronously, send the remainder of the subpages on the page. Because we are sending the remainder of the page in *subpage* units, the transfers can be *sequenced* to ship the most likely to be accessed subpages first.

Lazy subpage fetch is likely to perform poorly if the program eventually touches many subpages of the faulted page; i.e., fetching all of the subpages, one at a time, will be much worse than faulting the full page. On the other hand, if the page is read-only and the program touches only one or two subpages, the approach might be beneficial. Lazy subpage fetch is similar in many ways to a system with small pages; we compare the two approaches at the end of this section.

Eager fullpage fetch reduces the initial fault latency by transferring only a single subpage; it then *overlaps* the I/O of the remainder of the page with the execution of the program. Note that this approach could benefit execution in two ways. Assume that a program faults a subpage i on a non-resident page, P. First, if the time between that fault on i and the access to another subpage on P is greater than the transfer time for the rest of the page, we will have transferred the full page yet suffered the latency of only the single subpage, i. Second, if the program faults subpage j on *another* page, Q, before accessing another subpage on P, then the transfer of P can be overlapped with the transfer of j. In the first case computing is overlapped with I/O, while in the second two I/O transfers are overlapped.

The third scheme, subpage pipelining, is somewhat similar to eager fullpage fetch. However, in this case, the faster arrival of additional subpages has the potential to further reduce the time the program waits for the remote data, in cases where the time between the fault and the next subpage access on P is less than the full-page transfer time. As we will see, the first subpages to arrive following arrival of the faulted subpage take relatively little additional latency. This scheme is similar in some ways to the pipelining of cache data into cache lines over small buses [4].

It is interesting to compare our subpage schemes with an architecture that uses small pages (i.e., the size of one of our subpages). A major disadvantage of the small page scheme, relative to subpages, is the reduced TLB coverage and therefore higher TLB miss rate that small pages would incur. Furthermore, previous work [13] has shown that although smaller transfers offer the potential for increased locality, this advantage is outweighed by the increased overhead of the multiple requests required. We performed experiments to confirm that this is true for our environment as well. Therefore, we consider neither small pages nor lazy subpage fetch in this paper.

2.2 Discussion

From the presentation above, we saw that there are two ways to benefit from subpages: computational overlap and I/O overlap. It is not obvious *a priori* whether this overlap can, in fact, be achieved. In particular, for eager fullpage fetch, the interval between arrival of the subpage and arrival of the follow-on full page is long, for example, on the order of 72,000 memory accesses for a 1K subpage. To completely overlap computation with that interval would require that no accesses occur to the incomplete parts of the page while the I/O is in progress, which is unlikely. To overlap I/O, on the other hand, requires that the program fault another page before it accesses a non-resident subpage on the page; yet for acceptable program performance, the average time between page faults must be much larger than the time to fetch a page.

This issue of overlap causes a tension between different subpage sizes. For small subpages, the subpage fault will complete quickly, leaving more time for overlap. On the other hand, larger subpages are likely to reduce the probability of access to incomplete parts of the page. In addition, with subpage pipelining, larger subpages reduce the number of network interrupts.

The quantitative performance gain depends on two types of factors: technological factors, such as the software overhead of message transfers and the ratio between network speed and memory speed, and behavioral factors, such as the access pattern of an executing program. In the following sections we examine these issues through simulation and measurement. Our goal is to answer the following questions:

- To what extent can eager fullpage fetch improve performance, if at all? How does this vary across applications?
- What is the optimal subpage size to use? We have described tensions in both directions.
- How much of the benefit is due to overlapped I/O vs. overlapped computation? To what extent is this benefit affected by the value of the fixed overheads?
- How much additional benefit can be obtained by pipelining?

3 Experimental Methodology

To study the effect of subpage transfers on remote memory operations, we implemented both a trace-driven simulator and a software prototype. Our simulator provides a detailed analysis of the policies and mechanisms we wish to examine and allows us to experiment with a range of system parameters. For our prototype, we constructed a software implementation of subpages on the DEC Alpha 250 266-MHz workstation. We used the prototype to provide measured software management and network overheads for the simulator, and to validate simulated results where possible. The prototype and simulator are described below.

3.1 System Prototype

Our prototype is implemented as an extension to GMS, a full global memory management system described in [7]. The system is integrated with Digital Unix V3.2; nodes are connected by a DEC AN2 155 Mb/sec ATM network [1].

Subpage protection is implemented in software by modifying the PALcode [17] on the DEC Alpha 250. Our system keeps 32 subpage valid bits for each page, one bit for each 256-byte block; the valid bits indicate which subpages are currently valid (subpages are a multiple of 256 bytes). When a page fault occurs, we (1) allocate a page of physical memory, (2) transfer the first *subpage* from remote memory on another node into its proper position on that page, (3) set the corresponding subpage valid bits for that page, (4) turn *off* read/write access to the page, and (5) continue program execution. When the program attempts to access data on a page that is incomplete (i.e., not all of its subpages are loaded), the hardware traps to PALcode to handle the access protection violation. PALcode then checks the valid bits to see whether the attempted access is to a subpage that is loaded; if so, then the PALcode emulates the read or write access. This emulation slows execution, but only on incomplete pages; as soon as the remainder of the data for a page arrives, read/write access to the page is re-enabled.[1]

3.1.1 Prototype Evaluation

Table 1 shows the performance of emulated read/write operations on our prototype, as well as several other measurements for comparison. The PALcode caches the subpage valid bits for each emulated operation; a "fast" load or store occurs when an emulated operation is to the same page as the previous emulated operation. As the table shows, a fast load is 6.5 times slower than an L2 cache hit, and 1.6 times faster than an L2 miss (assuming the loaded data is in cache). Prototype measurements and simulation results indicate that emulation slowed execution by less than 1% for the workloads we examined.

For the simulation results reported in this paper, we assume TLB-based hardware support, which has no overhead associated with accessing resident subpages. This simply consists of an additional valid bit in the TLB per subpage.[2]

To gain a detailed understanding of performance, we instrumented our prototype to log crucial events. We extracted median latencies for these events from logs produced by running a memory-intensive program on our instrumented kernel configured for various subpage alternatives. These values were then used to calibrate the simulator (as described in Section 3.2) and are summarized in the rest of this section.

Table 2 shows the page-fault latencies for various subpage sizes.[3] **Subpage** latency is the time until the program resumes execution after a page fault. This latency includes two components: (1) a fixed cost of about 0.27 milliseconds for handling the fault, locating the page in the network, sending a request message to the node storing the page, processing the request message, and resuming the

[1] An alternative technique would be that used by the Wisconsin Wind Tunnel project [15, 14] to implement additional state bits for SVM on the CM5; they used ECC bits to cause faults, however, this would still require emulating writes, and the Alpha 250 has imprecise exceptions on data parity errors, making use of parity difficult or impossible. Similar techniques have been used for trace production as well [22].

[2] The IBM 801 used a similar scheme to manage transactions on units of less than a page – in their case, for each 128-byte line [3].

[3] We have optimized the performance of global memory operations along the lines described in [21], hence our latencies are slightly better than those reported in [7].

Operation	Performance	
	Cycles	Time
fast load	52	195 ns
slow load	95	361 ns
fast store	64	241 ns
slow store	102	383 ns
null PAL call	15	56 ns
L1 cache hit	3	11 ns
L2 cache hit	8	30 ns
L2 miss	84	315 ns

Table 1: Performance of PALcode Load/Store Emulation

Subpage Size (bytes)	Latency (ms)		Improvement Potential	
	Subpage	Rest of Page	Overlapped Execution	Sender Pipelining
256	0.45	1.49	50%	0%
512	0.47	1.46	47%	1%
1024	0.52	1.38	40%	7%
2048	0.66	1.25	23%	16%
4096	0.94	1.23	1%	17%
fullpage	-	1.48	-	-

Table 2: Page-fault Latencies for *Eager-Fullpage Fetch* from Remote Memory. Latencies are arrival times of subpage and rest of page. Improvement potential is given as a percentage of fullpage latency.

faulted program when the subpages arrives, and (2) a variable cost for transferring the subpage over the network and copying it into the faulting node's memory. **Rest of Page** latency is the time from the page fault until the entire page has been transferred. This latency includes the time to send the rest of the page over the network and the time to copy it into memory. The latency reduction of subpages comes at a cost of increased overhead. Subpages increase faulting-node CPU overhead by 0.08 ms to 0.48 ms and increase sending-node overhead by 0.05 ms to 0.16 ms.

Table 2 also shows that there are two ways that subpages can improve page-fault latency. The first, **Overlapped Execution**, is the amount of time between subpage and rest-of-page arrival that the faulted program can potentially run (this is less than the difference in the two latencies by the CPU overhead of receiving the rest of the page). The second, **Sender Pipelining**, is the improvement achieved by better pipelining of the rest-of-page transfer by the sending node.

Understanding the exact operation of data transfers in a complex system, particularly where pipelining and overlap may occur in both hardware and software, is often challenging. Figure 2 shows a detailed timeline containing the crucial components involved in a remote page fetch operation for full 8K pages, and 2K and 1K subpages using *eager fullpage fetch* on our hardware platform.

Each timeline contains 5 components that contribute to a remote paging operation from a requesting node to a remote node serving the faulted page:

1. *Req-CPU*: Computation on the requester (the faulting node), either to process global memory management code (the thick bar) or execute the application code (the dotted line).

2. *Req-DMA*: Time required by the requester's network controller to copy data between host memory and the controller.

3. *Wire*: Time required for transmission of data on the network interconnect.

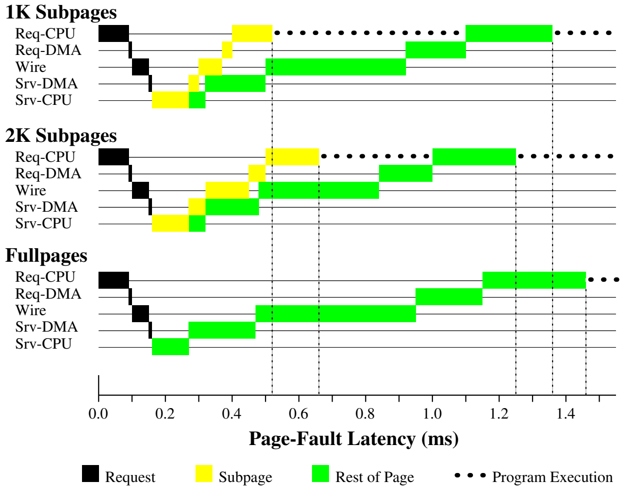

Figure 2: Remote Page Fetch Timeline for Fault on Requesting for a Page Stored on a Serving Node.

4. *Srv-DMA*: Time required on the server storing the page for the controller to copy data between host memory and the controller.

5. *Srv-CPU*: Execution on the server to process the request.

The shading on the bars indicates the work causing the particular component. For example, looking at the timeline for fullpage transfers, we see 4 black bars at the beginning of the transfer; these are for handling the fault on the requesting side and transmitting a control message to the server to request the remote page. The remaining bars are dark gray; these show the work required to transmit the page from the remote server to the requesting CPU, including the two DMAs and the wire time for the 8K page. In this case, the DMA on the requester completes at about 1.15 ms after the fault, and the application continues execution at 1.48 ms after the fault, as indicated by the dotted line.

For comparison, let's examine the timeline for a fault with 2K subpages. Again, the first 4 black bars show the sending of the request message. The light gray bars that follow are the return of the 2K subpage. Note that the DMA times and wire time for the 2K subpage are shorter than for the fullpage, so the application continues after .66 ms. The dark gray bars on this graph show the pipelining of the remainder of the page (in *eager fullpage fetch*) following the initial subpage transfer. Notice the extent to which components of this pipelined 6K transfer overlap other components of the initial 2K transfer. The second transfer arrives, interrupting the requesting CPU at approximately 1 ms. The application sees a window of .35 ms of execution before the second segment arrives, and then continues at 1.25 ms following the interrupt. In this case, then, not only did the application restart in half the time of the fullpage case, but the arrival of the *entire* transfer completed sooner, despite the fact that the same number of total bytes were sent. This occurs because of the overlap of parts of the DMA and wire transmission of the two transfers.

Finally, the top timeline shows similar detail for 1K subpages. It is surprising that the 1K case is slightly worse than the 2K case for completion of the total paging operation, finishing at close to 1.4 ms. This occurs because the DMA and wire transmission of the first 1K subpage is "too small" for optimal overlap; i.e., it completes too soon, leaving less chance for the server DMA of the remainder of the page to be fully overlapped. This leaves a "space" on the wire between the first subpage transfer and the remaining transfer that is larger than for the 2K case.

3.2 Trace-Driven Simulator

Our simulator models the memory behavior of applications executing in a global memory environment. It takes as input memory reference traces generated from applications instrumented by Atom [18]. It then simulates these memory references and models the effect of paging to both local disk and to remote memory. Paging policy is determined by a configurable memory management module; an LRU policy is used by default. Once all references have been consumed, the simulator produces as output a complete description of the paging behavior of the applications that generated the traces. This description details, among other things: the number and type of page faults, the number of faults overlapped with others, the contribution of the application's CPU time to total execution time, the time spent waiting for subpages, and the time for arrival of remaining page fragments on page faults.

The simulator models page faults to remote memory using a simplified model of the timelines shown in Figure 2, containing three components of each transfer: request time, on-the-wire time, and receive time. The request and receive times include software and DMA overheads on both nodes. These page fault components were chosen to permit modelling of the overlap of various operations in the page fault transfers. On the remote node, the request time for the remaining page fragment is overlapped with the wire time for the subpage. On the faulting node, the receive time for the subpage is overlapped with the wire time of the remaining page fragment. In addition to overlap, the simulator models congestion delays in the network.

The simulator estimates time using memory accesses as its clock events. In order to translate latencies (e.g., for subpage transfers) into events, we measured program execution time on a DEC Alpha for several applications; we then traced those applications and ran the traces through a cache simulator to model memory accesses. Using the results of the cache simulator, we then calculated the average time per trace event (i.e., per memory access) for these programs. For our applications, we calculated an average time per simulation event to be about 12 nanoseconds, i.e., 83,000 events correspond to one millisecond of execution time.

We validated the simulator by comparing its performance improvement estimates against those measured using the prototype. Both quantitative improvement for eager fullpage fetch and the trend with subpage size agreed with the prototype measures, i.e, both found the same optimial subpage size. We could not validate subpage pipelining, for reasons discussed in Section 4.3.

4 Performance Results

This section presents measurements of subpage performance and behavior from our trace-driven simulation. The purpose of these experiments is to determine how much of the potential for improvement (shown in Table 2 for eager fullpage fetch) is actually achieved in practice. We also examine the interplay between program behavior and achieved performance. As noted above, we parameterized the simulator using measurements taken from our prototype implementation. The simulator has the advantage of being easier to instrument and thus it gives us more insight into the behavior of the applications.

We traced and analyzed several applications executing in the simulated subpage environment. To see the effect of heavy paging activity, we ran the applications in different memory configurations. The applications we used were the following:

Modula-3 is the DEC SRC compiler for the Modula-3 [11] programming language. We traced a Modula-3 compilation of smalldb, a library that transparently maintains a copy of in-memory data structures on secondary storage. The trace includes 87 million memory references, and from 773 to 5655 faults, depending on the memory configuration.

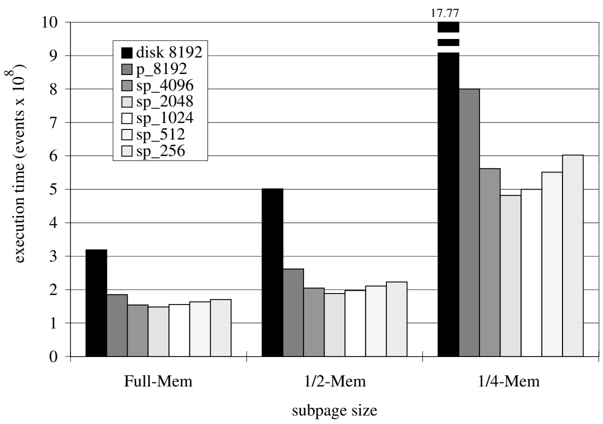

Figure 3: Subpage Performance for 3 Memory Sizes for Modula-3

ld is the unix object file linker. We traced a link of Digital Unix V3.2 to generate our ld data. The trace includes 102 million memory references, and from 6807 to 10629 page faults.

Atom [18] is the instrumentation software we used to generate the traces. To obtain this trace, we instrumented Atom itself (using Atom), then traced it while processing the gzip binary. The trace includes 73 million memory accesses and from 1175 to 5275 page faults.

Render [2] is a graphics rendering program that displays a computer-generated scene from a large (over 100MB) precomputed database. The trace includes 245 million memory references and from 1433 to 6145 page faults.

gdb is the GNU debugger. We traced the initialization phase of the debugger, run without loading a program. The trace includes .5 million references and from 138 to 882 page faults.

In the results below, we use the Modula-3 trace to evaluate the subpage mechanisms in detail. Modula-3 was average among the applications we studied with respect to performance improvement due to subpages. In general, the traces from all of the applications have the same basic behavior with respect to the various subpage options examined, and thus lead to the same conclusions. We present a summary of the performance gains for the subpage mechanisms for all of the applications at the end of this section.

4.1 Performance of Eager Fullpage Fetch

Figure 3 shows the performance of the Modula-3 trace for a number of subpage sizes and memory configurations. This graph represents a warm (global) cache situation, that is, all pages are assumed to initially reside in remote memory. For these measurements, we used eager fullpage fetch, i.e., the first subpage is transferred with the remainder of the page following immediately afterwards. Three memory configurations are shown: *full-mem*, in which the program is given as much local memory as it needs (in this case, the faults are all initial page faults), *1/2-mem*, in which the program executes in half its maximum memory, and *1/4-mem*, in which the program executes in one quarter of its maximum memory.

For each memory configuration, the left-most bar, marked `disk_8192`, shows the performance for this trace when *all* page faults are serviced from disk, i.e., there is no global memory. The

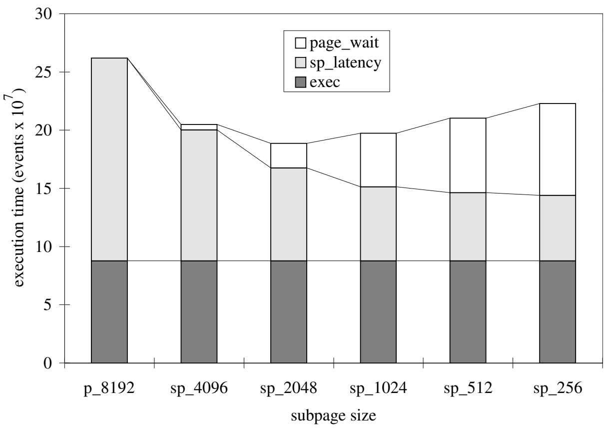

Figure 4: Subpage Performance for Modula-3 at 1/2 Memory

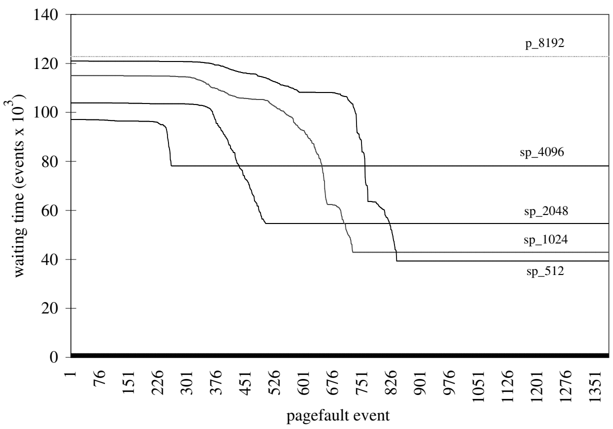

Figure 5: Sorted Per-Fault Waiting Times for Different Subpage Sizes. Each curve consists of three sections: (1) a horizontal segment on the right, representing the best case, where faults wait only the subpage latency, (2) a horizontal segment on the left, representing the worst case, where faults stall until the full page arrives, and (3) a middle region, where there is some overlap.

next bar, marked `p_8192`, shows the performance of global memory management using full 8K pages. As previous studies have shown, the performance improvement of global memory relative to disk is significant — in this case, the speedups range from 1.7 to 2.2, depending on memory availability [7]. The remaining bars show the performance for subpage sizes ranging from 4096 to 256 bytes.

Subpages offer improvement over full pages ranging from about 8% (for 256-byte subpages in full-mem) to 40% (for 2K subpages in 1/4-mem). The performance improvement of subpages increases as the program's memory demands are stressed, e.g., 1K subpages show a 16% improvement for full-mem, a 25% improvement at 1/2-mem, and a 38% improvement for the 1/4-mem configuration. Subpage sizes of 2K are the best choice for Modula-3, although all subpage sizes show improved performance relative to full pages. Over all the applications, subpage sizes of 1K or 2K were best, with only slight differences between them.

Figure 4 shows the results for 1/2-mem in more detail. Three components of the trace runtime are shown: the execution time of the program (exec), waiting time due to transfer of the first subpage on each faulted page (sp_latency), and time stalled waiting for the arrival of the remainder of the page (page_wait). The sp_latency bars demonstrate the fault latency improvement with smaller subpages; sp_latency decreases from 55% of total runtime at sp_4096 to 25% at sp_256. With decreasing latency, however, comes an increase in the page_wait component, from 2% at 4K to 35% at 256 bytes. This occurs for two reasons: first, smaller subpages increase the probability of accessing another subpage before either the page is complete or a miss on a different page occurs; second, because the subpage fault completes more quickly, there is increased *opportunity* (in time) to access a non-resident subpage before the full page arrives. Thus, there are both spatial and temporal reasons for the impact of smaller subpages.

In order for very small subpages to outperform larger ones, we will have to reduce this page_wait component. We examine one method of doing this, subpage pipelining, in Section 4.3.

4.2 Analysis of Subpage Performance

One might expect the benefit of eager fullpage fetch to increase as subpage size decreases, because the subpage fault completes more quickly, leaving more time to overlap. As we have seen, however, smaller subpages do not necessarily perform better. This is partially explained by the reduced opportunity for sender pipelining, as shown in Table 2. Additional insight is given in Figure 5, which shows the interaction of subpage size, latency, and waiting time. For each page fault, j, listed on the X axis, the graph plots the total waiting time that occurred for data on page j; waiting time includes the initial subpage latency plus any waiting time afterwards for arrival of the remainder of the page. The faults are sorted by waiting time, with the highest waiting times on the left, and the lowest waiting time on the right.

All of the subpage curves in Figure 5 are composed of three sections:

1. A horizontal segment in the lower right. The right intercept of this segment, for each curve, is the time to transfer the subpage of the specified size. These faults saw the best case: they waited only for the subpage transfer to complete, and then overlapped computation and I/O with the rest of the full-page transfer.

2. A horizontal segment in the upper left. The left intercept of this segment, for all curves, is the time to transfer the full page. These faults saw the worst case: they were not able to overlap the full-page I/O, and quickly blocked waiting for the remainder of the full page to arrive.

3. A small region in the middle, where some overlap occurred.

In this figure, we see that all subpage sizes benefit, by different amounts, relative to the normal 8K full-page global memory system. As well, we see that program behavior changes with subpage size in two distinct ways. As subpage size decreases, faults that are fully overlapped have higher benefit, as shown by the decreasing Y intercepts on the right. On the other hand, the marginal benefit decreases with decreased page size. Moreover, there are fewer faults that achieve best-case overlap, as shown by the decreasing *length* of the "best-case" (right-hand) segment. This decrease occurs because smaller subpages increase the probability of accessing data on another (non-resident) subpage. For Modula-3, 2K subpages offer the best trade-off between these alternatives.

As discussed in Section 2.2, full overlap should be difficult to achieve. It is therefore surprising that for all subpage sizes, a large fraction of the page faults achieve best-case overlap. If computational overlap accounted for a significant fraction of the total overlap, we might expect more of the page fault events to fall in the middle region of the curve. In fact, our measurements confirm that

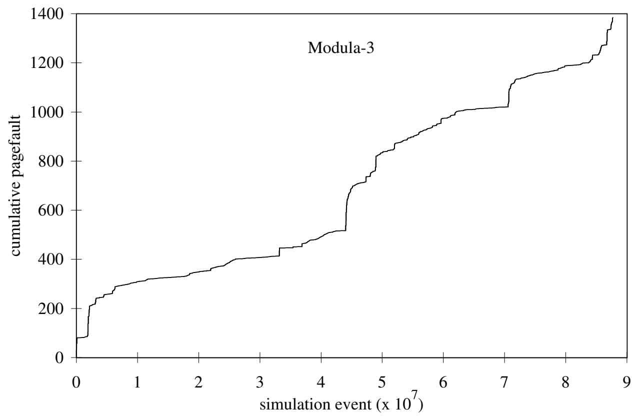

Figure 6: Temporal Clustering of Page Faults for Modula-3

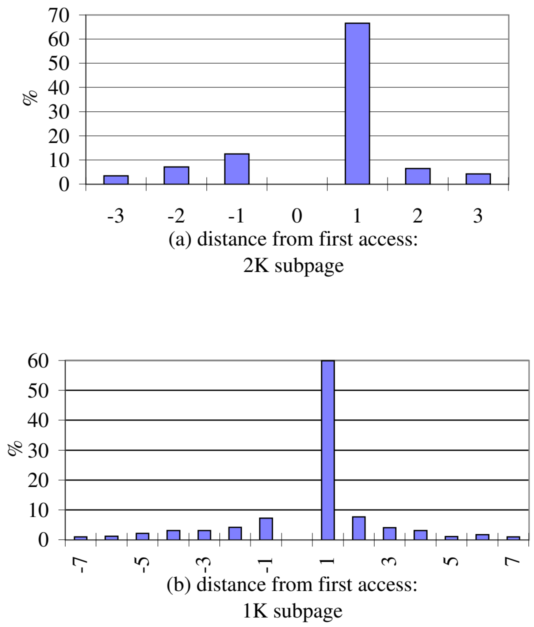

Figure 7: Distribution of Distances to Next Accessed Subpage on the Same Page

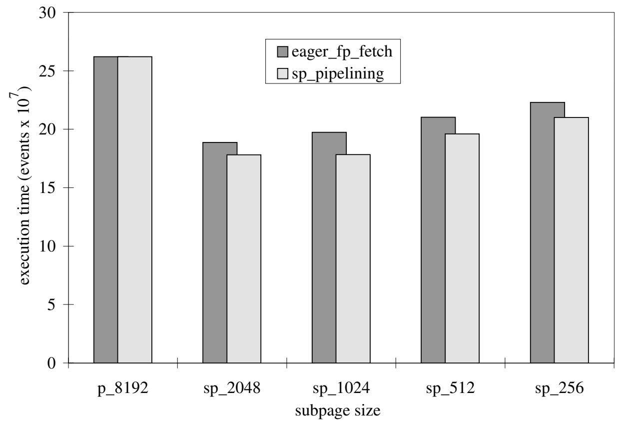

Figure 8: Comparison of Eager Fullpage Fetch and Subpage Pipelining (Modula-3, 1/2 mem)

most of the speedup can be attributed to parallel I/O, rather than computational overlap.

It is surprising that so much I/O overlap is possible, given that the average time between faults is much larger than the time to fetch the rest of the page. The reason this is possible is that many programs with low fault rates undergo periods of high faulting, e.g., during a phase change. Figure 6 plots the temporal clustering of page faults for Modula-3. For each simulation event, the graph shows the number of page faults that have occurred up to that point in time. Horizontal segments of the curve represent periods of high computation and little fault activity; vertical segments of the curve are periods of high fault rate. I/O overlap occurs mostly during the high-fault intervals; the larger the fraction of faults that occur during these periods of high faulting the greater the expected increase in performance from eager fullpage fetch.

4.3 Pipelining Subpage Access

From Figure 5, we see that small subpages reduce the initial fault latency, but they also increase the probability of blocking before the full page arrives. Recall that in all measurements shown above, the remainder of the page is transferred in a large message following the initial subpage transfer. An alternative scheme is to *pipeline* that remaining data in multiple smaller messages. Pipelining has the potential to further improve performance for several reasons. First, the initial transfer of the faulted subpage includes the additional software latency of finding the node storing the page, sending it a message, and so on. Once the source node has been contacted, it can send the pipelined sections immediately after the initial subpage; these transmissions will have no additional latency beyond the minimum due to the time on the wire, the controller, and the receiver-side interrupt. Therefore, small pipelined follow-on transfers will complete quickly, one at a time, after receipt of the initial subpage; if the receiver-side interrupt overhead is small, the faulting program will be able to continue to execute while the pipelined subpages arrive. Second, with pipelining we have the potential to sequence the pipelined segments if we have information about the likelihood of accessing particular subpages relative to the first fault. The goal is to have the pipelined subpages arrive in the order in which they are most likely to be accessed, so as to reduce the program's stall time.

The crucial issue, then, is to determine the most likely order of access. To explore this issue, we modified our simulator to record the subpage number of the first *different* subpage to be touched after the initial subpage fault. Figure 7 shows this pictorially. The X axis shows the distance (in subpages) from the initial faulted subpage to the subpage next accessed. The Y axis shows the percentage of next accesses for each subpage distance. Figure 7a shows results for 2K subpages, while Figure 7b shows results for 1K subpages. In both cases, we see that these programs exhibit significant spatial locality: there is a high likelihood that the next subpage faulted on the same page will be the next consecutive subpage (distance +1).

Below we report simulation results examining pipelining and its potential benefits. In the simulation results shown we assume zero CPU overhead on the receiving node for the follow-on pipelined subpages. In our current prototype using the AN2 controller, however, each pipelined subpage causes an interrupt whose handling cost exceeds the wire time for the subpage (e.g., the overhead is 68 μs for a 256-byte subpage and 91 μs for a 1K subpage), making computational overlap impossible. This overhead could be reduced by eliminating the need for copying. Eliminating the interrupt entirely is more difficult with the AN2, and would require a network controller that is able to update subpage valid bits directly. Therefore, on our current prototype, software pipelining does not

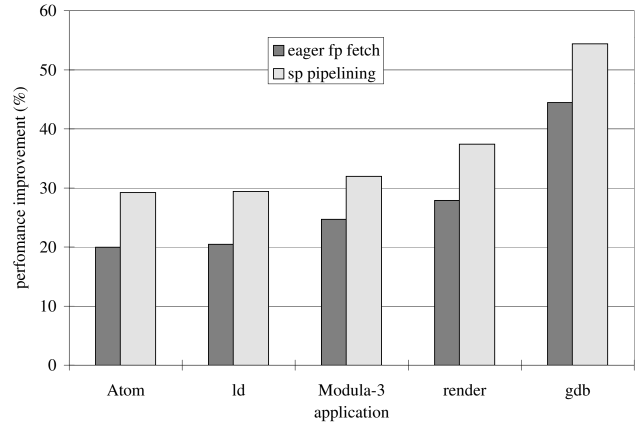

Figure 9: Reduction in Execution Time for Eager Fullpage Fetch and Subpage Pipelining (1/2-mem, 1K subpages).

outperform eager fullpage fetch. Our simulation results show the performance improvement subpage pipelining could yield if these overheads could be reduced or eliminated with a controller better suited to the task.

Based on the measurements from Figure 7, we simulated subpage pipelining where we transfer the faulted subpage and pipeline behind it the following and preceding subpages (the +1 and -1 subpages) on the page. The remainder of the page then follows in one message. Figure 8 shows the additional benefit that can be achieved through subpage pipelining on the Modula-3 trace with the 1/2 memory configuration, relative to subpages without pipelining. Note that pipelining will only reduce waiting that occurs after the first subpage arrives (shown as page_wait in Figure 4). At subpage size of 1K, pipelining reduces the page_wait component by 42%; however, relative to the entire execution at 1K, the reduction is only 10%. The improvement is larger for smaller memory configurations.

We can understand the effect of pipelining by looking back at the waiting time curves in Figure 5. Pipelining will not affect the length of the "best-case" (lower-right) part of the curves because it does not affect the subpage latency. It can, however, significantly reduce the length of the "worst-case" (upper-left) segments of the curves; with pipelining, an access that would have blocked waiting for the fullpage transfer time can continue much sooner, because the subpage it referenced has been pipelined and arrives more quickly.

In our studies, we simulated several different pipelining schemes. For example, in one experiment, we doubled the size of the pipeline transfers. That is, for 512-byte subpages, we sent the faulted 512-byte subpage followed by a pipelined transfer of the next 1K bytes. The motivation for choosing this strategy, particularly for small pages, is that there is little additional latency for doubling the length of the follow-on transfer. As another example, we transferred twice the subpage size for the initial fault, choosing to send either the preceding or following page along for the ride, depending on *where* in the subpage the faulted word was located. In general, we found that all of the schemes showed various amounts of improvement over eager fullpage fetch, but none made a substantial improvement relative to the basic scheme shown in Figure 8.

4.4 Summary

Overall, the simulation results for the Modula-3 compilation show that subpages have the potential to reduce latency for global memory access. Figure 9 summarizes the performance gains for eager fullpage fetch and subpage pipelining for all five of our applications, when executing in 1/2-mem configuration with 1K subpages.

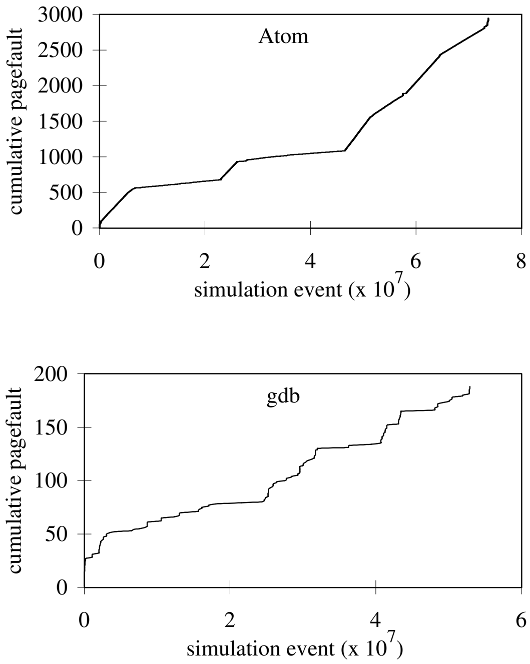

Figure 10: Temporal Clustering of Page Faults for gdb and Atom

All applications show measurable benefit due to subpages; in two cases the benefit is greater than for the Modula-3 compilation. This further indicates that subpage mechanisms may be worthwhile. The performance increase due to subpages ranges from 20% to 44% for eager fullpage fetch and from 30% to 54% for subpage pipelining.

For current high-speed networks and memory technology, most of the benefit we see for eager fullpage fetch derives from overlapped I/O. Indeed, we measured for each application in Figure 9 the percentage of speedup improvement due to overlapped I/O, which varied from 53% (for Atom) to 83% (for gdb).

Figure 10 further supports the observation that the applications that obtain the greatest speedup are those with the largest fraction of faults occurring during periods of high fault rate. Here we see that for gdb, periods of high fault rate account for the majority of the page faults, as indicated by the steep vertical jumps in the curve. In contrast, for Atom all regions have relatively low fault rate, as indicated by the relatively smooth increase in the curve. As we would expect, gdb benefits significantly more from subpages than Atom.

Applications running on any given network memory system will have different degrees of nonuniformity in their fault rates. Often applications will have significant periods of high fault rate due to phase changes. Thus, intuitively, the benefit due to overlapped I/O estimated in our simulation experiments is likely to be observed in real workloads.

We have found that subpage pipelining improves performance for all the applications we examined, assuming an intelligent controller that eliminates the CPU overhead for pipelined subpages. This benefit derives from a combination of increased computational overlap and the new opportunities for I/O overlap. Note that the relative improvement of subpage pipelining is larger for the applications that derive smaller benefit from eager fullpage fetch.

Finally, while for current technological parameters our simulations indicate that the optimal subpage size is about 2K, we might expect that size to decrease in the future, particularly for subpage pipelining, as the ratio of network speed to memory speed increases.

5 Conclusions

This paper examined the use of subpages to reduce the latency of network access in a network-wide memory system. Subpages are motivated by high-speed networks, whose transfer time is to a large extent proportional to message size.

We presented two techniques for using subpages. Eager fullpage fetch brings a faulted subpage and sends the remainder of the page as a large follow-on transfer. Subpage pipelining sends the page in smaller units following the faulted subpage, with the hope that those follow-on subpages will arrive (just) before they are needed.

Our prototype demonstrates that on ATM-speed networks, subpage transfers can substantially reduce fault latency relative to full pages. For example, on our Alpha/AN2 prototype on Digital Unix, a remote memory fault, even for 1K subpages, completes in about one third the time of an 8K full-page fault from remote memory. This is between 7 and 28 times faster than a fault serviced from disk by the NFS file system, depending on the nature of the file (e.g., sequential vs. random access). The need to fetch less than a page on a fault will increase with increasing page sizes, which are required by huge primary memories and small on-chip TLBs. Subpages would be best supported in hardware using extra per-page valid bits, but they can be supported in software as well, using several different techniques. Our prototype uses PALcode on the Alpha to trap access to incomplete pages, and emulates reads and writes directed to valid subpages. Despite the emulation, our prototype achieves speedup, e.g., 24% performance improvement over fullpages for eager fullpage fetch with 2K subpages on the Render application.

Our detailed simulations, validated by prototype measurements, show the performance benefits of using subpages. Our programs saw up to a 44% improvement when using subpages compared to full pages, and more with pipelining. Our "worst" application was able to decrease execution time by 20% with 1K subpages relative to full 8K pages, and by 29% when subpage pipelining was used. A detailed examination of the behavior of our applications shows that most of the benefit comes from I/O overlap.

Our analysis shows that spatial locality for these programs requires that the full page be transferred eventually. That is, simply reducing the page size to support smaller pages would actually degrade performance. However, by fetching most of the page asynchronously, as with eager fullpage fetch, we obtain significant benefit and improve TLB coverage relative to smaller pages as well.

We believe that in the future, as networks continue to improve, network-wide memories and subpage transfer units will be one key approach to achieving high performance.

Acknowledgements

Comments from Dylan McNamee, Jeff Chase, Ashutosh Tiwary, Vivek Narasayya, and the anonymous referees helped improve the quality of the paper. The authors would like to thank David Mazieres for his Alpha-PALcode editing tools and Ted Romer for help using them, Chandu Thekkath and Hal Murray for help in understanding AN2 network performance, David Conroy for explaining Alpha ECC handling, and Bradford Chamberlain and Dylan McNamee for the Render application. We would also like to thank the Digital Equipment Corporation Systems Research Center for providing us with the DEC AN2 network used in our prototype.

References

[1] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems*, 11(4):319–352, November 1993.

[2] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast rendering of complex environments using a spatial hierarchy. In *Proc. of Graphics Interface '96*, May 1996.

[3] Albert Chang and Mark F. Merge. 801 storage: Architecture and programming. *ACM Trans. on Computer Systems*, 6(1), February 1988.

[4] Douglas W. Clark, Butler W. Lampson, and Kenneth A. Pier. The memory system of a high-performance personal computer. *IEEE Trans. on Computers*, C-30(10), October 1981.

[5] Douglas Comer and James Griffioen. A new design for distributed systems: The remote memory model. In *Proceedings of the USENIX Summer Conference*, June 1990.

[6] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, November 1994.

[7] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[8] Edward W. Felten, Richard D. Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnick, Liviu Iftode, and Kai Li. Early experience with message-passing on the Shrimp multicomputer. In *Proc. of the 23rd International Symposium of Computer Architecture*, May 1996.

[9] Edward W. Felten and John Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, Department of Computer Science and Engineering, University of Washington, March 1991.

[10] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global memory management in client-server DBMS architectures. In *Proceedings of the 18th VLDB Conference*, August 1992.

[11] Samuel P. Harbison. *Modula-3*. Prentice Hall, Englewood Cliffs, NY, 1992.

[12] Liviu Iftode, Karin Petersen, and Kai Li. Memory servers for multicomputers. In *Proceedings of the IEEE Spring COMPCON '93*, pages 538–547, February 1993.

[13] Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenepoel. File access performance of diskless workstations. *ACM Trans. on Computer Systems*, 4(3), August 1986.

[14] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D.Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proc. of the ACM Sigmetrics Conf. on Measurement and Modelling of Computer Systems*, May 1993.

[15] Steve K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel support for the Wisconsin Wind Tunnel. In *Proc. of the 2nd USENIX Symp. on Micorkernels and Other Kernel Architectures*, September 1993.

[16] Ted Romer, Wayne Ohlrich, Anna Karlin, and Brian Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proc. of the 22nd Annual Int. Symp. on Computer Architecture*, June 1995.

[17] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, One Burlington Woods Drive, Burlington, MA 01803, 1992.

[18] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. Technical Report 94/2, DEC Western Research Lab, March 1994.

[19] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proc. of the 6th Int. Conf. on Arch. Support for Programming Languages and Operating Systems*, October 1994.

[20] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proc. of the 19th Annual Int. Symp. on Computer Architecture*, May 1992.

[21] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating data and control transfer in distributed systems. In *Proc. of the 6th Int. Conf. on Arch. Support for Prog. Languages and Operating Systems*, October 1994.

[22] Richard Uhlig, David Nagle, Trevor Mudge, and Stuart Sechrest. Trap-driven simulation with Tapeworm II. In *Proc. of the 6th Int. Conf. on Arch. Support for Prog. Languages and Operating Systems*, October 1994.

[23] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.