Performance and Optimization of Data Prefetching Strategies in Scalable Multiprocessors

RAFAEL H. SAAVEDRA, WEIHUA MAO, AND KAI HWANG

Dept. of Computer Science and Dept. of Electrical Engineering, University of Southern California, Los Angeles, California 90089-0781

Prefetching is one of several techniques for hiding and tolerating the large memory latencies of scalable multiprocessors. In this paper, we present a performance model for analyzing the limits and effectiveness of data prefetching. The model incorporates the effects of program behavior, network characteristics, cache coherency protocols, and memory consistency model. Our results indicate that, as long as there is enough extra network bandwidth, prefetching is very effective in hiding large latencies. In machines with sufficiently large caches to hold the program working set, the intra- and internode cache interference is marginally low enough to have any significant impact on prefetching performance. Furthermore, we reveal the fact that the effective prefetch distance plays a vital role and adapts extremely well to changes in cache miss rates and remote latencies, thus allowing a prefetch to be more effective in hiding latency. An adaptive algorithm is provided to optimize the prefetch distance. This is based on the dynamic behavior of the application, interconnection network, and distributed caches and memories. This optimization of the prefetch distance constitutes a significant advantage of prefetching over other latency tolerating techniques, such as multithreading. We show that the prefetch distance can be chosen constant, program-dependent, or decided by performance information. The optimal distance could be adaptively determined using both compile-time and runtime conditions. Our results are therefore useful not only to compiler writers, but also for the development of runtime support systems in multiprocessors. In large-scale systems, in which network traffic control predominates the performance, the ultimate goal is to match program behavior with machine behavior. © 1994 Academic Press, Inc.

1 This material presented here is based on research supported in part the National Science Foundation under Grants CCR-9308981 and CDA-9216321, and by the Advanced Research Projects Agency under Rome Laboratories Contract F30602-91-C-0146. This research was also partially supported by NSF Grant MIP-890417 with matching funds from USC.

1. INTRODUCTION

Reducing the long memory latencies in parallel machines is one of the fundamental problems facing computer architects today. When a processor stalls for the whole duration of a remote memory request, the processor utilization drops dramatically. To cope with this problem, the most promising hardware/software mechanisms are: (1) coherent caches, (2) relaxed memory consistency models, (3) data prefetching, and (4) processor multithreading. Coherent caches try to avoid sending remote requests by caching data close to the processors [Tang76, Cens78, YenW85, Agar90, Leno90]. Relaxed consistency memory models (event ordering models) reduce the latency of memory requests by pipelining them, and hide the latency of writes by buffering them and allowing reads to bypass writes [Dubu88, Sche88, Adve90, Ghar90].

Prefetching attempts to hide the latency of reads by issuing them ahead of time with the expectation that, by the time the processor needs it, the data will arrive at the node or at least will be moved closer to it [LeeR87, Port89]. Multithreading also attempts to hide the latency of reads, but, in this case, by context switching between active threads [Saa90, Agar90]. Although each thread still blocks for the duration of the remote transaction, the processor utilization increases by doing other work [Smit76, Hals88, Agar90]. If a significant fraction of the threads belong to the same program, then its execution time improves as well. The improvements from prefetching and multithreading complement those produced by coherent caches and relaxed memory models.

With respect to prefetching, several studies have been made using simulation techniques to evaluate the performance benefits of different flavors of prefetching. Some of the relevant aspects of prefetching considered in those studies are the following. First, there is the decision of what data to prefetch and when to prefetch it. Second, prefetching can either be the responsibility of the hardware [Baer91, Chen92] or the software (programmer or compiler) [Port89, Mowr92]. Software-controlled prefetching has been shown to be more effective than prefetching under hardware control, as the former offers more flexibility in terms of identifying the important data to prefetch and deciding when to issue prefetches [Mowr92]. Third, prefetching can be binding or non-binding.

In binding prefetching, data is prefetched directly into registers, eliminating the need to explicitly read it again later, but at the cost of making it non-accessible to the cache coherency protocol [LeeR87]. This severely limits how far prefetches can be moved away from their actual use. In nonbinding prefetching, though, the data only is prefetched into the cache (not necessarily the first-level
cache), and requires an explicit load from the processor [Mowr91, Mowr92]. An example of this is DASH, where prefetches are loaded into each cluster’s Remote Access Cache which effectively acts as a third-level cache for all processors in the same cluster. The overall consensus arising from all these studies is that nonbinding, software-controlled prefetching offers the most potential in mitigating memory latency.

Based on the performance effects of prefetching, we use a large set of machine and program parameters to determine the optimal prefetch distance towards program optimization. This may result in tangible performance benefits in future machines having possible thousands of processors and in which latencies can be significantly larger than those found in today’s machines. This optimization process attempts to answer the following questions: (1) How much latency can prefetching hide? (2) What is the performance impact of the cache interference induced by prefetching? (3) How does prefetching scale with machine size, cache line size, coverage factor, and run lengths? and (4) What are the effects that cache coherence protocols and relaxed memory models have on prefetching?

As long as the interconnection can provide the required bandwidth, prefetching provides a significant improvement in performance and can hide very large latencies even in machines with thousands of processors. Furthermore, we prove that the effective prefetch distance adapts very well to dynamic changes in the latency. This elasticity of the effective prefetch distance with respect to increases in the latency and higher cache miss rates may offer an advantage to prefetching, when compared with other memory tolerating techniques, such as multithreading, which do not offer the same degree of adaptation. The optimum prefetch distance, however, depends on dynamic program behavior, like cache miss rates, amount and effectiveness of prefetching, and the remote latency.

This paper is organized as follows: We introduce the basic architectural and program execution models in Sections 2 and 3. Our network and performance models for prefetching are formulated in Section 4. Remote memory access latency is expressed as a function of the network characteristics, coherence protocol and consistency memory model used. In Section 4 we validate the remote latency model against experimental measurements reported on the Stanford DASH prototype. We discuss limits and effectiveness of prefetching in Section 6. Then we present approximations to find the optimal prefetch distance in Section 7 and discuss how adaptive optimization techniques could be used to obtain better performance. Finally, we summarize research results obtained and comment on future work needed.

2. PROGRAM AND ARCHITECTURAL ASSUMPTIONS

In Fig. 1, we show the architectural model of a distributed shared-memory multiprocessor consisting of \( N \) processing nodes. An arbitrary interconnection network topology which appeals to scalability is assumed. However, we concentrate on a \( k \)-ary \( n \)-cube network, which has been adopted in many massively parallel processors [Dal90, Cry93]. Processing nodes are interconnected by channels having a width of \( W \) bytes. Each node consists of a processor, a cache, local memory, a directory, and a network interface linked by a memory bus.

The processor is connected directly to the cache. The physical memory is partitioned among the nodes. Coherency between the caches is maintained using a distributed, directory-based, and write-validation protocol similar to the one used in DASH [Len90]. The unit of coherence and transfer is the cache block (same size as a cache line) having a length of \( b_c \). Each node’s directory is responsible of keeping information about which remote nodes have copies of certain local memory blocks.

We assume the hardware enforces a relaxed consistency model in which the latency of writes is hidden from the processor by buffering them. The depth of the buffer should be large enough to contain all outstanding writes. We also assume that there is a machine instruction for prefetching and that normal loads stall the processor, so the only way to have several outstanding reads requests is by explicitly issuing prefetches. Because we are interested in evaluating the trade-offs present in prefetching, we do not set limits to the compiler in its ability to insert prefetches or to increase the prefetch distance. Therefore, we ignore synchronization operations which with respect to prefetching, tend to limit the maximum magnitude of the prefetch distance.

We model the effectiveness of the compiler in generating prefetches by using three parameters: (1) the coverage factor \( f \), which measures the fraction of the “original” cache misses (defined with respect to a non-prefetching execution) that are covered with prefetches; (2) the redundancy factor \( g \), which represents unnecessary prefetches of data that are already present in the cache; and (3) the static prefetch distance \( D_p^s \), which corresponds to the perfect case of full coverage (\( f = 1 \)) and complete hiding of latency.

Note that a miss covered by a prefetch not necessarily becomes a hit. Prefetches can be canceled by other accesses (hits and misses), prefetches, or invalidations, be-
TABLE I
Machine and Program Parameters

<table>
<thead>
<tr>
<th>Base machine parameter*</th>
<th>Value</th>
<th>Other relevant parameters</th>
</tr>
</thead>
</table>
| Run length: $R$ (cycles) | 100   | Number of execution intervals: $M$
| Prefetch overhead: $V$ (cycles) | 2     | Remote memory latency: $L_R$
| Number of processors: $N$ | 64    | Static prefetch distance: $D_S$
| Network dimension: $n$ | 3     | Effective prefetch distance: $D_e$
| Network radix: $k$ | 4     | Idle time per execution interval: $I(f)$
| Channel width: $W$ (bytes) | 2     | Time per execution interval (no prefetching): $T_s$
| Wire and switch delay: $T^* + T^*$ (cycles) | 2    | Time per execution interval (prefetching): $T_p$
| Local memory latency: $L_L$ (cycles) | 20   | Cache interference stall time: $T_{env}$
| Static remote memory latency: $L_R$ (cycles) | 123  | Prefetch stall time of local requests: $T_{cl}$
| Req/inv/ack message size: $S_B$ (bytes) | 6    | Prefetch stall time of remote requests: $T_{cr}$
| Cache line size: $h_c$ (bytes) | 16   | Prob. of extra miss (intranode): $P_{max}$
| Total cache size: (Kbytes) | 64   | Prob. of a cancel prefetch (intranode): $P_{c}$
| Average messages in critical set: $\alpha$ | 2.50  | Prob. of extra miss (intermode): $P_{max}'$
| Average messages in total set: $\beta$ | 3.75  | Prob. of a cancel prefetch (intermode): $P_{c}'$
| Read-exclusive probability: $P_{RX}$ | 0.3  | Coverage factor: $f$
| Prob. of local reads: $q$ | 0.5  | Channel utilization: $\rho$
| Average number of invalidations: $N_v$ | 1.2  |  
| Unnecessary prefetches per exe. interval: $r$ | 1.0  |  

*These values are used on most figures. Values of parameters differing from these are explicitly given in the corresponding figures. The static remote memory latency was computed from the parameters and represents the basic remote latency without contention ($L_R$).

before the processor has time to reference them. Other architectural and program parameters of the model are introduced in the relevant sections. Table I summarizes all important program and machine parameters that define the base configuration.

3. PROGRAM EXECUTION MODEL UNDER PREFETCHING

Let us begin by assuming that the program does not do any prefetching. Under this assumption, the execution of a program consists of a sequence of $M$ execution intervals, each delimited by two consecutive cache misses and having the following two components: (1) the processor executes instructions for $R$ cycles ($R = 1/m_{cache}$, where $m_{cache}$ is the cache miss rate), corresponding to a single run length, without experiencing a cache miss; and (2) with probability $q$ (alternatively $1 - q$) the cache miss at the end of each run length is satisfied by the local (remote) memory. A cache miss stalls the processor for $L_L$ ($L_R$) cycles. Hence, the program’s execution time without prefetching, $T_n$, is

$$T_n = M(R + I(0)) = M(R + qL_L + (1 - q)L_R).$$  \(1\)

where $M$ is the number of cache read misses (execution intervals) and $I(0)$ represents the average stall time per execution interval. Figure 2a illustrates this situation. The zero inside $I(0)$ indicates that no misses are being prefetched. In what follows, we refer to the execution of the program without prefetching as the original execution.

Under software-directed nonbinding prefetching, the compiler inserts prefetch instructions into the program with the goal of loading ahead of time those variables that, under static data dependency analysis and cache miss estimates, are expected to generate misses. However, because the compiler does not have complete information about the dynamic behavior of the program, it will not be able to successfully cover all misses. It is also important to understand that a miss covered by a prefetch may still stall the processor, if the prefetch arrives late or if it is canceled by other activity. Furthermore, the compiler may also insert unnecessary prefetches for those variables that generate hits in the original execution. Let $f(0 \leq f \leq 1)$ be the fraction of the original misses covered by prefetches (the coverage factor) and $g (g \geq 0)$ be the average number of unnecessary prefetches in a single execution interval. If a prefetch carries an overhead of $V$ cycles, then the cost of prefetching is $(f + g)V$ cycles per execution interval.

In addition to the overhead costs, prefetching also produces some destructive cache behavior. The main objective of prefetching is to load data into the cache ahead of time. Under nonbinding prefetching, these cache lines continue to be visible to the coherency protocol. Therefore, prefetched data have the potential of causing intra-(local) and internode (remote) cache interference. Internode interference manifests itself in two different ways: (1) a prefetched line can displace another cache line that under the original execution was a hit; and (2) a prefetched line can be removed from the cache by either an access or another prefetch before the processor has time to reference it. In the former case a prefetch generates a extra miss, while in the latter it cancels a prefetch. Similarly, internode interference happens when invalidations generated by prefetches occurring at other nodes
FIG. 2. The effect of prefetching in the execution time. Parts (a) and (b) show, respectively, the original execution without prefetching and the ideal execution when memory latency is zero. Part (c) represents the case where all the remote latency is covered by prefetching \(D_p \geq L_r\), while in part (d) the processor stalls for \(I(f)\) cycles because the prefetch arrived late.

Transform original local hits into misses or cancel prefetched data before they are referenced by the processor. By taking all these effects together we can compute the execution time under prefetching \(T_p\) as

\[
T_p(f) = M(R + I(f)) = M(R + (f + g)V + (1 - f)(qL_i + (1 - q)L_r) + f(T_{ext} + T_{s,1} + T_{s,2})),
\]

where \(T_{s,1}\) and \(T_{s,2}\) are the local prefetch stall time and remote prefetch stall time, and \(T_{ext}\) is the extra stall time caused by intra- and internode cache interference. Equation (2) assumes that the fraction of local to remote references does not change with prefetching. Machines with more than one processing node (processor/cache) per cluster, misses to remote locations sometimes can avoid sending messages, if these are satisfied by another cache in the same cluster. Prefetching, due to cache interference, can change some local requests into remote and vice versa. We assume this effect to be negligible. Note that in Eq. (2), \(M\) represents the number of original execution intervals (under no prefetching), and not the number of misses under prefetching. The speedup produced by prefetching is defined as

\[
\text{Speedup}(f) = T_n/T_p(f).
\]

3.1. Prefetch Distance and Stall Times

In order to compute \(T_{s,1}\) and \(T_{s,2}\), we need to introduce the concept of prefetch distance. The distance of a prefetch is the number of cycles from the moment the prefetch instruction is issued until the data is referenced by the processor. We make the distinction between static and effective prefetch distances. The static prefetch distance \(D_p^s\) is computed assuming full coverage of misses and that the latency of memory operations is completely hidden to become zero (ideal execution), while the effective prefetch distance \(D_p\) is the distance given the runtime values of the memory latencies. It is clear that the effective prefetch distance depends not only on \(D_p^s\), but also on the local and remote latencies, the amount of prefetching present in the program, as well as the effectiveness of prefetching and the amount of cache interference induced by it.

How the prefetch distance affects the amount of stall time experienced by the program is illustrated in Fig. 2. In Fig. 2a, we show the program’s original execution assuming no prefetching. An hypothetical prefetch inserted at the beginning of the second run length would have a prefetch distance of \(D_p\). This prefetch attempts to hide the latency of the miss that ends the third run length. Figure 2b illustrates the ideal case assuming that all memory latencies are reduced to zero.

Prefetching does not eliminate cache misses but only hide their latency from the processor. Because the effectiveness of prefetching results in a reduction of the original execution time, while the same (or more) misses have to be satisfied, a side effect of prefetching is to increase the network traffic. This increase in traffic, both in the node and in the interconnection, may result in larger local and remote latencies.\(^2\) Therefore, depending on the value \(D_p\), with respect to either the new local or remote latencies, the data will arrive on time or late with respect to the use. In the former case, illustrated in Fig. 2c, the program does not experience a delay. In Fig. 2d the processor has to stall by the difference between the actual latency and the prefetch distance \(D_p\). The following two equations give the corresponding stall times for local and remote prefetches:

\[
T_{s,1} = \begin{cases} 
q(L_i - D_p), & \text{if } D_p \leq L_i, \\
0, & \text{if } D_p > L_i.
\end{cases}
\]

\[
T_{s,2} = \begin{cases} 
(1 - q)(L_r - D_p), & \text{if } D_p \leq L_r, \\
0, & \text{if } D_p > L_r.
\end{cases}
\]

\(D_p\) is related to the static prefetch distance by the following equation:

\(^2\) In this paper, we did not consider changes in the local latency.
\[ D_p = \frac{R + I(f)}{R} D_p^* = \frac{T_p(f)}{RM} D_p^*, \]  

where \( T_p(f) \) was specified in Eq. (2).

3.2. Destructive Effects of Prefetching

The amount of cache interference caused by prefetching depends on implementation details on how prefetches are handled by the cache controller. If the prefetch is useful (covers a miss), the cache controller has to allocate a slot in the cache for the line that is being prefetched. There are basically three different moments at which this can happen: (1) issue time, (2) arrival time, and (3) use time. The actual allocation time determines the amount of cache interference that is generated as a result of displacing a useful line from the cache.

Of the three options above, the issue time is the easiest to implement but may cause the largest amount of interference. The other two options require placing the pending prefetches in a prefetch buffer [Cray93]. This prefetch buffer reduced the amount of interference, but it has to be visible to the cache coherence protocol. A victim buffer, where displaced lines are kept for some time with the expectation that they will be referenced again, will diminish the amount of interference. However, the victim buffer as well as the prefetch buffer have an inherent cost because they need to be visible to the coherency protocol.

In the analysis of the next section, we assume that prefetches are allocated in the cache at issue time. We make this assumption because we are interested in finding the maximum amount of interference produced by prefetching as well as quantifying how this affects the effectiveness of prefetching.

3.2.1. Intranoide Interference. First, let us define the window of interference as the interval of time from the moment a line is displaced from the cache to make room for a pending prefetch until the prefetched line is referenced. Now, intranode interference, with respect to the original execution, occurs when either one of the following situations happens: (1) An extra miss is generated that was a hit in the original execution. This hit is with respect to the original execution; or (2) a prefetch is canceled by an access (hit or miss) or another prefetch. When an original access (hit or miss) collides with a prefetch, the access becomes a miss.

Let \( P_{\text{miss}} \) be the probability of an extra miss and \( P_{\text{can}} \) be the probability that a prefetch is canceled. To compute both probabilities, we need to know how many memory references (hits and misses) are executed within a window of interference. We know, by definition, that when prefetches are allocated at issue time the window of interference equals the prefetch distance. We also know that under the original execution each run length ends with a cache miss. Therefore, within a window of interference, the number of cache misses \( n_{\text{miss}} \) is \( D_p^* / R \), and, assuming a hit ratio of \( h \), the number of cache hits \( n_{\text{hit}} \) is \( n_{\text{miss}} h (1 - h) \), while the number of memory references \( n_{\text{acc}} \) is \( n_{\text{miss}} + n_{\text{hit}} \).

We assume that each hit (miss) is independent of other hits (misses). In reality, due to spatial and temporal locality, there is a high probability that a location and its corresponding cache line will be referenced again. In [Thie89] the number of unique locations \( u(n) \) (the footprint) present in a sequence of \( n \) references is given by \( Kn^{1/\theta} \), where \( K \) and \( \theta \) are related to the working set and spatial locality of the program. To use this approximations here, all references to parameters \( n_{\text{hit}}, n_{\text{miss}}, \) and \( n_{\text{acc}} \) have to be replaced by \( u(n_{\text{hit}}), u(n_{\text{miss}}), \) and \( u(n_{\text{acc}}) \).

Figure 3 illustrates the scenario of a prefetch changing a hit into a miss. As the figure shows, the extra miss occurs only when a hit to location \( j \) is changed into a miss by a prefetch to \( i \) that displaces \( j \) from the cache and is executed before \( j \) in an attempt to cover a miss that occurs after \( j \). Now, assuming a direct-mapped cache having \( b \) lines, the probability that one of the \( n_{k} \) hits executed inside the window of interference will map to the same cache line as the corresponding prefetch is \( P_{\text{miss}} = 1 - (1 - 1/b)^{n_k} \).

Figure 4 illustrates the two possible situations that can result in a prefetch being canceled. In the figure, an access refers to either a hit or a miss. A prefetch \( i \) can be canceled either when a memory access \( j \) that has a cache conflict with \( i \) is executed inside \( i \)'s window of inter-

![Diagram](image-url)
ence, or when another prefetch conflicting with i is executed within i’s window of interference.

Let $P_{\text{can},a}$ and $P_{\text{can},b}$ be the respective probabilities of these two events happening, then the probability that a prefetch will be canceled is $P_{\text{can}} = P_{\text{can},a} + P_{\text{can},b} - P_{\text{can},a}P_{\text{can},b}$, where $P_{\text{can},a} = 1 - (1 - 1/b)^{n_{\text{can}}}$ and $P_{\text{can},b} = 1 - (1 - 1/b)^{n_{\text{can}}}$. Exponent $n_{\text{miss}}$ represents the number of prefetches executed during the window of interference.

In this analysis, we have assumed that when two prefetches collide in the cache, the first one is lost, which is easier to implement. Another alternative is to drop the second prefetch. Although, in both cases at least one prefetch is lost, the second alternative avoids sending a useless message, as long as the sending of messages occurs after their conflicts are detected/resolved.

3.2.2. Internode Interference. Independently of the time when a prefetched is allocated into the cache, exclusive prefetches will generate invalidations which, on other nodes, have the potential of causing new misses or of canceling prefetches. Although these invalidations were also sent in the original execution, prefetching alters the times at which the invalidations are sent and received. Now, the actual moment when a read-exclusive is issued and invalidations are sent depends on the particular characteristics of the coherence protocols. However, independently of this, if a prefetch is executed $D_p$ cycles before a miss, the corresponding invalidations will also be sent and received $D_p$ cycles ahead of time. We assume that all nodes behave in the same way with respect to prefetching; i.e., each node sends prefetches at the same rate and they are equally effective. This means that although events on each node execute at different times with respect to the original execution, the relative times with respect to events on other nodes (using prefetching as well) do not change. Therefore, a hit on node $A$ in the original execution can be changed into a miss by an invalidation sent from a node $B$ and coming from a prefetch. This is only possible if the invalidation in the original execution arrived within an interval of time having the same length as the window of interference from the moment the hit occurred in $A$. Figure 5 shows this situation.

To compute the stall time due to remote interference, we use the following argument. Let $P_{\text{RX}}$ and $N_{\text{inv}}$ be, respectively, the probability of sending a read-exclusive memory request and the average number of invalidations that a request generates. The probability that a node will receive an invalidation originating from a request issued by another node is $P_{\text{RX}} N_{\text{inv}}/(N-1)$, where $N$ is the number of nodes in the machine. Now, a node can receive invalidations from any node. Therefore, using a similar argument as in the case of intranode interference, we can compute $P_{\text{can}}$, the probability of a prefetch being canceled by one of the $P_{\text{RX}} N_{\text{inv}}$ invalidations, as $P_{\text{can}} = 1 - (1 - 1/b)^{n_{\text{can}}} P_{\text{RX}} N_{\text{inv}}$.

Similarly, we can compute the probability that a single invalidation changes a hit into a miss as $P_{\text{miss}} = 1 - (1 - 1/b)^{n_{\text{miss}}}.

Finally, by considering that an extra miss (hit lost) and a prefetch canceled will stall the processor by either $L_L$ or $L_L$ cycles, we can compute the average amount of time wasted as a result of intra- and internode interference $T_{\text{ext}}$ as

$$T_{\text{ext}} = (qL_L + (1 - q)L_L)(N_{\text{can}} + N_{\text{miss}}).$$ (7)
where
\[
N_{\text{can}} = p_{\text{can}}^{(\text{intra})} + p_{\text{can}}^{(\text{inter})} - p_{\text{can}}^{(\text{intra})} \cdot p_{\text{can}}^{(\text{inter})}
\]
and
\[
N_{\text{miss}} = p_{\text{miss}}^{(\text{intra})} + p_{\text{RX}} \cdot n_{\text{miss}} \cdot p_{\text{miss}}^{(\text{inter})}.
\]

(8)

Term \(p_{\text{RX}} \cdot n_{\text{miss}} \cdot p_{\text{miss}}^{(\text{inter})}\) is the number of misses induced by internode interference. Given that the probability that a prefetch will cover an original miss is \(f\), then \(fT_{\text{ext}}\) in Eq. (2) represents the amount of extra time added by prefetching.

4. LATENCY OF REMOTE MEMORY ACCESSES

The latency experienced by a memory request requiring a remote transaction depends not only on the hardware characteristics of the interconnect and the amount of contention in it, but also on the program behavior, cache coherence protocol and memory consistency model. In this section, we derive an expression for the latency of remote transactions as a function of the main parameters of our model. We start by discussing how the cache consistency protocol and the memory consistency model affect the stall time that a processor suffers when triggering a remote request, and based on this we extend Agarwal's latency model for direct network.

We also take into account that a fraction of the memory is local to a node and the different lengths of network messages. To make the discussion more concrete, we use DASH’s coherence protocol and memory model as an example in our discussion. However, our results are not limited to DASH and the same arguments apply to other coherence protocols and memory models.

The coherence protocol specifies the number and nature of the messages that have to be exchanged between the nodes as a result of the processor executing loads and stores. From the performance perspective, the relevant issue here is the number of messages that are exchanged before a request can be considered complete and the processor is allowed to resume execution. This number, however, depends not only on the details of the coherence protocol, but also on the memory consistency model enforced by the hardware.

For example, in a sequential consistency model all messages triggered by a memory request have to be received in order to consider it complete and no memory operation can start until all previous ones have completed. The main difference between strong and relaxed memory models is that in the latter, the processor only waits for a critical fraction of the messages to be sent and received. Moreover, new requests can start even before previous ones have finished.

Let us define the critical set of a memory request as the number of messages that, according to both the cache coherence protocol and the memory consistency model, have to be satisfied before a node can be allowed to continue execution. In addition, let the total set be the number of all messages triggered by a request. It is relatively easy to identify the critical and total set of each memory request by looking at the details of both the coherence protocol and memory consistency model.

Distinguishing between the critical and total sets is important in computing the latency of request sets. It is number of messages in the critical sets which directly determine the remote latency of a request as observed from the processor (node), while the total sets affect the amount of traffic in the network. The contention associated to this traffic in turns delays messages in the critical set.

4.1. An Improved Model for Direct Networks

Our derivation starts with Agarwal’s network latency model, which we adapt to the presence of a cache coherence protocol and a relaxed memory model. In latter sections we extend the model to include the effects of data prefetching.

In [Agar91, Agar92] Agarwal presented a network model for direct networks, in particular for \(k\)-ary \(n\)-cubes. More recently, Dally also validated it experimentally against measurements taken from the J-machine.
In its simplest form, Agarwal's model is given by

\[ L_{\text{network}} = (T^s + T^w)(h_{\text{ops}} + B - 1 + h_{\text{ops}}w), \]  

(9)

where \( T^s \) and \( T^w \) are the switch and wire delays, \( h_{\text{ops}} \) is the average number of hops that a message takes from source to destination, \( B \) is the average length of a message in units of flits, and \( w \) is the average delay due to contention in the switches. The term \( w \) is given by

\[ w = \frac{\rho B}{(1 - \rho)} \frac{(k_d - 1)}{k_d^3} \left(1 + \frac{1}{n}ight), \]

(10)

where \( \rho \) is the channel utilization, \( n \) is the dimensionality of the network, and \( k_d \) is the average distance traveled by a message in each dimension. The number of hops taken by a message is \( h_{\text{ops}} = k_d \rho t \). Agarwal also derived an expression for the channel utilization \( \rho \) in terms of the probability \( m \) of a node injecting a message into the network in a given cycle as

\[ \rho = mBk_d. \]

(11)

### 4.2. Distributed Coherent Caches and Relaxed Memory Consistency

Agarwal’s formulas were derived assuming that all messages in the network have the same length \( B \). This is not true in the case of messages sent by cache coherent protocols, because some carry data in addition to protocol control information. For example, in DASH the critical set of a read transaction require sending either two or three messages. These messages can have different lengths; e.g., a reply contains, in addition to control information, data which in this case is at least as large as the cache line. Here we assume that there are only two types of messages: short and long messages.

Short messages include requests, invalidations, and acknowledgements which have a length of \( S_0 \). Long messages correspond to replies and contain, in addition to a short message, a cache line. The size of a long message is \( S_0 + b_c \), where \( b_c \) is the cache line size. Let \( \alpha \) (alternatively \( \beta \)) be the average number of messages in the critical (total) set triggered by remote load (load and store) instructions, \( W \) be the channel width in bytes, and \( B_\alpha \) and \( B_\beta \) be the respective average message lengths (in units of flits). Stores are not included in \( \alpha \), because they do not stall the processor. However, they are included in \( \beta \), because they contribute to the total traffic in the network. We can express the critical and total average message lengths as

\[ B_\alpha = \frac{aS_0 + b_c}{\alpha W} \quad \text{and} \quad B_\beta = \frac{bS_0 + b_c}{\beta W}. \]

(12)

The reader should recall that a load (store) miss is implemented by the coherency protocol using a non-exclusive (exclusive) read. Only a load stalls the processor, while the store only stalls the write buffer. In both cases a writeback can be triggered when a remote dirty line has to be displaced from the cache to make room for the load or store. To simplify the analysis, we ignore writebacks as they represent a small fraction of the traffic and consist of a single long message.

Now, a processor only stalls until all messages in the critical set are sent and received, hence the latency experience by the node as a result of a load is

\[ L_t = \alpha L_{\text{network}} + (\alpha + 1)T_{\text{proc}}, \]

(13)

where \( T_{\text{proc}} \) is the average processing time required to process a message inside a node. We will assume that \( T_{\text{proc}} \) has a constant value and is independent of the type of message involved. A better model will incorporate contention at the node. This will make a distinction between the different types of messages. The extra one for the processing time appears because the local node send the first message and consumes the last one.

To include the effects of message size in the network delay expression given above, the two \( B \)'s in Eqs. (9) and (10) have to be replaced by \( B_\alpha \). This is because these \( B \)'s refer to messages in the critical set. The \( B \) used Eq. (11), however, refers to all messages traversing that channel; therefore, this particular \( B \) has to be replaced by \( B_\beta \).

If we substitute Eq. (12) in (9)–(11) and in turn in Eq. (13), we can express the remote latency as

\[ L_t = L_0 + \frac{C_1}{(1/m)} - C_2, \]

(14)

where \( L_0 \), \( C_1 \), and \( C_2 \) are given by

\[ L_0 = \alpha(T^s + T^w)(k_d(n + B_\alpha - 1) + (\alpha + 1)T_{\text{proc}}, \]

(15)

\[ C_1 = (T^s + T^w)\alpha B_\alpha B_\beta (k_d - 1)(n + 1), \quad \text{and} \quad C_2 = B_\beta k_d. \]

(16)

\( L_0 \) is the static memory latency, which represents the delay of a read request in the absence of contention.

Another assumption made in Agarwal’s model is that each node send messages at a rate of \( m \), and that these messages do not generate responses from other nodes. In our case, exclusive and non-exclusive reads are generated by a node at the same rate in which cache misses requiring remote transactions occur. Each of these requests, however, triggers on the average other \( \beta - 1 \) messages corresponding to indirect requests to dirty nodes, replies, invalidations, and acknowledgements. Therefore, under the assumption that all nodes have similar behavior, a node injects messages at a rate \( \beta \) times higher than the rate at which remote cache misses occur.

If the bidirectional channels are implemented as two independent channels, then the rate per channel is only \( \beta/2 \) times higher. In machines such as DASH, however,
where a cluster contains several processors, the traffic injected into the channel has to be increased in proportion to the number of processors per cluster.

If we ignore prefetching, we can compute the value of \( m \), in this case \( m_n \), as follows. A program generates a load miss during one original execution interval of length \( R \) cycles. Recall that \( q \) (alternatively \( 1 - q \)) is the probability that a miss will be satisfied in the local (remote) node, hence the rate at which messages are sent is

\[
m_n = \frac{(1 - q)\beta}{R + I(0)} = \frac{(1 - q)\beta}{R + qL_t + (1 - q)L_t}.
\]

The term \( (1 - q)\beta \) is the average number of messages sent between two cache misses.

Replacing this expression in Eqs. (14) and (15) and solving for \( L_t \) gives the solution

\[
L_t = \frac{L_0}{2} + \frac{\beta B_\| k_d}{2} \frac{R + qL_1}{2(1 - q)} + \frac{\Delta^{1/2}}{2},
\]

where

\[
\Delta = \left( L_0 - \frac{\beta B_\| k_d}{2} + \frac{R + qL_1}{(1 - q)} \right)^2 + 4(T^* + T^* \alpha \beta B_\|^1 B_\| k_d \| k_d - 1)(n + 1).
\]

Extending the network model to include prefetching, requires modifying Eq. (17) in two ways. First, the denominator in Eq. (17) corresponding to \( T_n/M = R + I(0) \) has to be changed to \( T_p/M = R + I(f) \). Second, the numerator has to include the extra misses triggered by prefetching. Hence, the rate at which messages are injected into the network under prefetching \( m_p \) is

\[
m_p = \frac{(1 - q)(1 + f(N_{can} + N_{min}))\beta}{R + (f + g)V + f(T_{ext} + T_{s,i} + T_{e,i}) + (1 - f)(qL_t + (1 - q)L_t)}.
\]

Recall that \( N_{can} \) and \( N_{min} \) are the average number of intra and internode canceled prefetches and extra misses triggered by prefetches in an execution interval (Eq. (8)). Using the same approach as in the previous paragraph allows us to compute the remote latency under prefetching.

### 5. Validation Experiments Against DASH Prototype Measurements

Here we evaluate the accuracy of some estimates obtained using the remote latency model derived in the last section against experimental measurements reported on the DASH prototype. Because the experimental results were taken for program not using prefetching, we use Eqs. (18) and (19) to compute the latency of remote transactions. Other performance metrics are shown on three applications as a function of the number of processors. The DASH architectural parameters and relevant program characteristics used were taken from the following sources [Leno90, Gupt92, Leno92].

Some minor modifications to our model are required in order to adapt to the particular characteristics of DASH. The most important ones include the following: (1) Each DASH cluster supports four processors; (2) there are two independent interconnects (requests and replies); (3) each bidirectional channel is implemented as two independent channels; (4) DASH’s interconnects reported in [Leno92] consist of \( 4 \times 2 \) meshes; and (5) all cache-to-cache and cache-to-local memory transactions as well as inbounding remote messages use the memory bus.

Dealing with issues (1)–(3) is straightforward. The rate at which messages are sent by a single DASH cluster \( m_{DASH} \) is four times higher than the rate given by Eq. (17). On the other hand, the two interconnects and two independent channels present in each point to point connection reduce the channel utilization by a factor of four. These two results give: \( \rho_{DASH} = m_{DASH} B_\| k_d^2/4 = mB_\| k_d \rho \).

Accounting for the fact that DASH’s interconnects are \( 4 \times 2 \) meshes requires modifying Eq. (10). The average number of hops per dimension \( k_d \) taken by a message in a \( 4 \times 2 \) mesh without end-around connections and with bidirectional channels is \( 7/8 = 0.875 \). Equation (10), however, gives a negative value when \( k_d < 1 \), implying that contention reduces network latency! This result is clearly incorrect. The problem arises by the fact that Eq. (10) is an approximation that is valid only in networks where the average radix is greater than 3. In low radix networks, however, the following exact expression for \( w \) should be used:

\[
w = \frac{\rho B}{(1 - \rho)k_d^3} \left( (k_d - 1) \left( 1 + \frac{2}{n} - \frac{1}{n^2 k_d} \right) + \frac{1}{n - \frac{1}{n^2 k_d}} \right).
\]

According to this equation, for interconnects where \( k_d \) equals one only the first term inside the outermost big parenthesis is zero.

Finally, the fact that all memory requests coming from the cluster’s processing nodes or from the network make use of the memory bus and access the directory requires us to model contention in these resources. To account for this extra contention we modified Eq. (15) by replacing \( T_{proc} \) with

\[
T_{proc} = T_{proc}(0) \left( 1 - \rho_{bus}/2 \right).
\]

where, \( T_{proc}(0) \) is the average processing time per message without any contention and \( \rho_{bus} \) is the bus utilization. The delay given by Eq. (22) assumes that a deterministic server with exponential arrival times (M/D/1). The
bus utilization can be computed from the bus transaction rate which is given by

$$m_{bus} = \frac{P_{clus}(q + (1 - q)/\beta)}{R + qL_t + (1 - q)L_r},$$

where $P_{clus}$ is the number of processors per cluster. The resulting model can now be used to estimate the latency of remote transactions of a particular application.

All parameters in the model, except for $T_{peak}(t)$, $\alpha$, $\beta$, and $N_{inv}$, can be extracted from the architectural descriptions given in the DASH papers. Parameters $\alpha$, $\beta$, and $N_{inv}$ are program dependent. The three programs we use for comparison are Barnes–Hut, Water, and LocusRoute. Barnes–Hut models the dynamic evolution of galaxies under gravitational forces; Water is a molecular dynamics code from the field of computational chemistry; and LocusRoute is a standard-cell placement tool that uses area to evaluate the quality of a placement.

A description of these applications can be found in [Sing91, Gupta92, Leno92]. Most of the relevant program statistics used here were extracted from experimental measurements reported in [Leno92], while information about the amount of invalidations sent by the programs were obtained from [Gupta92]. Invalidation results for Barnes–Hut were not reported in [Gupta92]. In this program, the number of invalidations was assumed as the average of the other two programs. Although, there is no basis for this assumption, we have found that the number of invalidations have a negligible effect on the remote latency.

In Tables II–IV we give our estimates, DASH’s measurement and overall results for each program. Each table contains three parts. The first part gives the estimates produced by our model for $\alpha$, $\beta$, the processor utilization, bus utilization, and remote memory latency. The second part shows the corresponding measurements as reported by the DASH group, while the third part indicates the relative error of the predicted latency with respect to the experimental measurement.

The results for Barnes–Hut and Water show a good match between the predicted and experimental measurements, with the remote latency errors ranging from $-5.9$ to $+5.4\%$. For LocusRoute, however, the error is small for up to 16 processors, but it increases significantly to $-11.0$ and $-27.6\%$ for 24 and 32 processors. This large discrepancy is due to the presence of a hot spot in LocusRoute [Leno92]. The cost array used to evaluate placement is allocated in a particular cluster and all nodes have to read the array from it. Therefore, the bus utilization on this cluster is much higher than on other clusters reaching more than $65\%$ on 32 processors. Our model, however, assumes a uniform load across clusters. Overall, ignoring the anomaly for LocustRoute, our estimates show a very good match against the experimental measurements.

Figure 6 shows the normalized components of the remote memory latency. These components are (1) cluster processing time, (2) network latency, (3) cluster and network contention. Although, the first two components assume no contention, they are not constant because they depend on $\alpha$ which is program dependent.

6. LIMITS AND EFFECTIVENESS OF PREFETCHING

The effectiveness of prefetching is analyzed below in terms of several program and machine variables. We set values to all the parameters in the model, which collectively we refer as the base configuration. The set of val-

---

### Table II

Predictions and DASH Experimental Results for Barnes–Hut

<table>
<thead>
<tr>
<th>Execution attribute</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages in critical set: $\alpha$</td>
<td>2.091</td>
<td>2.064</td>
<td>2.057</td>
<td>2.055</td>
<td>2.053</td>
</tr>
<tr>
<td>Messages in total set: $\beta$</td>
<td>2.339</td>
<td>2.293</td>
<td>2.300</td>
<td>2.304</td>
<td>2.313</td>
</tr>
<tr>
<td>Est. processor utilization</td>
<td>0.942</td>
<td>0.931</td>
<td>0.936</td>
<td>0.930</td>
<td>0.929</td>
</tr>
<tr>
<td>Bus utilization: $\rho_{bus}$</td>
<td>0.037</td>
<td>0.047</td>
<td>0.045</td>
<td>0.052</td>
<td>0.053</td>
</tr>
<tr>
<td>Remote latency: $L_t$ (cycles)</td>
<td>107.3</td>
<td>106.8</td>
<td>106.3</td>
<td>106.5</td>
<td>106.5</td>
</tr>
<tr>
<td>Est. processor utilization</td>
<td>0.942</td>
<td>0.929</td>
<td>0.934</td>
<td>0.925</td>
<td>0.923</td>
</tr>
<tr>
<td>Bus utilization: $\rho_{bus}$</td>
<td>0.098</td>
<td>0.110</td>
<td>0.102</td>
<td>0.109</td>
<td>0.110</td>
</tr>
<tr>
<td>Remote latency: $L_t$ (cycles)</td>
<td>104.2</td>
<td>106.2</td>
<td>109.1</td>
<td>110.9</td>
<td>113.1</td>
</tr>
<tr>
<td>Remote latency pred. error (%)</td>
<td>+2.98</td>
<td>+0.56</td>
<td>-2.57</td>
<td>-3.97</td>
<td>-5.84</td>
</tr>
</tbody>
</table>

### Table III

Predictions and DASH Experimental Results for Water

<table>
<thead>
<tr>
<th>Execution attribute</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages in critical set: $\alpha$</td>
<td>2.092</td>
<td>2.183</td>
<td>2.270</td>
<td>2.480</td>
<td>2.500</td>
</tr>
<tr>
<td>Messages in total set: $\beta$</td>
<td>2.996</td>
<td>3.179</td>
<td>3.327</td>
<td>3.678</td>
<td>3.696</td>
</tr>
<tr>
<td>Est. processor utilization</td>
<td>0.907</td>
<td>0.884</td>
<td>0.879</td>
<td>0.866</td>
<td>0.857</td>
</tr>
<tr>
<td>Bus utilization: $\rho_{bus}$</td>
<td>0.099</td>
<td>0.140</td>
<td>0.152</td>
<td>0.185</td>
<td>0.195</td>
</tr>
<tr>
<td>Remote latency: $L_t$ (cycles)</td>
<td>109.3</td>
<td>113.4</td>
<td>116.6</td>
<td>124.6</td>
<td>125.9</td>
</tr>
<tr>
<td>Est. processor utilization</td>
<td>0.932</td>
<td>0.911</td>
<td>0.906</td>
<td>0.892</td>
<td>0.880</td>
</tr>
<tr>
<td>Bus utilization: $\rho_{bus}$</td>
<td>0.145</td>
<td>0.13</td>
<td>0.166</td>
<td>0.177</td>
<td>0.188</td>
</tr>
<tr>
<td>Remote latency: $L_t$ (cycles)</td>
<td>104.5</td>
<td>108.1</td>
<td>111.6</td>
<td>118.3</td>
<td>120.6</td>
</tr>
<tr>
<td>Remote latency pred. error (%)</td>
<td>+4.60</td>
<td>+4.90</td>
<td>+4.48</td>
<td>+5.32</td>
<td>+4.39</td>
</tr>
</tbody>
</table>

### Table IV

Predictions and DASH Experimental Results for LocusRoute

<table>
<thead>
<tr>
<th>Execution attribute</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages in critical set: $\alpha$</td>
<td>2.029</td>
<td>2.208</td>
<td>2.280</td>
<td>2.336</td>
<td>2.350</td>
</tr>
<tr>
<td>Messages in total set: $\beta$</td>
<td>2.835</td>
<td>3.078</td>
<td>3.233</td>
<td>3.334</td>
<td>3.372</td>
</tr>
<tr>
<td>Est. processor utilization</td>
<td>0.762</td>
<td>0.725</td>
<td>0.696</td>
<td>0.677</td>
<td>0.667</td>
</tr>
<tr>
<td>Bus utilization: $\rho_{bus}$</td>
<td>0.240</td>
<td>0.286</td>
<td>0.333</td>
<td>0.372</td>
<td>0.398</td>
</tr>
<tr>
<td>Remote latency: $L_t$ (cycles)</td>
<td>115.5</td>
<td>126.0</td>
<td>130.6</td>
<td>133.2</td>
<td>132.5</td>
</tr>
<tr>
<td>Est. processor utilization</td>
<td>0.791</td>
<td>0.731</td>
<td>0.695</td>
<td>0.648</td>
<td>0.594</td>
</tr>
<tr>
<td>Bus utilization: $\rho_{bus}$</td>
<td>0.308</td>
<td>0.361</td>
<td>0.382</td>
<td>0.385</td>
<td>0.368</td>
</tr>
<tr>
<td>Remote latency: $L_t$ (cycles)</td>
<td>109.1</td>
<td>119.6</td>
<td>128.7</td>
<td>149.6</td>
<td>183.0</td>
</tr>
<tr>
<td>Remote latency pred. error (%)</td>
<td>+5.87</td>
<td>+5.35</td>
<td>+1.48</td>
<td>-10.96</td>
<td>-27.60</td>
</tr>
</tbody>
</table>
ues used in the base configuration are the same as those presented in Table I (Section 2).

We investigate the effects that the following variables have on prefetching: the static prefetch distance $D_p^*$, the run length $R$, the coverage factor $f$, the cache line $b_c$, the remote to local request ratio $q$, and the number of nodes $N$. In the cases where we scale the machine size $N$ from 64 to 4096 nodes, we also increase the channel width $W$ from 2 to 8 bytes, in order to maintain the same bisection utilization.

6.1. Effects of Prefetching on the Remote Latency

Here we present results about the sensitivity of the effective remote latency with respect to the static prefetch distance. Memory tolerating techniques, like prefetching and multithreading, have the peculiar characteristics that they tend to increase the magnitude of the remote latency in proportion to how successful they are in tolerating it. This has been called the feedback effect by Johnson [John92], and it is a consequence of the fact that both prefetching and multithreading tolerate latency by increasing the rate at which remote requests are injected into the network.

With respect to prefetching, this induced latency reaches its maximum at the point in which all prefetches arrive to the local cache just in time for the processor to reference the data. After this, no more latency can be hidden, therefore increasing $D_p^*$ can only decrease the magnitude of the remote latency by increasing the intranode and internode cache interference. The seemingly paradoxical result of more cache interference reducing the magnitude of the remote latency is resolved by observing that the extra cache interference stalls the processor for the full amount of the latency. This has the net effect of reducing the network utilization, which in turn result in less latency through less contention.

The graphs in Figure 7 show that both the cache line size $b_c$, and to a lesser degree the run length $R$, have a stronger effect in the latency than the coverage factor $f$ or the fraction of remote requests $q$. Larger cache lines increase the basic network latency and induces more contention at the switches. The first effect is evident on Fig. 7a in the magnitude of the remote latency as $D_p^*$ approaches zero, while the second effect is represented by a higher positive slope in the left half of the graph.

More important than the absolute size of the cache line is the ratio $b_c/W$; the number of flits that a cache line occupies. If we ignore the relation between the cache line and the run length, then increasing the size of the line doubles the channel and bisection network utilization. The only effective way of keeping the utilization of both constant is by either doubling the channel width or by halving the point to point wire and switch delays.

The run length $R$ affects the remote latency by increasing the rate at which messages are injected in the network. This results in higher channel utilization and node contention. Figure 7b indicates that, given the characteristics of our base machine, when $R < 20$ the channel utilization is high enough that hiding more latency with prefetching have a significant impact in the latency.

Scaling the size of the machine from 64 to 4096 nodes and under prefetching, without changing the network dimensionality, channel width, and the wire and switch delays, puts a significant stress on the network. Increasing the network radix from $k_0$ to $k_1$ increases the bisection utilization in proportion to $k_1/k_0$. This happens because the bisection traffic grows as a function of $O((k_1/k_0)^{a-1})$, while the bisection bandwidth grows as $O((k_1/k_0)^{a-1})$. In addition, messages have to travel a larger distance in each dimension, so contention increases. Figure 7d, however, shows that even with a channel width four times wider, on a machine with 4096 processors, prefetching significantly increases the magnitude of the remote latency.
FIG. 7. The effects of various parameters on remote latencies. Except for those numbers indicated in the figures, all other parameters are based on Table I.
FIG. 8. The effects of various parameters on the idle time in each original execution interval. Except for those numbers indicated in the figures, all other parameters are based on Table I.
6.2. Effectiveness of Prefetching in Reducing Idle Time

Figure 8 shows how fast the idle time $I(f)$ decreases as a function of the prefetch distance. The magnitude of the slope indicates how effective (or ineffective) is prefetching in hiding the remote latency. The most interesting result comes from Fig. 8c which shows that, even when short run lengths increase the remote latency more than long run lengths (Fig. 7c), prefetching somehow manages to hide not only the original latency, but also the extra latency resulting from a higher cache miss rate.

For example, when the prefetch distance is 100 cycles, Fig. 8c shows that the respective remote latencies for run lengths of 10 and 20 cycles are 158 and 142 cycles. These remote latencies translate into 89 and 81 cycles of idle time, assuming $q = 0.5$ and $L = 20$ cycles. Now, Fig. 8c clearly shows that the residual idle time not covered by prefetching is, in both cases, 21 cycles. This implies that prefetching hides 68 and 60 cycles of the original 89 and 81 cycles of idle time. Therefore, we can conclude that even when the combination of a higher miss rate with prefetching results in a higher induced remote latency, prefetching manages to hide most of it.

As we mentioned in Section 3, each original execution interval consists of a run length followed by either a local or remote latency. Under prefetching, this execution interval contains several other components. Figure 9 shows the normalized components of a single prefetching execution interval. Regions “Uncovered” and “Residual Idle” represent, respectively, misses for which prefetches were not issued and the latency left over when prefetches arrive after the location is referenced. Figure 9a shows prefetching hiding most of the latency of covered misses. The prefetch overhead and intranode interference represent less than 15% of the time. In contrast, the Figs. 9b and 9c exhibit a utilization of only 50%, and the cache interference in Fig. 9b, both intra- and inter-node, accounts for close to 60% of the ungauged utilization.

6.3. Impact of Prefetching on the Execution Time

Figure 10 shows the magnitude of the speedups produced by prefetching. With respect to the coverage factor and remote request probability, the results in Figs. 10e–f show that as the fraction of misses covered by prefetching increases, the amount of idle time decreases. The same effect is observed by increasing the fraction of the misses that require remote communication. Figures 10a and 10c–d, however, show that these improvements can only be obtained if there is enough extra bandwidth to absorb the larger network traffic resulting from prefetching.

An interesting result which can be extracted from all curves is that the optimal value for $D_p^*$, the point where the speedup reaches its maximum value, is always smaller than $L_0$ (show as a dotted vertical line in the figures). This result seems to imply that it is better for the compiler to place prefetches under the assumption that the remote latency is lower than the static remote latency. The resulting effective prefetch distance, though, will grow due to the residual idle time, uncovered misses, and the cache interference induced by prefetching to match the run-time remote latency. Furthermore, the difference between the optimum and the expected prefetch distance needed to cover the run-time latency grows when either the miss rate or the cache line to channel width ratio increase.

We believe the explanation for this behavior has to do with the fact that increases in cache miss rates or line sizes result in larger effective prefetch distances, but the corresponding increases in the remote latency or in the amount of cache interference is lower. This is shown in Fig. 11, where the effective prefetch distance $D_p$, remote

![FIG. 9. Normalized components in an execution interval.](image-url)
FIG. 10. The effects of various parameters on speedup. Except for those numbers indicated in the figures, all other parameters are based on Table I. The dotted lines represent the points where $D_p^c$'s match the values of the respective $L_c$'s.
7. OPTIMIZATION OF THE PREFETCH DISTANCE

The results presented in the previous section clearly show that the particular value of the optimal static prefetch distance which maximizes the amount of execution time improvement or speedup is very sensitive to several run-time parameters. Because compilers do not have access to run-time information, they have to use other methods to compute the distance between the prefetches and their corresponding references. In this section we discuss several possible approaches and attempt to evaluate their effectiveness.

7.1. Constant Prefetch Distance

The simplest estimate for the optimal static prefetch distance and the one most used by compilers is the remote latency without contention [Mowr92]:

$$D_p^{\text{opt}} = L_0.$$  (24)

This estimate is based on the notion that prefetching hides all remote latency and induces an insignificant amount of extra contention. Consequently, prefetches have to be issued $L_0$ cycles from the misses they attempt to cover. To account for contention and other factors, sometimes a larger remote latency is assumed. The approximation above, however, ignores important effects like the run length, network contention, coverage factor, and the distribution of misses.

7.2. Program-Dependent Prefetch Distance

A better estimate is obtained by assuming that the compiler has perfect information of the steady-state run-time behavior of the nested loops for which it is going to insert prefetches. We start by ignoring the cache interference effects induced by prefetching, i.e., $T_{\text{ext}} = 0$. Figure 12 indicates that the maximum speedup is achieved when the effective prefetch distance $D_p$ equals the effective remote latency $L_r$. From this result the compiler decides where to insert prefetches using the following equation, which is obtained by combining Eqs. (2) and (6):

$$D_p^* = \frac{R \cdot L_r}{R + I(f)}$$

$$\approx \frac{R \cdot L_r}{R + (f + g)E + (1 - f)(qL_r + (1 - q)L_t)}.$$  (25)

In reality, $R$, $q$, $f$, $g$, and $L_r$ are not known at compile time. The compiler, however, can derive an estimate for $R$ from the dataflow information used to decide which data to prefetch. Current state-of-the-art compilers do this by detecting which references have a high probability of missing on a given cache. This is accomplished by applying locality analysis to the loops and determining: (1) the amount of data reuse within the loops, and (2) the set of reuses that can be exploited by a cache of a particular size [Mowr92].
FIG. 12. The effects of various parameters on channel utilization. Except for those numbers indicated in the figures, all other parameters are based on Table I.
In general, references can exhibit spatial, temporal, or group reuse. Once the compiler finds out which references in the loop can potentially be misses, it also knows the rate at which cache misses are expected to happen. The inverse of the miss rate represents an estimate, albeit imperfect, of the expected run length \( R_{est} \). Now, if \( R_{est} \) is not very small (i.e., \( m_{cache} \) is not very large), then the impact of prefetches in the remote latency should be quite small, so \( L_r \) can be approximated by \( L_0 \), the remote latency without contention. In addition, some knowledge about how the data is allocated allows the compiler in estimating \( q \).

We still need to obtain estimates for \( f \) and \( g \). Because compilers attempt to cover all misses and avoid inserting unnecessary prefetches, the information derived from the locality analysis will not shed light on the actual effectiveness of prefetching. An alternative approach is to use heuristics based on program static characteristics complemented with a rule-of-thumb obtained from simulations about the effectiveness of prefetching. For example, Mowry et al. [Mowry92] have reported on the effectiveness of two prefetching algorithms: selective and indiscriminate. Both algorithms try to cover all misses with prefetches, but the second one makes no effort in avoiding the insertion of unnecessary prefetches.

The simulation results show average coverage factors of 65.5% (selective) and 69.2% (indiscriminate). In both cases the variance was quite large. The selective algorithm, however, was significantly better in avoiding unnecessary prefetches. Combined with some heuristics, these statistics can be used by the compiler to estimate \( f \) and \( g \). As more experience is gathered about the true effectiveness of prefetching algorithms, better estimates should be obtained.

Combining all these estimates we can now propose a new approximation for the optimal static prefetch distance as

\[
D_p^{(2)} = \frac{R_{est} \cdot L_0}{R_{est} + (f_{est} + g_{est})V + (1 - f_{est})} \cdot \frac{V}{q_{est}L_1 + (1 - q_{est})L_0}. \tag{26}
\]

A further improvement to Eq. (26) can be obtained by using \( R_{est} \) on the equations for the \( L_r \), derived in Section 4.2. From this, we can rewrite Eq. (25) as

\[
D_p^{(3)} = \frac{R_{est} \cdot L_0(R_{est}, f)}{R_{est} + (f + g)V + (1 - f)} \cdot \frac{V}{q_{est}L_1 + (1 - q_{est})L_0(R_{est}, f)}. \tag{27}
\]

Figure 13 shows how close to the optimal prefetch distance are the three estimates discussed in this section (Eqs. (24)–(27)). The bands shown in some of the figures correspond to ranges of estimates assuming that \( R_{est} \) has the potential of being underestimated or overestimated by 20%. As the results show, the estimates obtained using Eqs. (26) and (27) are quite close to the actual optimal values.

### 7.3. Adaptive Prefetch Distance Based on Performance Information

The above approximations depend on the compiler being able to obtain reasonable estimates about run-time performance parameters. In most situations, this is not possible due to the intrinsic complexity of the performance space. Basically, the range of performance exhibited by many programs running on multiprocessors is much wider than the corresponding performance on uniprocessors or vector machines. Furthermore, as processors get faster and faster and as the size of machines increase, this performance space becomes even larger. The big challenge for programmers and compilers is how to make sure that applications execute at the high end of the performance space.

The first step towards exploiting performance consists of understanding the conditions that affect it. For this purpose hardware performance monitors have been incorporated in several multiprocessors and have been used to analyze the complex interactions affecting the run-time behavior of applications. Some multiprocessors, like the Cray X-MP and Y-MP series, KSR1, and Stanford DASH even make this information available to the programmer through a collection of library routines [Wil90, KSR92, Leno92].

In most cases, however, only postmortem performance data (analyzed after the program has finished) has been used by programmers to improve their applications. In other cases it is the operating system who uses this type of data to make system-wide decisions about allocation and/or migration of processes and memory pages, load balancing, etc. We believe that compilers, or more specifically optimizers, should be the primarily consumers of the performance data offered by hardware monitors.

We are currently investigating how to make compilers and performance monitors work together at run-time in improving the performance of programs. Achieving this requires adapting and/or developing new optimization techniques which incorporate in their analysis, run-time program and machine performance information, as well as compile-time analysis.

We distinguish between two types of optimizations based on the use of performance data: dynamic optimization and adaptive optimization. In dynamic optimization, the compiler generates a parameterized version of the program that is instantiated at run-time using both program and performance dynamic information. In adaptive optimization the optimization parameters are reevaluated and adjusted with a certain frequency to reflect run-time variations of the relevant machine parameters.

Figure 14 illustrates how adaptive optimization could be used to modify the prefetch distance in an attempt to
FIG. 13. Accuracy of the three different prefetch distance estimates with respect to the cache line size (left) and run length (right). "Constant Distance" refers to Eq. (22). The intervals shown for "Static Estimate" (Eq. (24)) and "Semi-Dynamic Estimate" (Eq. (25)) are computed assuming that $R_{cm}$ can be overestimated or underestimated by 20\%.
C = 0;
for (inx = 0; inx < max; inx ++)
    C += A[inx] * B[inx];
end_for;

(a) Dot Product: Original Version

Pref_Dist = /* Prefetch distance is constant */
C = 0;
/* Prologue of Software Pipelining */
for (inx = 0; inx < Pref_Dist; inx ++)
    Prefetch (A[inx]);
    Prefetch (B[inx]);
end_for;

/* Body of Software Pipelining */
for (inx = 0; inx < max-Pref_Dist; inx ++)
    C += A[inx] * B[inx];
    Prefetch (A[inx] + Pref_Dist);
    Prefetch (B[inx + Pref_Dist]);
end_for;

/* Epilogue of Software Pipelining */
for (inx = max-Pref_Dist; inx < max; inx ++)
    C += A[inx] * B[inx];
end_for;

(b) Software Pipelining

Pref_Dist = /* First approximation of distance */
C = 0;
/* Prologue of Software Pipelining */
for (inx = 0; inx < Pref_Dist; inx ++)
    Prefetch (A[inx]);
    Prefetch (B[inx]);
end_for;

/* Body of Software Pipelining */
for (inx = 0; inx < max-Pref_Dist; inx ++)
    C += A[inx] * B[inx];
end_for;

disp = Abs(D_Pref_Dist);
if (D_Pref_Dist < 0) then /* Decrease distance */
    for (inx = jnx; inx < jnx + Abs(D_Pref_Dist); inx ++)
        C += A[inx] * B[inx];
    end_for;
    disp = Abs(D_Pref_Dist);
end_if;

if (D_Pref_Dist > 0) then /* Increase distance */
    for (inx = jnx; inx < jnx + D_Pref_Dist; inx ++)
        Prefetch (A[inx] + Pref_Dist);
        Prefetch (B[inx + Pref_Dist]);
    end_for;
    disp = 0;
end_if;

Pref_Dist += D_Pref_Dist;
for (inx = jnx + disp; inx < jnx + inc_interval; inx ++)
    C += A[inx] * B[inx];
    Prefetch (A[inx] + Pref_Dist);
    Prefetch (B[inx + Pref_Dist]);
end_for;

/* Epilogue of Software Pipelining */
for (inx = max-Pref_Dist; inx < max; inx ++)
    C += A[inx] * B[inx];
end_for;

(c) Adaptive Software Pipelining

FIG. 14. Static and adaptive software pipelining. In part (a) we show the code to compute the dot product between two vectors as originally written by the programmer. If static software pipelining is used to hide the latency of vectors A and B, the original code is transformed into three disjoint loops: prologue, body, and epilogue (as shown in part (b)). Note that the prefetch distance Pref_Dist remains constant for the duration of the pipeline. In adaptive software pipelining (part (c)) the loop implementing the pipeline body is broken into several execution segments (outermost loop). At the beginning of each segment the effectiveness of the prefetch distance is evaluated and based on this an increment or a decrement to the distance is computed (D_Pref_Dist). The first two loops modify the prefetch distance in the direction of the change, while the third loop executes the rest of the computation under a new prefetch distance. The code assumes that D_Pref_Dist <= inc_interval.

hide as much remote latency as possible without incurring too much cache interference. We assume that the remote latency changes in unpredictable ways, as a result of the activities of all the nodes in the system. Figure 14a represents the original code which computes the dot product between two vectors. If the compiler concludes that the two accesses to vectors A and B have a high probability of being misses, it can decide to use software pipelining to hide the latency (Fig. 14b). It consists of three disjoint loops: prologue, body, and epilogue. The body (steady-state section of the pipeline) executes the original code and prefetches values that will be needed Pref_Dist iterations in the future. The prologue starts the pipeline of prefetched, while the epilogue finishes the computation, after all values have been prefetched. In traditional software pipelining the prefetch distance is maintained constant for the duration of the pipeline.

In Fig. 14c, we show how the body of the software pipeline can be modified to support an adaptive prefetch distance. Instead of running the main body without interruption, the execution is broken into some number of intervals. At the beginning of each interval, code is executed to evaluate the effectiveness of the current prefetch distance. If the prefetch distance is too short (alternatively long), then a small loop is executed which increases (decreased) the distance by D_Pref_Dist iterations. After this change, the rest of the loop is executed under the new prefetch distance. Our research indicates
that three or four performance registers is all that is needed to conclude whether the prefetch distance has to be changed. We are currently evaluating the performance benefits which can be expected from adaptive optimization.

8. DISCUSSION AND CONCLUSIONS

One of our main results is that the effective prefetch distance is very elastic, i.e., it dynamically adjusts, within the availability of network bandwidth, to changes in the run length, coverage factor, and cache interference. This property of the prefetch distance may prove to be an important factor in favor of prefetching when compared with other latency tolerating techniques, in particular, multithreading. In multithreading, the proportion of the latency which is tolerated directly determines the processor utilization \( U_{\text{mthread}} \). Using \( C \) physical contexts the utilization, assuming the processor is not saturated, can be expressed as (recall that \( R = 1/m_{\text{cache}} \))

\[
U_{\text{mthread}} = \frac{C}{L_t + R} = \frac{C}{L_t \cdot m_{\text{cache}} + 1}. \tag{28}
\]

Now, we know that increases in cache miss rates and the remote latency, resulting from behavioral changes in the program with respect to physical and temporal locality, produce a corresponding increment in the magnitude of \( L_t \) and \( m_{\text{cache}} \) [Saav90, Agar92]. As a result of this, a lesser amount of latency is tolerated and processor efficiency drops. The only way to counterbalance this loss of efficiency is by increasing the number of contexts. Changing the number of physical contexts, however, is not a viable option.

The only other alternative is to virtualize the threads and share the processor among more threads than the number of physical contexts. An example of such a processor is APRIL which is used in the MIT Alewife prototype [Agar90]. Virtual multithreading, however, imposes the extra penalty of having to swap threads in and out of the processor. In addition the processor has to detect somehow variations in the cache miss rate, the remote memory latency, and the cache interference in order to dynamically adjust the number of virtual threads. Achieving this, without overly complicating the processor, is a challenging problem.

The performance model for prefetching presented here incorporates most of the relevant machine and program parameters expected to affect the effectiveness of prefetching. Results from the model indicate that prefetching can tolerate large latencies as long as the network bandwidth can support the expected increase in message rate. Moreover, our results also indicate that the cache miss rate and coverage factor are more important in determining the amount of improvement which can be derived from prefetching than the pernicious effects created by self and remote cache interference.

The effective prefetch distance is quite elastic with respect to dynamic changes in the cache miss rate, memory latency, and cache interference. This makes prefetching an attractive memory tolerating technique when compared with other approaches that are not as resilient to the complex performance interactions exhibited by parallel machines. Our optimization algorithm suggests the upgrade of the compiler to exploit prefetching. It is the job of compiler writers to develop new and better prefetching algorithms which can capture a larger fraction of the performance benefits.

ACKNOWLEDGMENT

We thank Jacqueline Chame from USC for her comments on earlier drafts of this paper.

REFERENCES


[John92] Johnson, K. L. The impact of communication locality on


RAFAEL H. SAAVEDRA is an assistant professor in the Computer Science Department at the University of Southern California. He received his Ph.D. in Computer Science from UC Berkeley in 1992.

WEIHUA MAO received his B.S. in computer science and M.S. in computer engineering from the Tsinghua University, Beijing, China, in 1985 and 1988, respectively. He received his M.S. in electrical engineering from the University of Southern California in 1991. He is now a Ph.D. candidate in computer engineering at the University of Southern California. His research topics include performance modeling and simulation of scalable multiprocessor systems.

KAI HWANG received his Ph.D. from the University of California at Berkeley. He is a professor of electrical engineering and computer science at the University of Southern California. He has engaged in computer research and teaching for 25 years. He is the author or coauthor of over 120 scientific papers, and 5 books in the areas of computer architecture, digital arithmetic, and parallel processing. His latest book, Advanced Computer Architecture: Parallelism, Scalability, Programmability (McGraw-Hill, New York, 1993), has been adopted by more than 40 universities in the United States. Dr. Hwang is the founding Coeditor-in-Chief of the Journal of Parallel and Distributed Computing. An IEEE Fellow and a distinguished ACM lecturer, his present research focuses on scalable multiprocessors using distributed shared memory. Presently, he collaborates with MIT Lincoln Laboratory and Motorola, Inc., on embedded parallel computing applications. He can be reached via E-mail: kaihwang@aloha.usc.edu.