Efficient shared-memory support for parallel graph reduction

Andrew J. Bennett
Paul H. J. Kelly

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate
London SW7 2BZ
United Kingdom

e-mail: `p.kelly@doc.ic.ac.uk`

July 21, 1996

Abstract

This paper presents the results of a simulation study of cache coherency issues in parallel implementations of functional programming languages. Parallel graph reduction uses a heap shared between processors for all synchronisation and communication. We show that a high degree of spatial locality is often present and that the rate of synchronisation is much greater than for imperative programs. We propose a modified coherency protocol with static cache line ownership and show that this allows locality to be exploited to at least the level of a conventional protocol, but without the unnecessary serialisation and network transactions this usually causes. The new protocol avoids false sharing, and makes it possible to reduce the number of messages exchanged, but relies on increasing the size of the cache lines exchanged to do so. It is therefore of most benefit with a high-bandwidth interconnection network with relatively high communication latencies or message handling overheads.

Keywords: cache organisation, simulation, declarative languages, shared-memory

1 Introduction

Parallel graph reduction (PGR) uses a shared graph structure to manage and synchronise the parallel execution of a functional program. It provides a simple model of communication and synchronisation between processors. The performance of parallel programs operating under this regime depends critically on the provision of access to the shared heap with very high average performance. In this paper we investigate the hypothesis that its usage patterns display distinctive properties of importance in the design and optimisation of multicache implementations of shared-memory.

The most successful implementations of PGR (*e.g.* Goldberg's Buckwheat system [19] and Augustsson and Johnsson's $\langle\nu, G\rangle$-machine [5]) have used general-purpose shared-memory multiprocessors based on snooping cache coherency protocols [4]. For modest numbers of processors these systems implement a shared heap with near-ideal performance, but the number of processors is limited by contention for the snooping bus.

Meanwhile there have been numerous attempts to implement PGR on more scalable architectures, interesting examples being Alfalfa [19], Alice [23], GRIP [37], and George's Butterfly implementation [18]. None of these has achieved performance nearly as satisfactory as Augustsson and Johnsson's $\langle \nu, G \rangle$-machine. A key reason for this has been the difficulty of communicating and sharing data structures.

Recent developments in the design of large shared-memory machines offer the prospect that an approach like the $\langle \nu, G \rangle$-machine might be scaled to large configurations. Large shared-memory systems, like the Kendall Square Research KSR1 [27], Stanford DASH [32] and DDM [21], rely on caches to reduce communications load, and use directory-based invalidation protocols to keep track of cached copies to prevent out-of-date copies from being used incorrectly. This paper studies the effectiveness of such coherency protocols on multiprocessors when executing parallel functional programs using PGR and identifies the factors limiting scalability. We investigate the use of standard protocol extensions which relax coherency, and study a modified protocol which exploits the characteristics of the PGR regime. We are particularly concerned with modern multiprocessors which have relatively high latency and high bandwidth interconnection networks since these characteristics greatly influence the behaviour and speedup of programs.

Preliminary results from this work were given in [7], and have been presented in conference papers [8, 9]. This paper reports results from a larger and more interesting benchmark suite, and includes more detailed analyses of simulation results, weak consistency and interconnection network bandwidth issues.

The remainder of the paper is structured as follows: graph reduction is reviewed in Section 2. The architecture of shared-memory multiprocessors and the major design issues are described in Section 3. The memory reference characteristics of PGR and our optimised coherency protocol are presented in Section 4. We then describe our simulation environment in Section 5. Simulation results are presented and analysed in Section 6. Further issues and related work are reviewed in Section 7, and we present our conclusions in Section 8.

2 Parallel graph reduction

The graph reduction model was first proposed by Wadsworth [45] and forms the basis of most modern compiled implementations of functional languages. Essentially the idea is that a parameter expression can be represented in unevaluated form by a "closure": a heap cell containing a code pointer and pointers to the variables on which the expression depends. In general the variables may in turn be closures, leading to a graph structure representing many tasks together with their data dependence structure. When the parameter is finally needed, the closure is activated: the code pointer is used as an entry point, and the rest of the closure as the environment for the evaluation of the suspended parameter expression. After evaluation, the closure is updated with a copy of the result, so that re-evaluation is not needed if another reference to the parameter is encountered.

Graph reduction implements a *call-by-need* parameter passing mechanism, *i.e.* the arguments of a function are only evaluated when their values are required. This laziness allows programs to manipulate infinite data structures: only as much of the data structure is evaluated as is necessary to produce the result of the program. Considerable efforts have been directed towards creating implementations of lazy evaluation which are as efficient as possible. However, it is usually the case that call-by-need is more costly than call-by-value and call-by-reference used in imperative languages due to the overhead of allocating and updating closures in the heap. For this reason call-by-value is used in place of call-by-need whenever possible by using the results of program analyses.

Given this structure, parallelism is easily added. At any time a closure can be "sparked", *i.e.* added to a pool of work to be distributed to threads executing on idle processors. The graph structure represented by the closures is then used to coordinate and communicate between evaluation processes. When a thread needs the value of some other closure, the closure is demanded. There are three cases:

- If the closure has already been evaluated its value can be read immediately.
- If the closure has not been evaluated, it is marked busy and then evaluated by the thread which needs it.
- If the closure is already marked busy, some other thread has begun to evaluate it, and the thread cannot proceed until it has been updated by its result. The thread blocks on the closure. When evaluation of the closure has finished, all such blocked threads are released. When a thread blocks, the processor on which it is running schedules another runnable thread, or assigns a new thread to another sparked closure from the work pool.

Two aspects of an implementation of PGR require particular attention: scheduling and shared heap support. Scheduling policies and techniques are discussed in Section 5.2. In this paper we concentrate on supporting the shared heap required by PGR. It is essential that it is supported with high efficiency in order to achieve speedups.

3 Cache-coherent multiprocessors

The performance of parallel programs operating under the PGR regime depends critically on the provision of access to a shared heap with very high average performance. It is freedom from explicit considerations of data placement which gives the parallel graph reduction model its simplicity, and it must be supported at reasonable cost.

The most straightforward way to implement shared-memory is to construct a high-performance interconnection network to route requests to read and write shared-memory to the appropriate memory bank. This approach has been taken in the BBN Butterfly [14], and the IBM RP^3 prototype [38]. This has the problem that communication is incurred for all accesses to shared memory, whereas a cache-based scheme would allow programs which demonstrate some locality to benefit from much lower memory access latencies, and a much lower network traffic level.

In a uniprocessor, a cache mechanism places commonly-used data in local fast memory rather than in slower, bulk storage such as dynamic RAM. In a multiprocessor this is especially attractive since communication paths to and from bulk storage are likely to suffer contention with activity generated by other processors.

The cache coherency problem arises from the presence of multiple copies of the same datum in different processors' caches. When a write occurs to the datum's address, some mechanism is needed to prevent out-of-date cached copies from being used in the future. Cache coherency protocols consist of policies and distributed data structures to organise this. A survey of design issues in coherent cache multiprocessors is given in [40]. There are many design variables:

1. Replication/migration vs. remote access: where no temporal or spatial locality exists, replication or migration of the datum yields no benefit.

2. Cache line size: the unit of replication/migration is a compromise between bandwidth, the benefit of spatial locality, and "false sharing". False sharing occurs when different data on the same cache line are subjected to the same migration decisions leading to coherency traffic which could be avoided with a different data placement (described in detail in Section 3.2).
3. Invalidation vs. update: when a write occurs, updates could be sent to all remote copies. This is a benefit if the remote copies are being actively read, but invalidating the remote copies could avoid further coherency messages.
4. Delayed ("weak") consistency: in order to increase parallelism and reduce network traffic, some implementations allow coherency actions to be delayed. With strong consistency, the value read by a program is guaranteed to be the result of the most recent write to that address. If a process is allowed to proceed after a write before all invalidations or updates have been acted upon, the semantics of the shared store may be compromised.
5. Directory and ownership representation: with migratory ownership, complex distributed data structures are involved.

In this paper we concentrate on the first four issues outlined above. Results presented in Section 6 are based on simulations of three forms of shared-memory: an ideal shared-memory in which all accesses take some small constant time, a conventional invalidation shared-memory similar to schemes used in commercial multiprocessors, and a scheme based on a new optimised protocol. The invalidation protocol is described in the next section, and the new optimised protocol in Section 4.1.

3.1 An invalidation protocol

The protocol we simulate is essentially similar to the Berkeley Ownership Illinois protocols [4], which have been used in commercial snooping-bus shared-memory machines such as the Sequent Symmetry [34]. We describe the protocol as it is used in a bus-based multiprocessor, consisting of a set of processing elements (PEs), each comprising a processor and a cache interconnected by a simple bus. Further details of our simulation and our simplifying assumptions can be found in Section 5.3.

The shared-memory region is divided into a number of cache lines of some constant size. Each line is owned by a particular PE; ownership changes dynamically according to coherency transactions. When a PE attempts to read a line which is not in its local cache, it sends a read request to the owner of the line (by broadcasting on the bus) which responds with a copy of the line. The requesting PE adds it to its cache and proceeds to use it. In this way, multiple copies of lines come into existence in the system. A write to a line that is cached locally but not remotely can take place without using the network. The difficulty occurs when a write occurs to a line that exists in multiple caches: before the write is allowed to proceed the remote copies must be invalidated to prevent them being used again. The requesting PE becomes the new owner of the line when the write has completed.

Bus-based systems provide support for invalidating remote copies of lines using broadcast: all PEs monitor ("snoop") the bus and respond to transactions as necessary. Such systems have the advantage that the identities of the PEs which have copies of each line (the "copy set") do not need to be maintained. The bus quickly becomes a bottleneck as more PEs are added to the system since it cannot provide adequate bandwidth.

The same protocol can be used on more general networks which do not provide such strong support for broadcasting, provided that:

- Directories are used to record the copy set of each line.
- A mechanism to locate the owner of lines is adopted.

Directory schemes are described and evaluated in [2]. See [21] and [33] for discussions of mechanisms used to locate the owner of lines.

3.2 Cache line size, locality and false sharing

The size of cache lines is a critical issue in cache-based shared-memory implementations. Small line sizes, of the order of a few tens of bytes, were usual in first-generation bus systems such as the Sequent Balance [41]. Small lines had the advantage that few bus cycles were required to transfer a copy of a line, allowing as many as 30 processors to be used without contention for the bus becoming a limiting factor. This is possible because bus arbitration is so fast. Larger line sizes would have either required more bus cycles (resulting in increased latency of memory operations and greater network contention) or a wider bus (which is more expensive). The situation with modern multiprocessors with richer interconnection networks is quite different: contention is less of a problem since many separate links exist, and message latencies are dominated by the startup time which is independent of the message size. Such architectures therefore favour the use of large messages and cache line sizes.

Large lines also improve the price/performance ratio of the cache since they require less space to store the tag bits. Thus less expensive static RAM is required for a cache of a fixed size.

For example, on the Meiko CS-2 the time required to exchange messages of various sizes are [6]:

- 1 byte: 10μs
- 500 bytes: 21μs
- 2000 bytes: 50μs

The latency thus consists of a fixed startup time and a data transfer time which is a linear function of the message size.

Line size can significantly affect the performance of a program. Large lines allow *spatial locality* to be exploited: fetching a line causes all objects located on it to be copied into the local cache, eliminating further requests if those objects are accessed whilst the copy of the line remains in the local cache. Unfortunately, large lines also increase the *false sharing* problem. This is the sharing of cache lines without sharing of data [17]. The root of the problem is that a line may contain several objects which are used independently. For example, consider the case where processor A creates four objects in shared-memory. Processors B, C, D and E each address one object, and read and write it several times over the same period of time. If each object is allocated on a different line the first write made by each of processors B, C, D and E will invalidate the remote copy held by processor A, allowing subsequent reads and writes to be served by local caches. However, if all the objects are allocated on the same cache line, the coherency overhead will be significantly greater since each write will cause all other copies of the line to be invalidated, and subsequent reads will cause multiple copies of the line to created again, which will need to be invalidated by the next write.

We refer to this effect, in which unnecessary message exchanges take place, as *line stealing*. The net result is that considerable extra network traffic and serialisation occurs which is not required by the data sharing behaviour of the program. The average latency of shared-memory accesses is therefore greater than necessary, resulting in extended execution times.

3.3 Relaxing consistency

False sharing has been identified as a major cause of unnecessary network transactions in studies of imperative benchmarks, *e.g.* [20]. The conventional approach to reducing (but not eliminating) the impact of the false sharing problem is to relax consistency in a controlled way [42].

Lamport formalised the term sequential consistency in which it is guaranteed that a read by any processor of any location will return the last value written to that location [29]. Because of the unnecessary invalidation and serialisation this incurs, protocols which offer weaker consistency properties have been proposed (*e.g.* [17]).

These advanced protocols are based on the concept of weak ordering [1], and offer performance improvements by allowing invalidations generated by a thread while inside a critical section to be processed in parallel with computation. This increases concurrency by allowing the processor to continue executing the instructions following the write without waiting for the invalidation to complete. The number of unnecessary invalidations caused by false sharing is therefore reduced, and coherence is enforced at synchronisation points only. Such schemes have been shown to be effective on some imperative benchmark programs [42].

Closures are small objects (about 20 bytes each in our implementation), a small fraction of the size of cache lines favoured by modern multiprocessors, and therefore false sharing could be a significant problem. In Section 6 we use simulation to quantify the line stealing effect caused by false sharing. In the next section we consider whether weak ordering can be used with PGR, and how significant the performance improvement could be.

4 The memory reference characteristics of PGR

The performance of a multicache shared-memory depends heavily on the pattern of memory references made by a program. We have described above how large cache lines allow spatial locality to be exploited. Sequential imperative programs exhibit significant locality, *e.g.* temporal locality of code accesses in loops, spatial and temporal locality of stack accesses, and spatial locality of array accesses. This is also present in parallel imperative programs, although it is inevitably application dependent. Previous work has indicated that locality is also present in sequential functional programs: several simulation studies have shown the advantage of using large cache lines with graph reduction [28, 31].

We present simulation results which quantify the opposing effects of locality and false sharing in Section 6. In the remainder of this section we discuss the synchronisation scheme of PGR and determine how consistency can be relaxed in order to reduce line stealing and thereby increase the performance of the shared-memory.

Objects in the shared-memory region are allocated, read, updated and shared quite differently in PGR than in typical imperative programs. For example, the following are inherent in the PGR model:

- There is a high turnover of closures, *i.e.* many closures are not accessed again soon after being created because they were only required for intermediate results.
- Each closure is logically updated at most once (by its result). However, an implementation of PGR also requires a closure to be updated when a thread has gained the right to evaluate it (see below).
- All writes to a shared object (updating it with its result and gaining the right to evaluate it) occur in critical sections.

- There is a high rate of synchronisation (far greater than more widely-studied program types such as the SPLASH suite [20]).

These characteristics are a direct result of the synchronisation scheme used in PGR. During a closure's lifetime, it may exist in three different states:

INACTIVE: The closure has not been evaluated, and no thread has yet gained the right to evaluate it.

ACTIVE: The closure has not been evaluated, but a thread has gained the right to evaluate it and will update it with its result at some time in the future.

EVALUATED: The closure has been evaluated and will not be updated again.

Some closures are created in the EVALUATED state, but others, created INACTIVE, will become ACTIVE when they are first demanded, and then EVALUATED when finished. The state transitions from INACTIVE to ACTIVE, and from ACTIVE to EVALUATED both require mutual exclusion: the former to prevent more than one thread from evaluating a closure, the latter to eliminate a potential race condition when a thread blocks on a closure which is in the process of being updated with its result by another thread. A state field and a spin lock are added to each closure in order to achieve this when using a conventional shared-memory. Each state transition requires write accesses to both the lock and the state field.

Unfortunately weak consistency protocols have very little potential benefit due to the high rate of synchronisation exhibited by PGR. For example, consider a thread attempting to acquire the right to evaluate a closure on a cache line which is owned by another PE. This requires a read, a lock acquire, a write, and a lock release. In a delayed consistency protocol (*e.g.* [17]) the first read would obtain a copy of the cache line. The lock acquire is implemented by a write operation which therefore takes place on a line which is present in more than one cache, requiring an invalidation to be issued. The weak ordering model allows this invalidation to be sent and performed in parallel with the instructions following the write. However, the subsequent write is immediately followed by a lock release operation which will block until any pending invalidations have been performed. Little will be gained by not blocking when the lock acquire takes place. The key problem is that the number of memory references between the lock acquire and release is too small to allow invalidations to take place in parallel with computation.

The case when a closure is updated is similar: the closure is first locked, the update takes place (a single write) and then the closure is unlocked. Again the write burst is very short.

Although a number of simulation studies of imperative programs have shown that significant improvements in performance can be achieved using delayed consistency protocols (*e.g.* [42]), the potential improvement is negligible in this case due to the high rate of synchronisation inherent in the PGR model.

4.1 The two-level ownership protocol

Although the high rate of synchronisation inherent in PGR results in only a small potential benefit from using delayed consistency, the synchronisation scheme can still be used as the basis of an optimised protocol.

The optimised protocol is called two-level ownership because the ownership of cache lines is separated from the ownership of the closures stored on them. In fact line ownership is static: the PE which allocates closures on a particular line is designated the owner of that line, and remains so for the entire execution of the program. It is assumed that other PEs can identify the owner of a line quickly (*i.e.* it is either implicit in the address, or it can be found in a small local data structure). The copy of a line held by its owner is the "master" copy which is always coherent. Copies of lines are made whenever a PE accesses the master copy. Note that all copies of a line other than the master are neither updated nor invalidated when subsequent writes occur, as explained below.

A new memory instruction, "acquire", is used to drive the state transitions of closures, and the PE which gains the right to evaluate a closure becomes its owner. The action that results from issuing acquire on the state field of a closure in each of the three possible states are:

INACTIVE: The field of the master copy of the closure is atomically set to ACTIVE, and the old value returned.

ACTIVE: The value of the field of the master copy is returned.

EVALUATED: The value of the field is returned (from any copy of the line).

A "write-through" instruction is used to update an ACTIVE closure by its result. The master copy of a line must be accessed whenever a state transition of a closure may occur, *i.e.* if a PE initiates a transition on a closure it did not create, a network transaction is required, and the requesting PE receives a new copy of the line. However, reading the value of an EVALUATED closure can still be satisfied by a local copy of the line if the closure is in the EVALUATED state. This is the only situation in which an operation can be satisfied by a PE other than that which owns the line.

Incoherent copies of lines come into existence when a line which resides in more than one cache is updated: only the master copy is updated. This can be done safely since any access to an incoherent copy will result in a network transaction with the owner of the line (whose copy is always fully coherent) if the closure is not in the EVALUATED state. Evaluated closures can be read from any copy of a line since they are never updated.

How can spatial and temporal locality be exploited in such a scheme? Accesses made by a PE to closures it created can be served by the local cache. Since EVALUATED closures can be read from cached copies of lines, spatial locality of reads can be exploited by using large lines. Note that spatial and temporal locality of writes to remote lines cannot be exploited. False sharing still occurs, but line stealing is eliminated entirely since invalidation is not used.

There is a complication relating to reading from cached copies of lines. A copy of a line may have been taken by a PE before the closure being accessed was created, *i.e.* the local copy of the line does not contain a copy of the closure at all. This is resolved by associating a counter with each cache line which records how much of the line has been allocated. Since allocation is incremental, a single counter per cache line will suffice. As the owner of the line allocates new closures on it, the counter is incremented to reflect the extent of the line that has been used so far. When a copy of a line is taken the value of the counter is also copied. A read to a cached copy of a line which exceeds the counter value will automatically initiate a network transaction to obtain a new copy of the line.

In summary, the two-level ownership protocol eliminates invalidation (and thereby line stealing) which can cause unnecessary coherency traffic and serialisation. The protocol is unusual in that, although replication is used, it is restricted to data which will not be updated again. In this respect it is similar to subblock placement: only certain parts (*i.e.* subblocks) of a line are useable [25]. However, its performance relative to a conventional invalidation protocol is inevitably program dependent. In the next section we describe our simulation environment and benchmark programs, and in Section 6 we analyse simulation results in order to determine which protocol offers the best performance, and under what conditions.

5 Experimental design

The performance and behaviour of a set of benchmark programs executing under PGR with three types of shared-memory (ideal, invalidation, and two-level ownership) was studied using a series of simulation experiments. A comprehensive description of the experimental design can be found in [7]. Here the most important aspects are summarised.

We have chosen to use simulation rather than an implementation on real hardware since it offers a number of distinct advantages: it allows the behaviour of the system to be closely monitored without affecting its behaviour, permitting, for example, counts of important events to be made without changing the schedule of the computation or its simulated execution time. In addition it allows important design parameters of the parallel machine to be changed easily which is not possible on real hardware, *e.g.* the latency and bandwidth of the interconnection network. Finally, simulation provides a deterministic environment in which unusual or erroneous behaviour can be repeated simply be repeating the run. Validation of the simulation, in the form of comparisons between simulated performance and actual performance on a Sequent Balance multiprocessor, can be found in [7].

Our complete implementation consists of an optimising compiler for a functional language that generates an executable when linked with a parallel run-time system, which contains procedures to build and enter closures, the code for primitive functions, etc. The source language is a lazy, higher-order functional language in the tradition of SASL and Haskell [26]. The primary objective in building an optimising compiler for a lazy functional language is to reduce the frequency at which claims and references are made to the heap. It is therefore of great importance that the compiler used in our experiments should perform well. We have adopted the compiler developed for the FAST project [13], and although comparing compilers is difficult, we are confident that the system is competitive with the state of the art [24]. It also, conveniently, generates C, making generated code very easy to instrument and modify.

Each processor allocates closures from its own independent part of the heap, and therefore communication is never required when closures are created.

Note that a number of assumptions and simplifications have been made in our experimental design: it is a compromise between the need to model the important effects, and the need to study these effects in isolation. Our objective is to learn general lessons about a large class of systems, and we are therefore less concerned that the experiments predict the actual performance of some production system, as to do so we would have to introduce many factors that are orthogonal to the issues we intend to study. These simplifications are outlined in Sections 5.2 and 5.3.

5.1 Simulation technique

The most obvious way to simulate an application program on some target architecture is by a full emulation of the program, one instruction at a time. A scheme of this type has the advantage of being highly reliable, *i.e.* the results of the simulation can be expected to correspond closely to an implementation of the architecture having the same specification as that being modelled. The chief drawback is the huge computational resources required for the simulation.

An address trace is commonly used to evaluate memory hierarchy designs. The trace-driven simulation technique extends naturally to parallel systems, but its validity is questionable for a number of reasons. The root of the problem is that trace-driven multiprocessor simulation generally cannot represent interacting processes correctly. The sequence of instruction and data references made by a program on a uniprocessor architecture is independent of that architecture, but this is not the case for a multiprocessor since different architectural designs can change the relative timing of competing requests for a resource. This can be resolved in statically-scheduled application programs (see [16]), but not in dynamically-scheduled systems such as parallel graph reduction, where the relative timing of events determines the allocation of work to CPUs.

To avoid the validity issues of trace-driven simulation, we have adopted execution-driven simulation, which does not suffer from these problems since it accurately models process interactions. The execution of the application program is interleaved with the simulation of the target architecture. Each processor is represented by a process, and sequences of application instructions are directly executed on the simulation host until a "global event" is generated. A global event is a process interaction (*i.e.* an action that can alter the execution of another simulated processor), such as accesses to the shared heap and scheduling operations (such as sparking a closure, and unblocking a thread). Between consecutive global events generated by a processor, the actions of that processor cannot affect the others. These local events are only important in that they affect the timing of global events. Unlike trace-driven simulation, the execution-driven approach requires reexecution of the program for each set of simulation parameters.

5.2 Scheduling

Parallel graph reduction is inherently dynamically scheduled, and mechanisms are required to distribute sparked tasks around the machine. Logically a single shared task pool is used, but this will inevitably become a source of contention in a large-scale system if implemented as a single queue. The usual approach taken in a large system is to equip each processor with a local task queue to which closures it sparks are added and from which new work can be fetched when necessary. Mechanisms are therefore required to distribute tasks around the machine on demand. Policies relating to this form of scheduling are described and evaluated experimentally in [19].

We have adopted such a work stealing scheme, except that we model ideal behaviour when a PE needs to find a sparked task and none is available locally: the oldest sparked task is used. We do not account for the communications traffic involved in locating the oldest sparked task in the machine, only the cost of transfering it to the local PE. Simulating this ideal behaviour is reasonable because local task queues are usually empty only at the start and end of the computation; between these times all PEs are busy and have surpluses of sparked tasks. The ideal scheme is therefore only unrealistic at the start of the computation, which is a relatively small proportion of the execution time of the benchmark programs used.

If a PE cannot find another task anywhere it is removed from the set of processes under simulation and added to a queue of idle processors. It is either resumed later when work becomes available, or remains in this state until evaluation of the functional program has been completed and the simulation system terminates.

5.3 Simulated architecture

A simple model of a shared-memory parallel architecture has been adopted: the system consists of a set of processing elements (each comprising a processor and a large cache) interconnected by a network. The processor model is based on a simple 32-bit RISC (*i.e.* load/store) device. It is assumed that stack, private data and code regions of each process are served by separate perfect cache systems; each read or write to these areas has a latency of one cycle.

Between consecutive global events produced by an application process, an amount of time is spent accessing local memory, performing primitive arithmetic operations and in other local operations: the clock associated with each application process must be altered to account for this time in order to maintain a correct ordering of global events. Compiler generated code and the runtime system are annotated with cycle counting code, which assigns a cost to each local event instruction.

In order to isolate the essential communications incurred by the applications and the cache-coherency protocol, we simulate caches large enough to hold all the data so that no replacement or capacity-related communications occur. This simplification has been made since we wish to concentrate on the performance of the protocols themselves, *i.e.* the network traffic generated when maintaining the coherence of the heap. For similar reasons we simulate a contention free interconnection network.

5.4 Benchmark functional programs

The suite of programs we have been using are as follows:

nfib	compute the n^{th} Fibonacci number
nqueens	compute a safe arrangement of queens on an $n \times n$ chess board
quad	find the integral of a cubic function using adaptive quadrature
matmult	multiply two $n \times n$ matrices
wave	simulation of tidal flow in an estuary
db	a "pipelined" database transaction processor

The first four are standard small benchmark programs taken from Goldberg's thesis [19]. Results from these programs can be found in our earlier papers [8, 9]. Of these four, only the results of *matmult* are shown here. It is only included since its behaviour is particularly easy to understand. It is very much a best case program – it is highly parallel, and its inputs are large data structures which are not updated, and clearly significant opportunities exist for exploiting locality. The algorithm is naive: the problem is subdivided at runtime into tasks which each multiply a row by a column. The matrices used are 10 squared.

The estuary simulation, *wave*, divides an estuary into a square matrix of sub-areas and the action of tides is simulated for some number of iterations. The results of each iteration are represented by three matrices which are consumed by the following iteration [44]. The generation of the three matrices can be done concurrently, and requires elements of each input matrix to be read. Three iterations over 5x5 matrices were used.

The transaction processing program is the largest of our programs and is based upon Trinder's pipelined database system [43]. The program uses a database represented by a binary tree. Database operations (*i.e.* lookups and updates) are grouped into transactions which either commit or abort. One hundred transactions, each of ten operations are used, and the tree contains 10000 leaf nodes (*i.e.* database records). It is an interesting program for a number of reasons: its high rate of synchronisation reflects a fundamental requirement of the algorithm, unlike many benchmarks (*e.g.* matrix multiply). It displays spatial locality, yet this is not because it has a statically-predictable access pattern. The operation of the program is described in detail in [10].

We have chosen these programs because they have different data sharing characteristics and therefore exercise the protocols in different ways. They are representative of several classes of functional programs (*e.g.* *wave* is a typical iterative scientific program), but it cannot be assumed that all functional programs will show similar results.

5.5 Garbage collection

When storage allocated from the shared heap becomes free, it should be recycled for reuse. In a parallel system a parallel garbage collector is needed, and the area is the subject of intensive research. The behaviour of the garbage collector may interfere with normal program execution in two ways: firstly, it may change the relative timing of processes, depending on when it is activated, and whether all processors collect concurrently. Secondly, garbage collection may substantially change the pattern in which store is allocated.

An important simplification has been made here: no garbage collection is done at all. Instead, each processor is allocated a large contiguous segment of the shared address space, from which it allocates closures as required.

The motivation for this decision is as follows: to introduce a garbage collector into the simulation would require the selection of one of the available algorithms, and the results would then be applicable only when a similar collector is in use. Unfortunately, there is no obvious candidate: there is no consensus on how garbage collection should be done in large shared-memory systems. However, any copying or compacting collector would finish by handing the application program a contiguous segment of free store. Thus, there is a significant period during which heap closure allocation proceeds in a simple pattern (incremental allocation is used). The assumption is that this is the case all the time. Although we do not simulate garbage collection, it is an important issue which we return to in Section 7.

6 Simulation results

Simulations were made of each program operating with an ideal shared-memory, the invalidation protocol, and the two-level ownership protocol using the following simulation parameters: 1, 2, 4, 8, 16 and 32 PEs, and cache line sizes of 1, 2, 4, 8, 16, 32, 64, 128 and 256 closures. For most of our simulations we assumed a low-latency interconnection network in which shared-memory references which cannot be satisfied by the local cache have a latency of 10 cycles, whereas those that can take 1 cycle. Note that, in order to make results easier to interpret, transaction latencies have been kept constant despite varying the cache line size. The influence of the latency and bandwidth of the network is addressed in Section 6.4.

6.1 Ideal shared-memory

An ideal shared-memory is defined to be one in which all accesses have some small constant latency. Specifically the processor never stalls due to memory accesses. Results from ideal shared-memory are useful for several reasons, *e.g.* they allow scheduling mechanisms and policies to be studied without being affected by the performance of the shared-memory. We use the results here to set performance standards against which results from more realistic shared-memory implementations can be compared.

Relative performance curves (for up to 64 PEs) are shown in Figure 1. Each program shows near linear speedup when only a small number of processors are used. The speedup curves diverge from the ideal curve when more processors are used since most processors are idle at the start and end of the computation, which now forms a larger part of the execution time. The transaction processor *db* shows an asymptotic speedup of 20, which is achieved with 22 PEs; the amount of parallelism present is limited by the transaction structure as described in [10].

These curves indicate that each program scales well with the number of processors, given the amount of parallelism present, *i.e.* the scheduling policies have been effective. Instrumentation of the simulator shows that speedup is non-linear solely due to startup and shutdown effects, demonstrating that the optimal speedup has nearly been achieved. Minor scheduling variations can increase the shutdown period, but the resulting impact on execution time is relatively small.

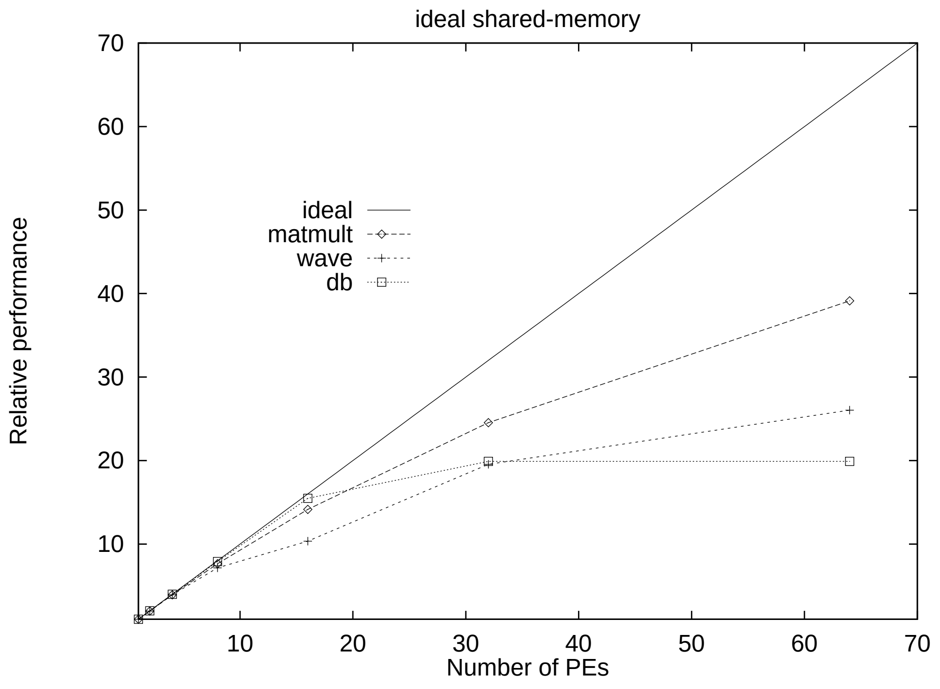

Figure 1: Relative performance of the benchmark programs with ideal shared-memory

This can be seen in Figure 1 for *wave* with 16 processors.

6.2 Network usage in multicache shared-memory

The simulated execution time of a program is determined to a large extent by the latency of network transactions. Consequently, relative performance figures can be confusing. Since we want to compare the performance of the two cache coherency protocols in a way that is independent of network latency, we have adopted the metric of "cache transaction ratio", defined to be the proportion of shared-memory accesses made by a program which require use of the network. Multicache schemes are designed to minimise average memory reference latency by minimising the cache transaction ratio. For example, for the invalidation protocol, a read to a line which is present in the local cache does not require use of the network, but a write to a line which is present in more than one cache does.

The graphs in Figure 2 show cache transaction ratio plotted against cache line size for a variety of processor configurations for each benchmark program. The graphs in the left column were produced from simulations of the invalidation protocol, the right column from the two-level ownership protocol.

6.2.1 Invalidation protocol

The invalidation protocol graphs clearly show the locality and false sharing effects described in Section 3.2. For example, consider *db* with 32 PEs: for the smallest line size, the cache transaction ratio is about 11%. As the line size is increased to 2 closures, the cache transaction ratio is reduced to 8%. This is a direct result of spatial locality: sometimes the extra closure which is copied on a read miss is referred to, and when this occurs it does not need to be copied across the network. Note that the reduction in cache transaction ratio is very significant: more than a quarter of shared-

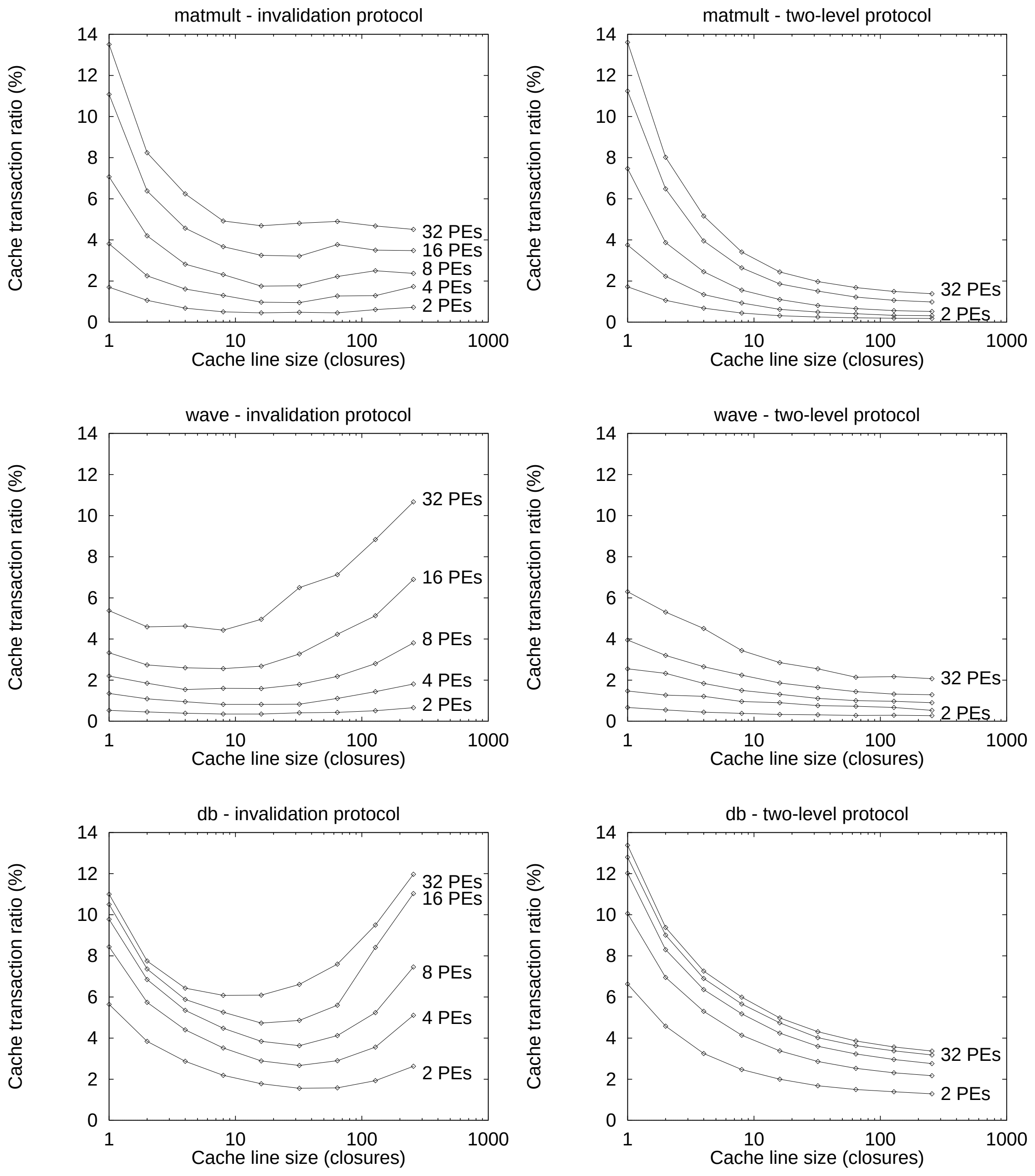

Figure 2: Cache transaction ratio as a function of cache line size for the invalidation protocol (left column) and the two-level ownership protocol (right column)

memory references which required use of the network at the minimum line size can now be satisfied by the local cache. As line size is increased again, a further reduction in cache transaction ratio occurs. However, after the line size exceeds 16 closures the trend reverses: the cache transaction ratio begins to increase. This occurs when the magnitude of the line stealing effect outweighs the gain from locality.

Similar trends are seen for *db* with different numbers of processors. Notice that, as the number of processors is increased from two:

- The cache transaction ratio increases, and
- The line size at which the minimum cache transaction ratio is observed decreases.

The first effect is caused by essentially the same amount of data being shared by more processors: it becomes more likely that a line will be referred to by another PE, and therefore more likely that a write hit will require some remote invalidations, and that subsequent reads will miss in the cache due to an invalidation by another processor.

Since using more processors results in more invalidation and reduced execution times, the *rate* of invalidation increases, *i.e.* lines remain in caches for shorter periods of time and it becomes less likely that the extra closures which are copied on a read miss will be referred to before the line is invalidated. That is, it is harder to exploit spatial locality. Consequently the point at which false sharing begins to outweigh spatial locality occurs at a smaller line size.

Similar trends are seen for *matmult* and *wave*: in each case the interaction between locality and false sharing is seen. However, some differences are apparent: *matmult* can easily benefit from using large cache lines since each thread multiplies a row and a column, and therefore significant reductions in cache transaction ratio are seen as the line size is increased. The line stealing effect is not severe in this case because the input matrices are not updated, and the corresponding cache lines can be copied into each cache and will not be invalidated. Since the line stealing effect is not severe, the cache transaction ratio falls until the cache line size reaches about 64 closures for the invalidation protocol, and after this point only a negligible increase is seen.

It appears to be harder to exploit locality with *wave* since only relatively small reductions in cache transaction ratio are seen. This is due to the small size of the shared data structures used. However, false sharing still occurs: the optimum line size is about 8 closures.

Note that, for a fixed cache size, increasing line size will result in increased conflict misses, thereby reducing the optimum line size. Since we have used infinite caches, we have not measured this effect. Conflicts might affect both caching schemes in similar ways. If conflicts are considerably more frequent with the two-level protocol for a certain cache size, the invalidation protocol may offer better performance.

In summary, we have used cache transaction ratio to measure the effectiveness of the invalidation protocol. Simulation results have shown the interaction between spatial locality and false sharing. Spatial locality is application dependent, but can be significant. Line stealing has been shown to be a major problem with all programs with large lines.

6.2.2 Two-level ownership protocol

Cache transaction ratio graphs for the two-level ownership protocol are shown in the right column of Figure 2. It is immediately clear that increasing line size always reduces cache transaction ratio: line stealing has been entirely eliminated. Improvements due to locality can now take place unhindered. The reduction in cache transaction ratio under the invalidation protocol as the line size is increased is initially significant but line stealing acts to reduce the improvement. The new protocol shows a similar reduction in cache transaction ratio as the line size is increased to about 8 closures, but it continues to fall as the cache line size is increased.

Although the line stealing effect has been eliminated, the cache transaction ratio recorded for the minimum line sizes is often higher than the corresponding value for the invalidation protocol. For example, at the minimum line size, *db* with 32 PEs has a cache transaction ratio of 13.5% with the new protocol, but only 11% with the invalidation protocol. We will see why this occurs in Section 6.3.2.

Although the cache transaction ratio is often small (*e.g.* 5%) the effect it has on simulated execution time is determined by the latencies of the coherency transactions which are many times greater than the latency of a read-hit for the class of architecture we envisage. Therefore even relatively small variations produced by changing line size can have a significant effect on performance.

These results indicate that we have succeeded in exploiting spatial locality to at least the same extent as under the invalidation protocol and have also succeeded in eliminating unnecessary coherency transactions due to line stealing.

6.3 Quantifying spatial locality and line stealing

It is apparent that line size does affect performance, but the contributions of spatial locality and line stealing are not clear. Although line stealing can be eliminated, it is at the expense of requiring that all writes to remotely created objects use the network. The simulator uses monitoring information to enable these effects to be measured separately. For each protocol, a classification of heap references is produced, allowing any gains from spatial locality and losses due to line stealing to be quantified. Monitoring information associated with each closure records the elapsed simulated time at which the closure was last written, the set of PEs that have accessed the closure since that time, and the last PE to write to the closure. Information stored with each line describes the time at which each PE last took a copy of that line, the PE that owns it, etc. This information allows the simulator to classify each heap reference. In the following section we study reads, leaving writes for Section 6.3.2.

6.3.1 Reads

Heap read references for the invalidation protocol are classified as follows:

Simple-read: the PE has accessed an up-to-date copy of the closure before, and has a coherent copy of the associated cache line.

Mandatory-read: the PE has never had an up-to-date copy of the closure in its cache, and communication must take place. Either the PE has not accessed the closure before, or it has been updated by a remote PE since it did. This is necessary communication required by the data sharing characteristics of the program.

Gain-read: the PE has not accessed an up-to-date copy of the closure before, but has a coherent copy of the cache line, *i.e.* a gain from spatial locality.

Loss-read: the PE had an up-to-date copy of the closure in its cache at some time in the past, but the line has been invalidated, *i.e.* a loss from line stealing caused by false sharing.

The classification for the two-level protocol is similar except that the loss-read class is eliminated because invalidation does not take place.

Graphs of read access types for 32 PE simulations with both protocols are shown in Figure 3. On each graph the solid line separates read references not requiring network use (simple-reads and gain-reads below) from those that do (mandatory-reads and loss-reads above).

First, consider *db* with the invalidation protocol:

- When the line size is 1 closure, clearly there is no opportunity to exploit spatial locality, but also false sharing cannot occur. Therefore all reads are either simple-reads or mandatory-reads (only about 15% of references are mandatory-reads).

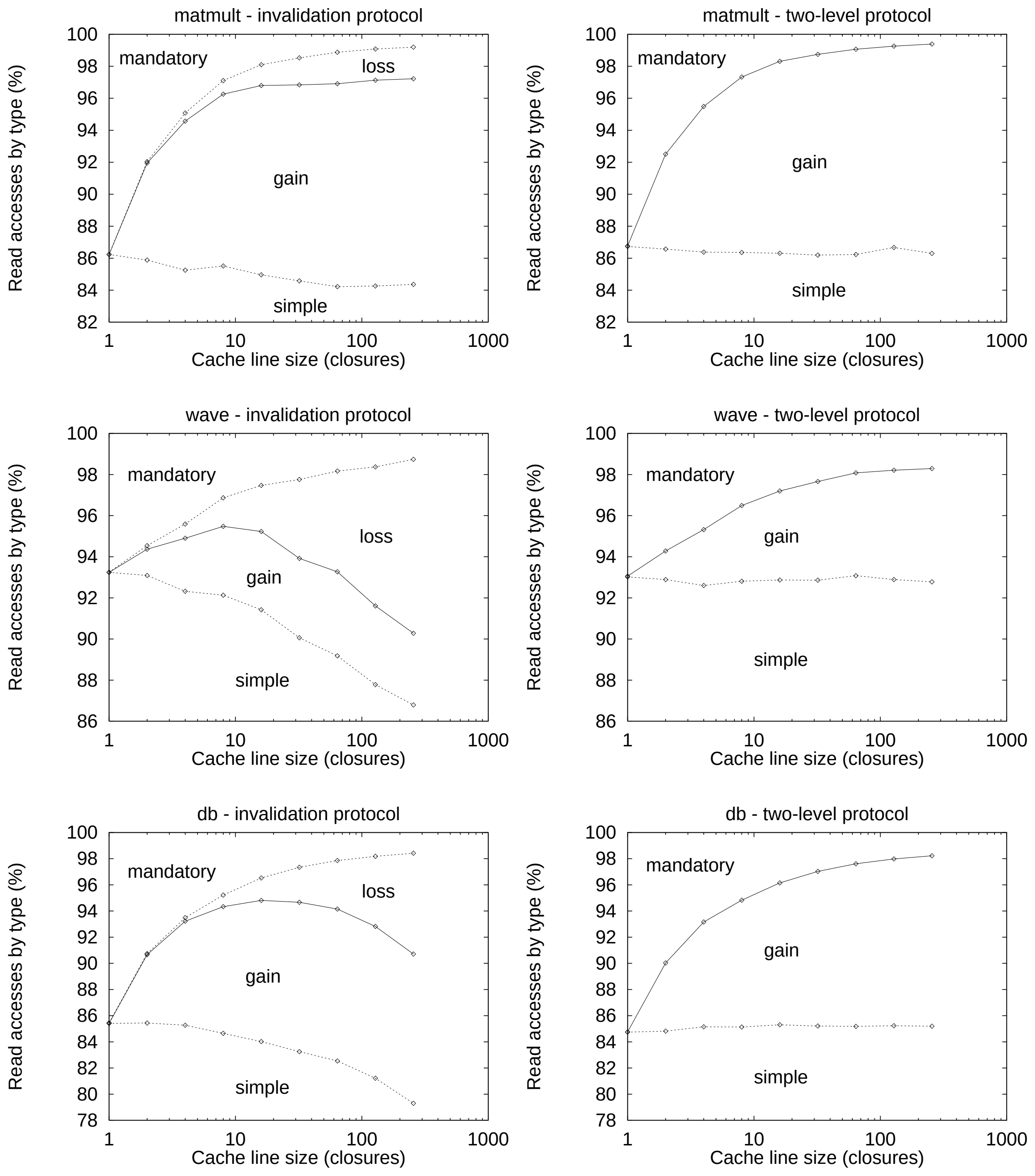

Figure 3: Distribution of reads for the invalidation protocol (left column) and the two-level ownership protocol (right column) with 32 PEs

- As the line size is increased, spatial locality gains are made, and the number of mandatory-read accesses decreases. This trend continues for larger line sizes, ultimately reducing mandatory-reads to about 2%.
- Line stealing occurs even when the line size is 2 closures, and increases steadily for larger line sizes.
- Read references not requiring network use peak at a line size of 32 closures.

The corresponding graph for *db* with the two-level protocol is similar: again 15% of reads are mandatory-reads at the minimum line size, and again, as line size is increased the replacement of mandatory-reads by gain-reads occurs. However this time no losses occur. Note in particular that the mandatory-read regions are almost the same: this means that the loss-reads of the invalidation protocol have all been replaced by cheaper gain-reads and simple-reads. Both *matmult* and *wave* show similar results; only the magnitude of each region is different.

These results show that the two-level ownership protocol can exploit locality of reads to at least the same extent as the invalidation protocol, but without losses due to false sharing.

6.3.2 Writes

The classification scheme for write references for the invalidation protocol concentrates on relating the sharing activity of lines to that of the closures allocated on them. In particular it quantifies both the advantage and disadvantage of invalidation.

Allocation-write: a write-miss caused by allocating a new closure on an unused cache line. It can be serviced simply by allocating a new line and writing to it, *i.e.* no communication is required.

Simple-write: a write-hit which is either an allocation of a closure, or an update of a closure which has not been accessed by another PE. In either case communication is not required.

Mandatory-write: a write-hit or a write-miss, but in either case a coherent copy of this closure has been accessed by another PE, and communication is required.

Gain-write: a write-hit which does not require invalidation, although coherent copies of the closure have been accessed by other PEs.

Loss-write: a write-hit or a write-miss, which is either an allocation, or an update of a closure which has not been accessed by another PE, but communication is required.

The corresponding classification for the two-level protocol is:

Allocation-write: a write-miss caused by allocating a new closure on an as yet unused cache line. Again, a fresh line can be allocated quickly without communication.

Local-write: a write to a cache line owned by the PE, *i.e.* no communication required.

Remote-write: a write to an object owned by a remote PE, thus requiring use of the network.

Graphs of write accesses for 32 PE simulations with both protocols are shown in Figure 4. Again, the solid line separates write references not requiring network use from those that do.

Under the invalidation protocol,

- The vast majority of writes are to newly-allocated memory, and are classified either as "simple" or "allocation" depending on whether a fresh cache line is used.

  When large lines are used, fewer writes refer to completely fresh ("allocation") cache lines. However, large lines lead to an increased risk that a remote read will result in a remote copy being taken. Because of this false sharing, a subsequent write would cause an invalidation, and be classified as a "loss". However, later writes to the line would benefit from this invalidation and be counted as "gain".

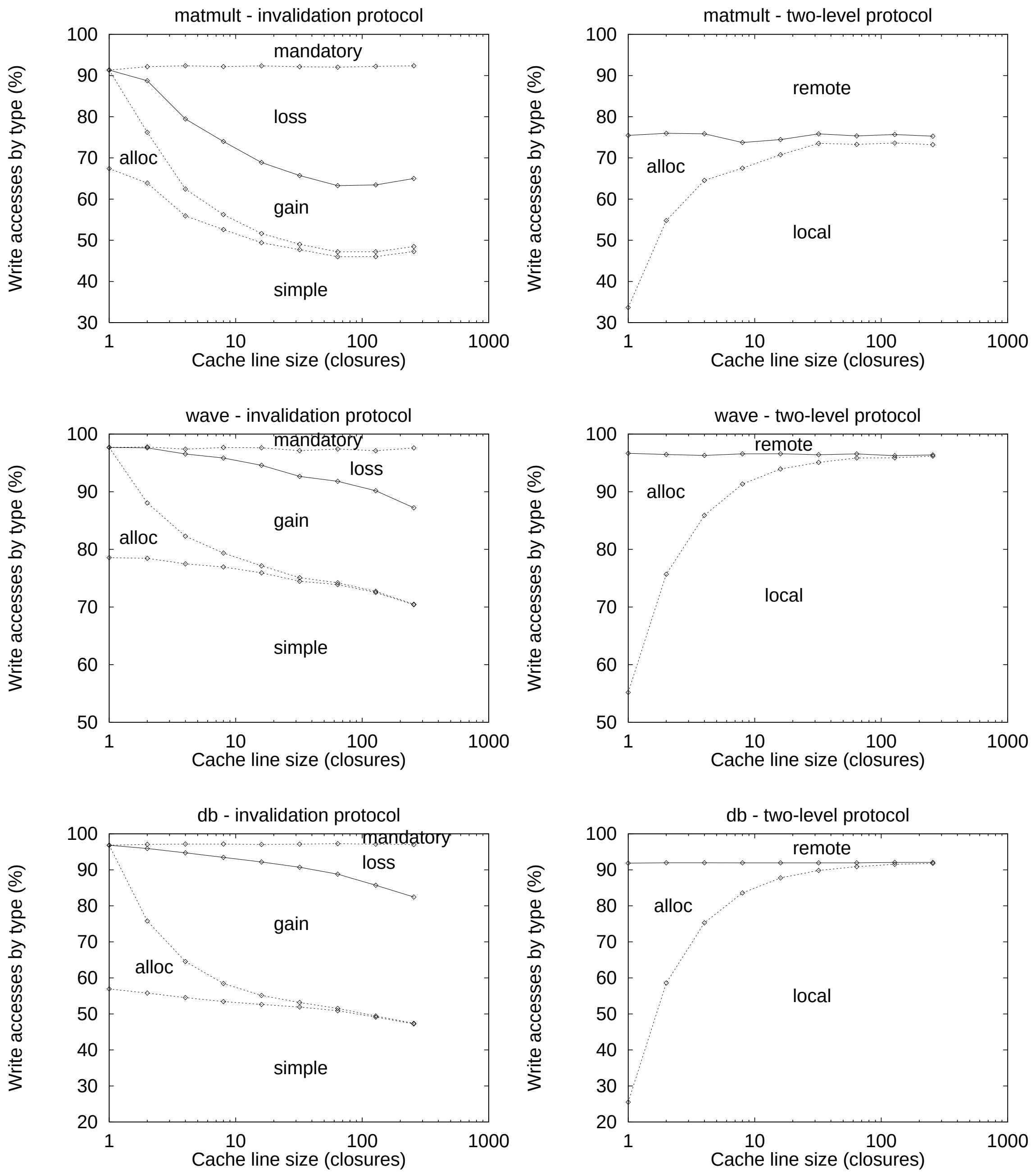

Figure 4: Distribution of writes for the invalidation protocol (left column) and the two-level ownership protocol (right column) with 32 PEs

- Gain writes arise when the first of a spatially-related series of writes performs the invalidation needed to allow several writes to proceed with exclusive access.

  Gain writes increase as cache line size increases, reflecting spatial locality for writes.

  Note that there is evidence of no write gain due to temporal locality; this is why no gains occur at very short cache lines

- Mandatory writes are almost constant, as expected. These are writes to a closure which itself has been referred to by another processor, so, regardless of cache line size, an invalidation will be needed (some small variation may occur due to dynamic scheduling).

- Losses are caused by false sharing, as in the read case. Since the same amount of data is stored in fewer lines it is more likely that an individual line will be copied into more than one cache, and therefore more likely that a write will require communication.

  As indicated by the solid curve, only loss and mandatory writes involve communication. In all three benchmarks it is clear that losses due to false sharing are a major cause of unnecessary communication.

The two-level protocol operates at a higher level than reads and writes so direct comparison is not meaningful. However:

- "Local" and "allocation" writes behave in much the same way as "simple" and "allocation" under the invalidation protocol. Note that the "local" class includes writes to closures of which copies do exist elsewhere (*i.e.* writes by the owner of the line). In the new protocol no broadcast of invalidations or updates is needed.

  For this reason, "local" subsumes some references which would be classified as mandatory or even gains in the invalidation protocol.

- Remote writes involve closures created by another processor. In the new protocol, there is no migration of ownership so these always involve communication in order to keep the creator's copy up to date.

  It is sometimes possible for the invalidation protocol to honour these writes without communication, since the processor issuing the write may already have exclusive access to the cache line (this is the "gain" case).

- The new protocol may get worse write miss rates than the invalidation protocol, but this is compensated for by improved read miss rates. This effect would be largest at large cache line sizes, but invalidations due to false sharing interfere.

In summary, the new protocol is marginally worse at very small cache lines because it lacks migration of ownership, but for medium and large cache lines it achieves a reduction in the number of messages by exploiting spatial locality without unnecessary communications due to spurious invalidations.

6.4 Relative performance

Large cache lines are useful when communication performance is dominated by latency, rather than by bandwidth. In this section we briefly illustrate how bandwidth affects the performance of both protocols by studying a simple interconnection network model in which message transfer time consists of a latency and a function of the message size.

In the results presented so far, we have compared the performance and behaviour of the two protocols by counting the number of network transactions required (the cache transaction ratio). This has the advantage that it allows us to compare the protocols in a general way. The effect that cache transaction ratio has on the execution time and speedup of an application depends on the speed of processors, the latency and bandwidth of the interconnection network, etc. For example, minor variations in cache transaction ratio will have little effect on a system with slow processors and a fast network since execution time will be dominated by CPU time, whereas it would have a major impact when fast processors and a slow network are used. The results presented above are therefore independent of CPU and network speeds.

To illustrate how varying bandwidth might affect performance, we have chosen to simulate a multiprocessor with very fast processors and a high bandwidth network. The processor speed and network bandwidths used are optimistic by today's standards and do not represent a particular machine, but have instead been chosen to clearly illustrate that the advantages gained by using large cache lines with any protocol are reduced by increased network latency.

Figure 5 shows what happens with the *db* benchmark with 8 PEs under the two-level ownership protocol. For simplicity we assume a fully-connected interconnection network with no blocking or contention for cache controllers. The graph shows the relative speedup with increasing bandwidth. Messages which do not carry data take 20 cycles, whereas those that do take 20 cycles, plus 0.3, 0.2, 0.1, 0.02, 0.01 and 0 cycles per closure (*i.e.* about 3, 5, 10, 50, 100 closures per cycle).

The curve for the infinite bandwidth shows the expected behaviour: increasing line size reduces execution time, and thereby increasing speedup. For any given limited bandwidth there is an optimum cache line size. For comparison, we show the same experiment using the invalidation protocol in Figure 6. This shows lower performance, and the benefit of large cache lines is not fully realised.

When the network bandwidth is relatively low, the optimum cache line size is quite small and the relative benefit of the two-level protocol over the invalidation protocol is minimal. The two-level protocol is most valuable when network bandwidth is high compared to the message latency or start-up overhead. These results demonstrate that, although the contention problems inherent in invalidation have been solved, other factors must be taken into consideration when determining the line size to be used.

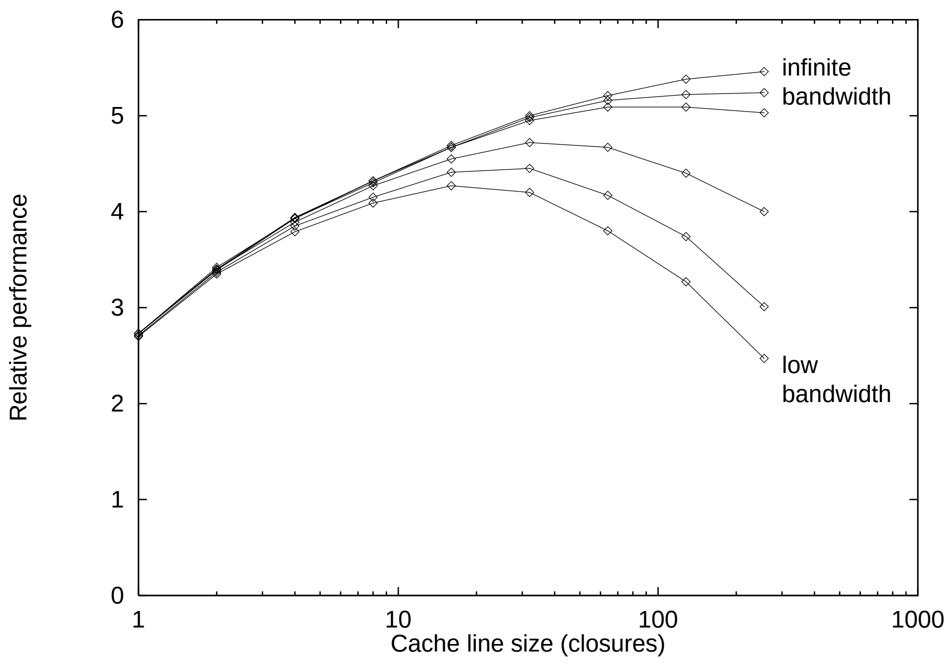

Figure 5: Speedup of *db* as a function of bandwidth, two-level protocol

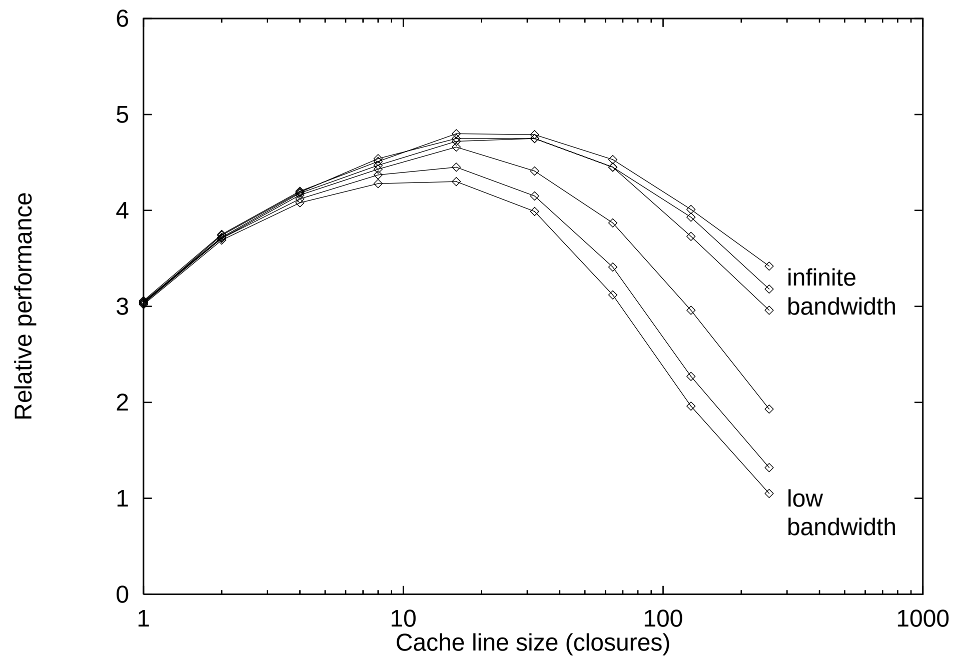

Figure 6: Speedup of *db* as a function of bandwidth, invalidation protocol

7 Discussion and related work

We have shown that a careful implementation of parallel graph reduction can use substantially fewer messages than an implementation based on a conventional shared memory multiprocessor. This is shown by our experimental results for a invalidation cache coherency protocol. A non-cached shared memory machine such as the BBN Butterfly would be worse, since every read would incur communication. It is also easy to see that an update protocol would involve more messages since updates would have to be multicast to every processor holding a copy of each cache line whether or not the value is actually read.

This would be academic if special-purpose hardware were required to support the new protocol efficiently, but this is not so. Communication occurs only when a processor performs an acquire or update operation on a closure, and in both cases a test is already needed by the PGR algorithm. Some support for memory management may prove useful in order to locate a local cached copy of a line; both hardware MMU and hash table schemes can be used effectively. Thus an implementation on a message-passing machine would be fairly natural and we have been investigating various approaches. Interestingly, it is also possible to use a shared-memory machine: each processor would be allocated a separate region of the address space, and copying would be used instead of message passing.

The new protocol works because of the single-assignment property of closures in the PGR graph. Garbage collection is therefore important, and has not been addressed in this study. Garbage collection can be used to improve the cache hit rate by reusing addresses (thereby avoiding replacement of useful data; see, for example, [46]). More awkwardly, garbage collection changes the way data is laid out in memory, and therefore influences spatial locality. Copying collectors are widely used, and these tend to improve spatial locality by copying in a depth-first fashion (see for example [12]).

Arguably, our results form part of an argument in favour of writing programs in a functional, single assignment style for performance reasons. Avoiding mutable objects allows updates and invalidations to be eliminated, so avoiding false sharing. The garbage collection involved would be an unwelcome overhead, but not necessarily large, and it may lead to better locality. The overhead of building and inspecting closures is unfortunate, but it too can be controlled – for example, sophisticated run-time mechanisms can be used to ensure that closures are only built when parallelism is needed, and otherwise efficient sequential, closure-free code can be used (see [39] and [35]).

Finally, we should search for other situations (*i.e.* shared abstract data types) where cache coherency can be optimised.

7.1 Related work

Several authors have studied the cache performance issues with heaps and garbage collection, most interestingly Wilson et al [46] and Appel [3]. Our work concentrates instead on coherency.

The GRIP machine [37] uses a kind of cache mechanism: nodes are built in local memory and a subgraph is flushed to the globally-accessible memory only when another processor might access it. In the Glasgow group's more recent work (see for example, [22]) they propose a more cache-like scheme where, when a node is referenced by a remote processor, a large message is packed with as much of the subgraph of which it is the root as will fit. This is similar in objective to our work, but should use network bandwidth more efficiently at the cost of much higher processing overheads.

Langendoen, Muller and Vree [30] avoided coherency problems in their parallel functional language system by a different means: their "sandwich" parallelism annotation ensures that the sub-tasks need refer only to read-only subgraphs.

A similar approach to ours has been proposed in a dataflow context by Dennis and Gao [15], but they have not evaluated the scheme. The idea is applicable in dataflow-related contexts such as Nikhil's Cid [36] (where related cache-related ideas are discussed) and Blumofe et al's Cilk [11].

7.2 Further work

This work has raised many issues deserving further investigation:

- Improvements in the two-level ownership protocol, such as a more flexible ownership policy to reduce contention for access to the processor that owns a cache line. We have assumed that a memory unit (or cache) can handle accesses from all other processors simultaneously. This is overly optimistic and warrants further study.
- Investigation into the overheads of garbage collection.
- A more complex architectural model, including network and cache controller contention effects, and finite capacity effects.

8 Conclusions

We have investigated a conventional parallel functional language implementation which uses a heap shared between processors for all synchronisation and communication. Parallelism was introduced manually using the "spark" model used in parallel graph reduction. We have executed a range of simple parallel functional programs, and we have studied their memory sharing behaviour using a simulator. After using an ideal memory system to establish the potential performance of each benchmark, we studied an invalidation protocol (as used in most modern shared-memory multiprocessors), and this led us to formulate a novel two-level ownership protocol.

Our main goal was to reduce the number of messages exchanged between processors. We found that this can be done by using large cache lines – when a processor requests a remote graph node, we transfer the entire cache line in which the requested node falls. This works because storage is allocated contiguously and most of our example programs display substantial spatial locality.

Unfortunately, using the invalidation protocol this reduction is limited because processors contend for exclusive ownership of each cache line in order to ensure consistency when writes occur. With large cache lines, this "false sharing" effect happens increasingly often, and our results show that this leads to poor performance.

Our new protocol is specially designed for the nodes of the PGR graph, which are accessed in a disciplined way. This allows large cache lines to be replicated in many processors' caches without the need for invalidations or updates to be sent when a write occurs. Instead, when a processor reads a cell it checks a "valid" flag to decide whether to refer to the line's owner. Each line is owned by the processor which first allocates it.

This protocol avoids the false sharing problem and allows spatial locality of reads to be exploited without limit. Our simulation results illustrate this property. The new protocol is less powerful than the invalidation protocol in that it provides no means for ownership to migrate on demand. Our simulations show that this leads to an increase in write misses but this is insignificant compared with the benefit of avoiding false sharing.

The protocol makes it possible to reduce the number of messages exchanged, but relies on increasing the size of the cache lines exchanged to do so. It is therefore of most benefit with a high-bandwidth interconnection network with relatively high communication latencies or message handling overheads. The benefits are minimal when bandwidth is relatively low, since the optimal cache line size is small and incurs little false sharing.

Our simulation involves simplifications in many areas, some of which might be interesting to investigate in further work. Nonetheless the results are sufficient to support the conclusion that our special-purpose coherence protocol has a substantial advantage over conventional cache-coherence protocols.

Acknowledgements

This work was funded by the U.K. Engineering and Physical Sciences Research Council through a Research Studentship, and projects GR/F 35081 (*FAST*), GR/J 14448 (*Compaqt*) and GR/J 99117 (*CRAMP*). The helpful comments of Ross Paterson, Tony Field, Simon Peyton Jones, Kevin Hammond and the three anonymous reviewers were very much appreciated.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. *17th Annual International Symposium on Computer Architecture, Seattle, May, in Computer Architecture News*, 18(2):2–14, June 1990.

[2] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. *15th Annual International Symposium on Computer Architecture, Honolulu, May, in Computer Architecture News*, 16(2):280–289, May 1988.

[3] Andrew Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. Technical Report CS-TR-450-94, Princeton University, March 1994.

[4] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[5] Lennart Augustsson and Thomas Johnsson. Parallel graph reduction with the $\langle \nu, G \rangle$-machine. In *Fourth International Conference on Functional Programming Languages and Computer Architecture, London, September*, pages 202–213, 1989.

[6] Eric Barton, James Cownie, and Moray McLaren. Message passing on the Meiko CS-2. *Parallel Computing*, 20(4):497–507, April 1994.

[7] Andrew J. Bennett. *Parallel graph reduction for shared-memory architectures.* PhD thesis, Department of Computing, Imperial College, London, July 1993.

[8] Andrew J. Bennett and Paul H. J. Kelly. Locality and false sharing in coherent-cache parallel graph reduction. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE 93 Parallel Architectures and Languages Europe, Munich, June 1993*, volume 694 of *Lecture Notes in Computer Science*, pages 329–340, Berlin, 1993. Springer-Verlag.

[9] Andrew J. Bennett and Paul H. J. Kelly. Eliminating invalidation in coherent-cache parallel graph reduction. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE 94 Parallel Architectures and Languages Europe, Athens, July 1994*, volume 817 of *Lecture Notes in Computer Science*, pages 375–386, Berlin, 1994. Springer-Verlag.

[10] Andrew J. Bennett, Paul H. J. Kelly, and Ross Paterson. Derivation and performance of a pipelined transaction processor. In *IEEE Symposium on Parallel and Distribued Processing, Dallas*, pages 178–185, October 1994.

[11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[12] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[13] Stuart Cox, Shell-Ying Huang, Paul Kelly, Junxian Liu, and Frank Taylor. An implementation of static process networks. In D. Etiemble and J.-C Syre, editors, *PARLE 92 Parallel Architectures and Languages Europe, Paris, June 1992*, volume 605 of *Lecture Notes in Computer Science*, pages 497–512, Berlin, 1992. Springer-Verlag.

[14] W. Crowther. Performance measurements on a 128-node Butterfly parallel processor. In *International Conference on Parallel Processing, August*, pages 531–540, 1985.

[15] Jack B. Dennis and Guang R. Gao. Memory models and cache management for a multithreaded program execution model. Computation Structures Group Memo 362, Laboratory for Computer Science, MIT, 545, Technology Square, Cambridge, Massachussetts 02139, USA, October 1994.

[16] M. Dubois, F. A. Briggs, I. Patil, and M. Balakrishnan. Trace-driven simulations of parallel and distributed algorithms in multiprocessors. In *International Conference on Parallel Processing, August*, pages 909–915, 1986.

[17] Michel Dubois. Delayed consistency. In Michel Dubois and Shreekant S. Thakkar, editors, *Workshop on Scalable Shared Memory Multiprocessors, Seattle, May*, pages 207–218, Boston, 1992. Kluwer Academic Publishers.

[18] Lal George. An abstract machine for parallel graph reduction. In *Fourth International Conference on Functional Programming Languages and Computer Architecture, London, September*, pages 214–229, 1989.

[19] Benjamin F. Goldberg. *Multiprocessor Execution of Functional Programs.* PhD thesis, Yale University, New Haven CT, 1988.

[20] Anoop Gupta and Wolf-Dietrich Weber. Cache invalidation patterns in shared-memory multiprocessors. *IEEE Transactions on Computers*, C-41(7):794–810, July 1992.

[21] Erik Hagersten. *Toward Scalable Cache Only Memory Architectures.* PhD thesis, Swedish Institute of Computer Science, Stockholm, Sweden, 1992.

[22] K. Hammond, J. S. Mattson Jr, A. S Partridge, S.L. Peyton Jones, and P.W. Trinder. GUM: a portable parallel implementation of Haskell. In *Workshop on the Implementation of Functional Languages, Sweden*, 1995.

[23] P. G. Harrison and M. J. Reeve. The parallel graph reduction machine, Alice. In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction: Proceedings of a Workshop, Santa Fe, September 1986*, volume 279 of *Lecture Notes in Computer Science*, pages 181–202, Berlin, 1987. Springer-Verlag.

[24] Pieter H. Hartel and Koen G. Langendoen. Benchmarking implementations of lazy functional languages. In *Proceedings of the Conference on Functional Programming Langauges and Computer Architecture, Copenhagen, June*, 1993.

[25] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach.* Morgan Kaufman, San Mateo, California, 1990.

[26] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler. Report on the programming language Haskell – a non-strict purely functional language, version 1.2. *SIGPLAN Notices*, 27(5):1–162, May 1992.

[27] Kendall Square Research. *KSR1 Principles of Operations*, 1992.

[28] Philip J Koopman, Jr., Peter Lee, and Daniel P. Siewiorek. Cache behaviour of combinator graph reduction. *ACM Transactions on Programming Languages and Systems*, 14(2):265–297, April 1992.

[29] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[30] K. G. Langendoen, H. L. Muller, and W. G. Vree. Memory management for parallel tasks in shared memory. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management*, pages 165–178, St. Malo, France, Sep 1992. Springer-Verlag. LNCS 637.

[31] Koen Langendoen and Dirk-Jan Agterkamp. Cache behaviour of lazy functional programs. Proceedings of the Fourth International Workshop on Parallel Implementations of Functional Languages, Aachen, September 1992, Aachener Informatik Berichte 92-19, RWTH Aachen, Germany, 1992.

[32] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. *17th Annual International Symposium on Computer Architecture, Seattle, May, in Computer Architecture News*, 18(2):148–159, May 1990.

[33] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[34] Tom Lovett and Shreekant Thakkar. The Symmetry multiprocessor system. In *International Conference on Parallel Processing, Pennsylvania, August*, pages 303–310, 1988.

[35] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[36] Rishiyur S. Nikhil. Cid : A parallel, shared-memory C for distributed-memory machines. In *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing, Ithaca NY*. Springer Verlag, August 1994.

[37] Simon L. Peyton Jones, Chris Clack, and Jon Salkild. High-performance parallel graph reduction. In E. Odijk, M. Rem, and J.-C Syre, editors, *PARLE 89 Parallel Architectures and Languages Europe, Eindhoven, June 1989*, volume 365 of *Lecture Notes in Computer Science*, pages 193–206, Berlin, 1989. Springer-Verlag.

[38] G. F. Pfister. An introduction to the IBM Research Parallel Processor Prototype (RP3). In J. Dongarra, editor, *Experimental Parallel Computing Architectures*, pages 123–140. North Holland, 1987.

[39] David Rushall. *Task Exposure in the Parallel Implementation of Functional Programming Languages*. PhD thesis, University of Manchester, 1995.

[40] Per Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[41] Shreekant Thakkar, Paul Gifford, and Gary Fielland. The Balance multiprocessor system. *IEEE Micro*, 8(1):57–69, February 1988.

[42] Josep Torrellas and John Hennessy. Estimating the performance advantages of relaxing consistency in a shared-memory multiprocessor. In *International Conference on Parallel Processing, Pennsylvania State University, August*, pages 26–34, 1990.

[43] Phil Trinder. *A Functional Database*. PhD thesis, Computing Laboratory, Oxford University, Oxford, U.K., 1989.

[44] Willem Vree. *Design Considerations for a Parallel Reduction Machine*. PhD thesis, University of Amsterdam, 1989.

[45] C. P. Wadsworth. *Semantics and Pragmatics of the lambda calculus*. PhD thesis, Oxford University, Oxford, U.K., 1971.

[46] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching consideration for generational garbage collection. Technical Report UIC-EECS-90-5, University of Illinois at Chicago EECS Dept., Chicago, Illinois, December 1990.