

Automatic Scheduling for Cache Only Memory Architectures *

Ronald Moore

Bernd Klauer

Klaus Waldschmidt

J. W. Goethe-University
Technische Informatik
Frankfurt, Germany

Email: {moore, klauer, waldsch}@ti.informatik.uni-frankfurt.de

Published in the *Proceedings of the
Third International Conference on Massively Parallel Computing Systems (MPCS '98)*
Colorado Springs, CO, April 1998

Abstract

For parallel and distributed systems to gain wider acceptance than they have to date, they must become significantly easier to program. Fundamentally, parallel programming is more difficult than sequential programming as long as data and computation must be distributed by the programmer. Cache Only Memory Architectures (COMAs) provide a Distributed Shared Memory (DSM) where data distribution is performed automatically and transparently. This paper generalizes this idea to achieve the same distribution for computation, thus arriving at an automatic and transparent form of scheduling. Once computation distribution (scheduling) is as automatic and transparent as data distribution, parallel computers become approximately as easy to program as sequential computers.

Where COMA literature normally makes no assumptions concerning the parallel programs which use the DSM, we use special compiler techniques originally developed for multithreaded and dataflow architectures. Having done so, we can specify ways of significantly simplifying the basic COMA coherency protocols, while at the same time enabling automatic, transparent, adaptive run-time scheduling.

*This work was supported in part with funds of the Deutsche Forschungsgemeinschaft under reference number WA 357/11-2 within the priority program "System and Circuit Technology for Massively Parallel Computation".

1. Introduction

For parallel and distributed systems to gain more acceptance than they have to date, they will need to be scalable, affordable — but most importantly, they must be made nearly as easy to program as sequential systems. Ideally, we would like to be able to take programs written in conventional languages (including existing programs, a.k.a. "dirty decks", "legacy software") and recompile them for parallel architectures, thus freeing the programmer from all additional effort above and beyond that necessary to program a conventional computer. This in turn implies that either the compiler, the hardware, or both, must address the fundamental issue of *distribution*. This problem is two-fold: Both *data* and *computation* must somehow be distributed.

The problems of data distribution and computation distribution have traditionally been addressed separately. On the one hand, for data, Cache Only Memory Architectures (COMAs) provide an automatic and transparent form of self-distributing Distributed Shared Memory (DSM) [10]. In this model, each processor has a local memory which has been augmented so that it acts like a giant, slow cache. A memory augmented in this fashion is called an "attraction memory"; Data migrates from one attraction memory to another based on demand (attractive forces) and congestion (dissipative forces). However, the COMA literature unanimously treats *computation* distribution as the responsibility of the application program.

On the other hand, the *dataflow paradigm* [6], [3] and later, the multithreaded architectures (e.g. [15], [5], or [7]), provide us with an attractive way to attack the scheduling problem. Each program is represented as a graph whose vertices represent *threads*, and the time at which a thread is

executed is determined by the availability of its arguments. Threads can be distributed in a number of ways. Nonetheless, the multithreading paradigm has little to say concerning data distribution.

This paper attempts to bring data distribution concepts from Cache Only Memory Architectures together with scheduling concepts from Multithreaded Architectures, in order to arrive at one unified, simplified, cohesive abstract model of computation. The fusion of data and computation distribution is the central principle guiding the development of a new architecture being developed by the authors, named SDAARC (Self-Distributing Associative ARChitecture, pronounced so as to rhyme with “stark”). The SDAARC proposal also includes a novel network structure, which will not be discussed in this paper. The network structures are discussed in [13]. The architecture described here is completely general. We expect it to be attractive wherever the performance of parallel hardware is needed, but where the difficulty of writing parallel software is currently a deterrent to using parallel hardware.

The rest of this paper is organized as follows: Section 2 reviews the foundations for our proposal. Section 3 discusses how the COMA concept can be used to implement automatic scheduling. A comparison to related research is presented in section 4, and concluding remarks are presented in section 5.

2. Foundations

In order to establish the terminology for our new proposal, this section reviews two architectures: COMAs (section 2.1), and Multithreaded architectures (section 2.2). Roughly speaking, COMAs provide the techniques for data distribution, and the multithreaded architectures provide the techniques for automatic scheduling.

2.1. COMA: Cache Only Memory Architectures

Caches were first introduced into distributed shared memories for reasons of expediency, since they provide a cost effective way to reduce memory latency. With the introduction of caches, the memory begins to be able to automatically distribute data. The idea of a COMA builds upon the automatic data distribution capability of caches. Abstractly, we can isolate three main components in a conventional cache: a *memory*, a *cache controller*, which executes the *cache coherency protocol*, and an associative *directory* which determines whether a given data segment (cache line) is in the cache’s own memory, or elsewhere. The first component, the memory, is conventionally small, fast static RAM. However, if we isolate the last two components, we can use them to augment (large, slow) local memories. Memories augmented in this way are called *attraction*

memories to distinguish them from conventional caches.

While the data distribution in non-COMA DSMs is the responsibility of the programmer and/or the compiler, a Distributed Shared Memory built entirely out of attraction memories automatically and transparently performs data distribution.

Cache Only Memories are used in the Swedish Institute of Computer Science Data Diffusion Machine (SICS DDM)[10], the (meanwhile defunct) Kendall Square Research KSR1 [1], and the Simple COMA [16].

2.2. Multithreaded Architectures

Multithreading architectures hide memory and communication latencies by switching contexts. Multithreading provides the basis for a simple, consistent way of encapsulating multiple concurrent computations. While the literature includes a wide range of sometimes contradictory definitions, the following tenants can be taken to characterize multithreading for our purposes in this paper:

- The program is partitioned into threads, and the threads are partitioned into *microthreads* (in the sense used in [15]). Each microthread is a sequence of instructions which neither waits nor loops, but instead terminates within a constant maximum time. Thus, each iteration of a loop, and each function call, represents at least one microthread. If a thread has to wait for some resource (e.g. for input or for a memory load), then one microthread finishes by issuing the request, and another microthread picks up once the requested resource is available. A microthread is considered *ready* if and only if all of its operands are present in its framelet.
- Each thread has its own execution context, including its own execution stack *frames*. Since threads are concurrent, the frames are organized in *cactus stacks* [7] (which are actually not stacks, but trees). While more than one microthread *can* generally share one frame, recent findings indicate that providing one frame for each microthread optimizes performance [2]. In this case, we speak of *framelets*.
- Computation in multithreaded multiprocessor architectures is (often) driven by sending and receiving *active messages* [18]. Sending an active message to a microthread supplies that thread with one or more of its operands. Every active message contains sufficient information for the receiving processor to find the instructions needed to process the data in the message (minimally, an instruction pointer and a frame pointer). Note that the sending and receiving processor can be one and the same.

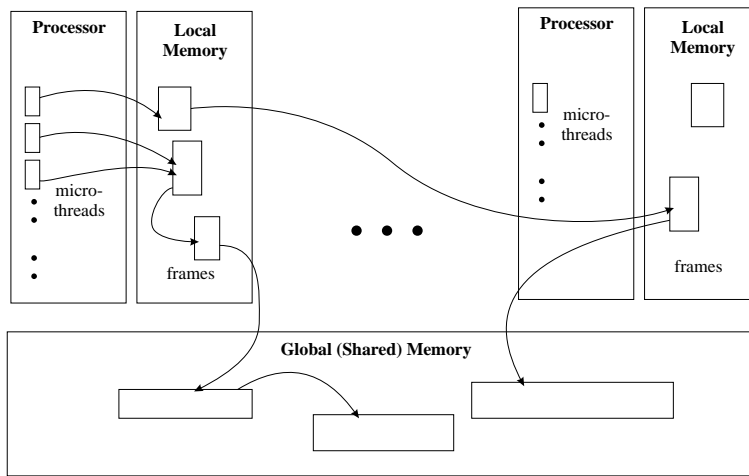


Figure 1. Conventional Multithreaded Multiprocessor Scheduling (compare [15], fig. 2)

The interaction of these concepts and terms is illustrated in figure 1.

Active messages do not, in and of themselves, solve the scheduling problem. We still need a way to distribute the frames and microthreads across the processors. Microthread distribution is sometimes done explicitly, by the programmer or the compiler (e.g. [15]). In other systems, such as [7], a processor with no work polls its neighbors. This scheduling is *demand driven*. Whenever no processor is idle, no distribution takes place. Further, an idle processor must wait for at least two messages to be transmitted (one looking for work, one to receive work) before restarting.

3. COMA-based Automatic Scheduling

A COMA which distributes data can also be used to distribute computation, that is, to schedule: First, we partition the program into microthreads. Every microthread can then be thought of as being intimately linked to one data object: the framelet which stores its execution context. We can then optimize a COMA to automatically distribute the frames. Subsequently, each microthread is executed on the processor where its framelet is resident. Microthreads are farmed out whenever a framelet is moved out of one attraction memory into another. Thus, automatic frame distribution implies automatic computation distribution.

Since frames require special handling, we stipulate that each attraction memory be partitioned into three logically separate virtual caches: one for *instructions*, one for non-frame *data*, and one for *frames*.

Abstractly, there exists one virtual shared pool of framelets. Each attraction memory stores a subset of this pool. Each processor executes a tight inner loop, in which it executes the ready microthreads corresponding to the

framelets in its attraction memory. This concept is illustrated in figure 2.

The remainder of section 3 is organized as follows: Section 3.1 specifies preliminary assumptions and definitions. In section 3.2, we specify how simple programs, without random access data structures, can be automatically distributed amongst the processors. In section 3.3, we extend this concept to incorporate random access data structures. Section 3.4 describes object migration.

3.1. Preliminaries

This section presents basic assumptions and definitions concerning program and data representation, message format, the underlying parallel hardware, and the object states over which the cache coherency protocol is defined.

Program and Data Representation All three partitions of the DSM (instruction, data and frames) are taken to consist of sets of *objects*. These objects can either be of fixed size (like traditional cache lines), or of variable length.

Each object has a unique *object identifier*, which represents its global or virtual address (but not its physical address). One extra object identifier is reserved for each attraction memory site, so that messages can be sent directly to a given site (e.g., in order to create an object which does not yet exist anywhere).

We make no assumptions concerning the source language used to program the architecture, except that it should be possible to translate source code into dataflow graphs. Each subprogram is represented by one dataflow graph. These graphs are then partitioned hierarchically, so that the leaves of the partition-tree represent microthreads (as defined in section 2.2).

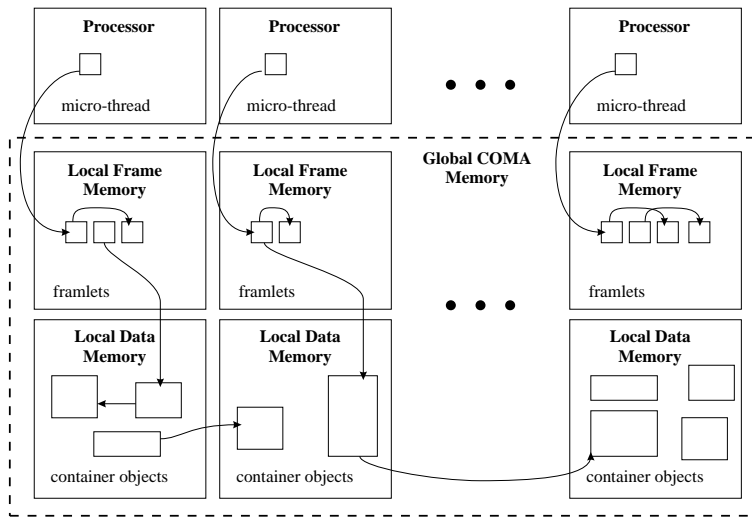


Figure 2. Automatic COMA-based scheduling (compare figure 1)

Message Format All messages sent on the network contain (at least) the following fields:

- A *tag*, for specifying what kind of transaction this message represents (e.g. operand *apply*, object *evict* — see below for a discussion of the various transactions);
- A *destination* field, which is an *object identifier* and which designates the recipient(s) of that message. The *destination* field usually specifies a data object, but can also explicitly identify an attraction memory site (see above);
- A *return address*, which is also an object identifier, which can be left blank. The exact meaning of this field depends on the transaction being performed, but typical uses include identifying the sender of a message, or specifying to whom data should be sent on a *read* operation;
- *Load* information, communicating approximately how heavily loaded the sending site was at the time the message was sent;
- A *data* field, containing the actual data being transmitted. This includes, minimally, the *state* of an object.

Architecture To avoid restricting this proposal unnecessarily, we assume only a very basic abstract architecture, as illustrated in figure 3. We assume a topology consisting of various *attraction memory sites*, connected by a *network*.

Each site consists functionally of a *processor*, a COMA protocol *controller*, and a local *memory*. The controller can be implemented either in hardware or in software. If the controller is implemented in hardware, any level 1 and level

2 caches will have to be allow for the controller and the processor to share the memory. Since even PC processors today come with a level 1 cache that supports a multiprocessor cache coherency protocol, this is not a difficult requirement to meet.

As explained above, each message has a *destination* object identifier. We stipulate that the network must be capable of making sure that each message arrives at (at least) all the sites where the destination object is currently resident. Note that the recipients of most messages will be determined dynamically (at run-time), based on the current distribution of the objects.

Various networks from the literature meet this requirement. The simplest would be a conventional shared bus, where *every* site receives a copy of *every* message (thus trivially fulfilling the requirement). More parallelism is possible using directory-based networks such those reviewed in [17], or the network proposed for SDAARC in [13].

Object States We can now specify the basic states for the objects in the attraction memories. We start with the 4 states in the well known MESI protocol: *Modified*, *Exclusive*, *Shared* and *Invalid*. The *Modified* state is not applicable in a COMA setting, so we can drop it. (“Modified” means modified relative to the main memory. In a COMA, there is no main memory). The *Shared* state however is no longer sufficient. We need to designate one copy of a shared data object as the “canonical” copy. Thus, we split the *Shared* state into two states: *Original* and *Clone*.

Finally, we need a transient state during object migration. We call this state *Leaving*. An object is *Leaving* its original home from the time it is evicted until the time the original site receives confirmation that the object has found

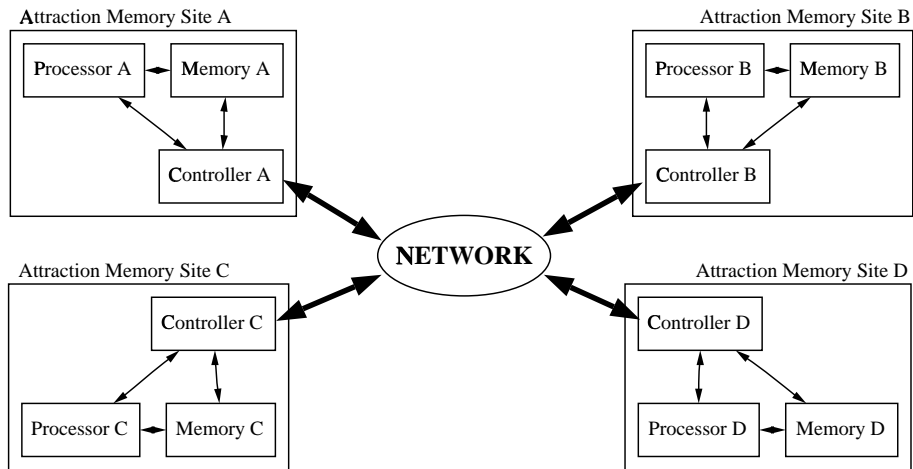


Figure 3. Abstract Architecture

a new home. We return to object migration in section 3.4, below.

We have transformed the MESI protocol into an ECOLI protocol. An object can thus be in one of 5 states, relative to a given attraction memory, as summarized in table 1. This can be compared to the 7 states needed for the SICS DDM: *Invalid, Exclusive, Shared, Reading, Answering, Waiting* and *Leaving* [11].

For a specific implementation, given details about the actual network, some simplifications of the protocol might be possible. For example, if the network is a shared bus, and if the bus arbiter can be relied upon to choose between multiple recipients when necessary, then the *Leaving* state is unnecessary, and the *Original* and *Clone* states can be merged into one *Shared* state. The current protocol is a compromise between the dual design demands of a simple cache coherency protocol and a simple, scalable network.

3.2. Basis System: Storing and Scheduling Microthreads

Microthreads are objects which produce and consume operands. As such, the most basic, underlying transaction between microthreads is the exchange of operands. We begin the exposition by considering a system supporting only the following operations: framelets are created, destroyed, and operands are passed between framelets. We delay treatment of random access data until section 3.3.

We stipulate that the frame cache coherency protocol should be a *write-update* (and not *write-invalidate*). The main costs of write-update protocols come from objects which remain in a cache long after they are needed there. Framelets, however, have a relatively short life-time, and a well defined moment when they can be erased (namely,

after execution).

We now wish to examine in more detail the events which take place when an operand is sent from one framelet to another. These actions are summarized in table 2 and illustrated in figure 4.

Let m_A be the microthread currently being executed, and let f_A be its framelet. Assume that an operand is to be sent from framelet f_A to framelet f_B . What then happens is dependent on the state of f_B in the attraction memory where m_A is being executed. The necessary transactions are obvious for write hits, i.e. when f_B is in either the *Exclusive* or the *Original* state: If the object is in the *Exclusive* state, the processor can simply store the operand in the framelet, and need not notify other processors or cause any network traffic whatsoever. If the object is in the *Original* state, the processor can immediately write to it, but must additionally send a message to the network so that any copies (necessarily in the *clone* state) can be updated. Storing an operand into a framelet may cause that framelet to become *ready*. In this case, the object goes into the scheduling queue for the local processor.

For the remaining three states (when f_B is in either the *clone*, *invalid* or *leaving* states), two options come into consideration: Seen from within the multithreaded paradigm, exchanging an operand is equivalent to *sending a message*. Seen from within the DSM paradigm (and thus also from with the COMA paradigm), exchanging an operand is a *write operation to a framelet*. This difference is significant: in the multithreaded case, the operand data is moved to the site where the framelet is resident; in the DSM case, the framelet is moved to the site performing the write operation. Note that the first option requires one network transaction, while the second option requires two (one to request the framelet, and one to deliver it).

State	Semantic
<i>E - Exclusive:</i>	The object is in the attraction memory, and in no other attraction memory.
<i>C - Clone:</i>	The object is in the attraction memory, is <i>not</i> the “canonical copy”, and is in at least one other attraction memory.
<i>O - Original:</i>	The object is in the attraction memory, is the “canonical copy”, and might also be in other attraction memories.
<i>L - Leaving:</i>	The object was in this attraction memory, has been evicted, but the confirmation of arrival elsewhere has not yet been received.
<i>I - Invalid:</i>	The object is not in this attraction memory. Invalid means the same as <i>Not Present</i> ”.

Table 1. The five states of the ECOLI protocol

State of f_B in site A	Action taken on <i>apply</i> to f_B .
<i>E - Exclusive:</i>	The operation can proceed directly, without causing network traffic.
<i>C - Clone:</i>	The operand is sent to f_B , along with the return address of the <i>site</i> , and load information describing the actual state of the site, including the state of the f_A .
<i>O - Original:</i>	The operation can proceed without waiting, but the operand has to be copied onto the network, in order to update the other copies.
<i>L - Leaving:</i>	Framelet f_B was here, and has been evicted. As such, we presumably do not want f_B back right now. The operand is sent to f_B , along with a blank return address, inhibiting the chance that f_B can be sent to this site.
<i>I - Invalid:</i>	Same as <i>Clone</i> .

Table 2. Actions take when sending an operand to a frame.

Having stipulated that a special, optimized cache should be provided for framelets, we are free to support both of these two options. The actual decision can then be delayed until run-time, when it can be made based on the current load balance between the sites. We make this decision at the site where the *Original* copy of the object resides, in order to avoid the need for a new transitory state. Basically, the writing site sends the operand away, and sometimes receives as a result at a later time the framelet. This action is essentially identical for the remaining three states, and is summarized in table 2.

We continue the scenario by examining what happens when an operand arrives at a remote site. If f_B is either in the *Exclusive* or *Original* state at a remote site, then that site stores the operand in f_B , and decides, based on the load and state information included with the operand, whether the load balance would (probably) be improved by evicting f_B . Details concerning object eviction are discussed below in section 3.4. The exact method by which load information

is communicated, and how the decision is made whether to accept or evict, depend on implementation-specific details and are left open for the moment.

Finally, f_B can also be in the *Clone* state at some or the remote sites. Since this is an update protocol, these sites simply update their copy, leaving its state unchanged. Even if the framelet becomes *ready*, the microthread does not go into the scheduling queue. This happens only at the site where f_B is in the *Original* (or *Exclusive*) state.

3.3. Incorporating Data

Up until now we have ignored the data cache. Having seen how the frame cache works, we can now take a new look at the familiar COMA data cache. In a “normal” DSM, a read miss causes two (“split-phase”) network transactions, one asking for data, and one transmitting it. A write miss causes three or more transactions, one to ask for the data, one to receive it (after which it is changed), and additional

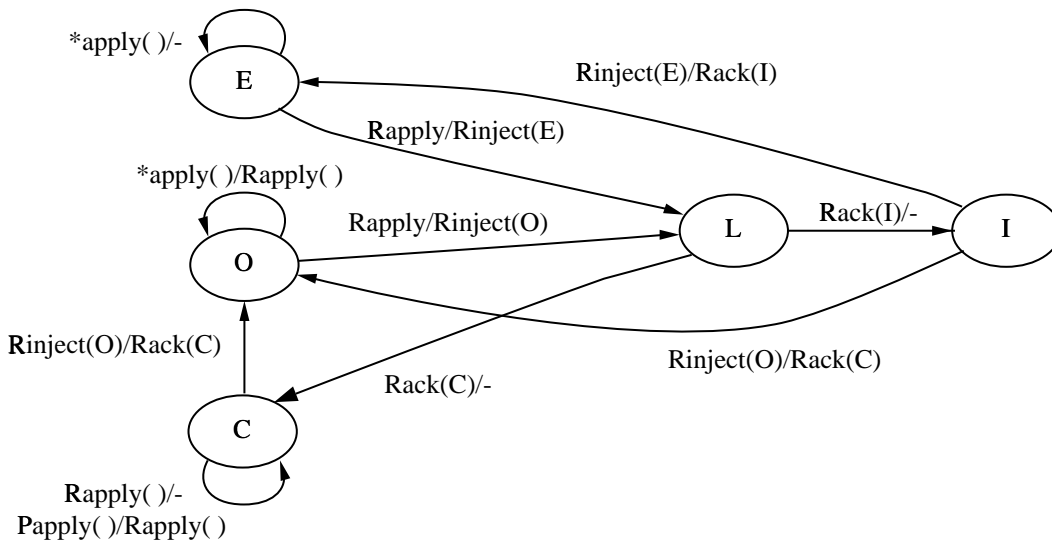


Figure 4. Cache coherency protocol for framelets. Edge Label Format: *incoming message/outgoing message*, **Message Format:** *ChannelMessage(State)*, where *Channel* is either *P* for the local Processor, *R* for Remote, or *** for either.

transactions to either update or invalidate other copies of the data.

Within the multithreaded paradigm, we see the situation differently. We see a random access data structure as a kind of frame with internal state. A store operation can thus be modeled as a function which takes three operands – a store address, the new value to be stored, and a return address – and returns one value – an acknowledgment (used for synchronization), which is sent to the return address. The return address will point to a framelet. Calling the store “function” changes the internal state of the data frame as a side-effect. Similarly, a load operation can be modeled as a function which takes two operands – a load address and a return address – and returns two values – the value retrieved from the load address and a synchronization, both of which are sent to the return address. Calling the load “function” has no side-effects for the data frame.

As such, we *could* model random access data structures with only the cache coherency protocol given so far (in section 3.2). However, this would be inefficient, since a load operation to an object in the *Clone* state would be passed to the site with the *Original* copy of the object. The *Clones* would, in other words, not be used efficiently. Further, choosing whether to send an object to the writing site on a write miss or not (as we did when applying an operand to a frame, see above) is not as attractive here, and is not a part of the protocol.

However, we have a fascinating new choice on a write miss: the data can either be sent to the site where the write

was performed, or to the site where the acknowledgment will be sent. We are currently experimenting with both options to see how the performance varies for different applications. Figure 5 shows the data cache protocol, assuming that the data is sent along with the acknowledgment (and *not* to the writing site).

3.4. Object Migration

Object migration is much the same for both the data and the frame caches. Objects migrate when they are *evicted* from an attraction memory where they currently reside. There are three important issues to consider, each of which is treated separately below: selecting objects to evict, selecting a destination for evicted objects, and how to actually perform the eviction.

Selecting Objects to Evict We stated one criteria for selecting an object to evict above: when a framelet receives an operand from a remote site, the controller may choose to send that framelet to that site. Object eviction can also happen either when space is needed for a new object (on an *inject* transaction), or simply when the protocol controller is idle and the attraction memory is too full. In the first case, we need to evict an object from the set in which the new object will be placed, while in the second case we can evict objects from any set deemed to be too full. Note that “too full” is an implementation-dependent term, which could be taken to mean simply “full”.

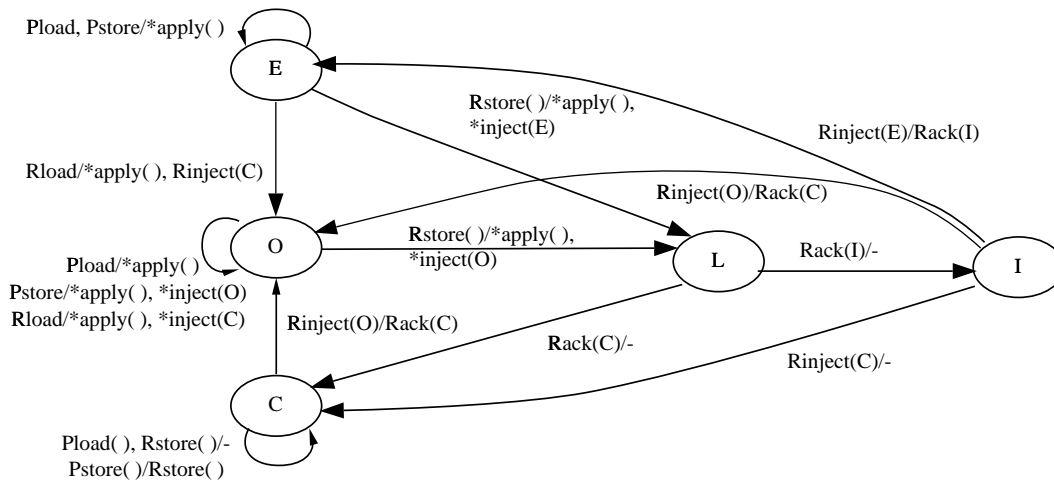


Figure 5. Cache coherency protocol for data (see figure 4 for edge label format)

Once a set is chosen, an object is chosen from that set according to the following criteria: First, objects which are in the *Invalid* state are chosen, since these objects represent empty slots. If no objects are in the *Invalid* state, objects in the *Clone* state are chosen, since they are the easiest to evict. Objects in the *Original* state are chosen next, and objects in the *Exclusive* state are evicted only when absolutely necessary. The least recently used *Exclusive* object will be evicted in this case.

Objects in the *Leaving* state are immune to eviction, since they are already *in the process* of being evicted. A problem arises if a set contains only objects in the *Leaving* state, since no object can be evicted to make room for the incoming object. This problem persists only until the evicting site receives confirmation that any one of the *Leaving* objects has arrived at its new home (see below). If hardware costs permit, a small buffer would be useful here to serve as a “waiting room” for this case. Ultimately however, even the buffer can become full. In this case, the *inject* transaction is forwarded to another site. A similar technique is used in the DDM, where an object can be forwarded from one site to another a number of times before eventually being bumped into some form of secondary storage [9]. However, it is better to define “too full” to mean “almost full”, and to start evicting objects *before* a given set is entirely full, in order to reduce the probability that a set can become full of *Leaving* objects.

Selecting a Destination for Evicted Objects Depending on the event which causes eviction, we can use different criteria to choose a destination for the evicted object. We saw above that an operand-apply operation can cause a framelet to migrate to the site from which the operand originated. Even when migrations are caused by congestion (sets be-

coming “too full”), the compiler can associate different objects together, and the new home for an evicted object can then be determined dynamically. For example, a framelet which contains a load (store) operation can often be associated with the address used in the load (store). Should the framelet be evicted, it can then be sent to the site currently containing the associated data object.

If all else fails, an object can simply be sent to that site with the least load, based on the most recent load information received on a message from that site. This method is far from perfect, since the load information will often be seriously out of date, but this method should be viewed only as a last resort.

Performing the Eviction The exact procedure followed during eviction depends on the state of the object being evicted: First, *Invalid* objects and *Clone* objects do not need to be evicted, they can simply be dropped from the attraction memory. Evicting objects in either the *Exclusive* or *Original* states is trickier. Since messages are (often) addressed to objects, and not to sites, it is essential that at least one copy of each object is present in some attraction memory at all times (until the object is erased). As such, objects in the *Exclusive* or *Original* States are kept in the *Leaving* state until the evicting site receives confirmation that the object has arrived at its new home.

While the object is in the *Leaving* state, all messages for that object are forwarded to the new site. This means that some messages might arrive twice. The network will need to be able to identify these messages and take them out of the traffic.

These state transitions are illustrated in figure 6.

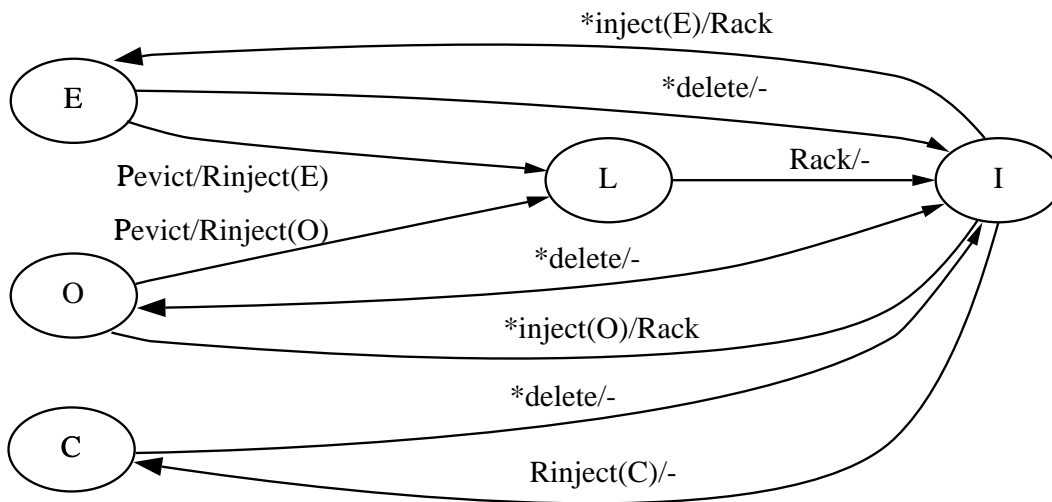


Figure 6. Cache coherency protocol for object creation, destruction and migration (see figure 4 for edge label format)

4. Related Work

The basic scheduling terminology (threads, frames, etc.) used in this paper comes from the multithreaded architectures, in particular from PRISC [15] and TAM [4]. See also [7] and [2]. Despite these similarities with multithreaded architectures, the scheduling algorithm here is significantly different from those found in the literature. As noted in Section 3, multithreaded architectures use explicit and/or demand-driven scheduling, while our proposal is both demand-driven *and* supply-driven.

This proposal can also be compared with other systems for automatic scheduling. Most automatic distribution schemes concentrate on scheduling loops and arrays. In contrast, our proposal can also handle irregular problems and dynamic data structures. It can also better adapt to dynamically changing loads on parallel machines shared by multiple users and/or multiple tasks.

An adaptive scheduler which distributes computation based on supply and not demand was presented in [8] for a very special class of algorithms (tree-structures) and multiprocessor topologies (rings). Despite the obvious difference in scope, important similarities remain. We find the empirical results presented in [8] very encouraging.

COMA-based scheduling appears to be unique to this proposal. Other COMAs do not treat program partitioning, but rather utilize the (static or dynamic) scheduling already built into existing parallel programs (see e.g. [10]). Multithreading is used with the DDM in [14], but again only to accelerate programs which have already been partitioned.

5. Conclusion

Ultimately, this architecture brings us one step closer to a new model of computation: We have bridged the gap between distributed shared memories and the dataflow paradigm. We have an abstract model of computation which is inherently parallel. The mapping of this abstract model onto the available hardware resources (memories and processors) is conceptually separate from the specification of the program.

This approach will be most suitable for irregular problems, or for applications where programming costs are a major factor – that is, for problems where the high costs of parallel programming currently deter the use of distributed computation.

This project is still in its earliest stages. We are currently working on modeling, simulating and verifying the correctness of the protocol. Simulation for performance estimation is also underway.

Once we have gained more experience with the basic model, we can extend this proposal to optimize important classes of data structures and simulation applications. The idea of representing random access data as functions with internal state opens up possibilities for modeling other objects, e.g. in neural networks or in simulation applications. Further, the virtual shared pool of framelets represents an interesting new container data structure (a form of distributed *set*) which could also be useful if made directly available in programmers. For example, it would be useful for storing the populations in evolutionary algorithms.

Acknowledgments Figures 3 – 6 were drawn entirely or partially by the dot program from AT&T. URL <http://www.research.att.com/sw/tools/graphviz/>.

References

- [1] G. S. Almasi and A. Gottlieb. Kendall Square Research KSR1. In *Highly Parallel Computing*, section 10.3.3, pages 549–553. Benjamin/Cummings Publishing Company, second edition, 1994.
- [2] M. Annavaram and W. A. Najjar. Comparison of two storage models in data-driven multithreaded architectures. In *Eighth IEEE Symposium on Parallel and Distributed Processing (SPDP)* [12], pages 122–129.
- [3] Arvind, L. Bic, and T. Ungerer. Evolution of dataflow computers. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, 1991.
- [4] D. Culler, A. Sah, K. Schauer, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. of 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa-Clara, CA, Apr. 1991. (Also available as Technical Report UCB/CSD 91/594, CS Div., University of California at Berkeley).
- [5] D. E. Culler, S. C. Goldstein, K. E. Schauer, and T. von Eicken. TAM — A compiler controlled Threaded Abstract Machine. In *Journal of Parallel and Distributed Computing, Special Issue on Dataflow*, June 1993.
- [6] J. B. Dennis. First version of a data flow procedure language. In *Lecture Notes in Computer Science*, volume 19. Springer Verlag, 1974.
- [7] S. C. Goldstein, K. E. Schauer, and D. Culler. Enabling primitives for compiling parallel languages. In *Languages, Compilers and Run-Time Systems for Scalable Systems*, pages 153–168. Kuwer Academic Press, 1996.
- [8] D. E. Gregory, L. Gao, A. L. Rosenberg, and P. R. Cohen. An empirical study of dynamic scheduling on rings of processors. In *Eighth IEEE Symposium on Parallel and Distributed Processing (SPDP)* [12], pages 470–473.
- [9] E. Hagersten. *Toward Scalable Cache Only Memory Architectures, 2nd Edition*. PhD thesis, Swedish Institute of Computer Science, Royal Institute of Technology, Stockholm, Sweden, July 1993.
- [10] E. Hagersten, A. Landin, and S. Haridi. DDM — A Cache-Only Memory Architecture. *IEEE Computer*, 25(9), 1992.
- [11] S. Haridi and E. Hagersten. The cache coherence protocol of the Data Diffusion Machine. In *Proceedings of the PARLE 89*, volume 1, pages 1–18. Springer-Verlag, 1989.
- [12] IEEE. *Eighth IEEE Symposium on Parallel and Distributed Processing (SPDP)*, New Orleans, LA, Oct. 1996. IEEE Computer Society Press.
- [13] R. Moore, B. Klauer, and K. Waldschmidt. A combined virtual shared memory and network which schedules. In *International Conference on Parallel and Distributed Systems (Euro-PDS '97)*, Barceona, Spain, June 1997.
- [14] H. L. Mueller, P. W. A. Stallard, and D. H. D. Warren. Hiding miss latencies with multithreading on the Data Diffusion Machine. In *Proceedings of the 1995 International Conference on Parallel Processing, ICPP'95*, volume 1, pages 178–185, Oconomowoc, WI, Aug. 1995. CRC Press.
- [15] R. S. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, pages 390–405, Portland, Oregon, Aug. 1993. Springer Verlag LNCS 768.
- [16] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple COMA. In *First IEEE Symposium on High Performance Computer Architecture*, pages 276–285, Raleigh, North Carolina, Jan. 1995.
- [17] P. Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, 1990.
- [18] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active messages: a mechanism for integrated communication and computation. In *Proc. of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. (Also available as Technical Report UCB/CSD 92/675, CS Div., University of California at Berkeley).