ENSEMBLE: A Communication Layer for Embedded Multi-Processor Systems

Sidney Cadot  Frits Kuijlmans  Koen Langendoen  Kees van Reeuwijk  Henk Sips

Faculty of Information Technology and Systems
Delft University of Technology, The Netherlands
{frits,koen,reeuwijk,sips}@pds.ti.tudelft.nl

ABSTRACT

The ENSEMBLE communication library exploits overlapping of message aggregation (computation) and DMA transfers (communication) for embedded multi-processor systems. In contrast to traditional communication libraries, ENSEMBLE operates on n-dimensional data descriptors that can be used to specify often-occurring data access patterns in n-dimensional arrays. This allows ENSEMBLE to setup a three-stage pack-transfer-unpack pipeline, effectively overlapping message aggregation and DMA transfers. ENSEMBLE is used to support Spar/Java, a Java-based language with SPMD annotations. Measurements on a TriMedia-based multi-processor system show that ENSEMBLE increases performance up to 39% for peer-to-peer communication, and up to 34% for all-to-all communication.

1. INTRODUCTION

The application domain for embedded systems is rapidly expanding. Embedded systems are now also used for multimedia processing (e.g., HDTV) that requires a high level of sustained performance. On the other hand, the life-time of applications is decreasing. This calls for a hybrid solution, in which a general-purpose processor is used for flexibility and multiple embedded processors (DSPs) for high-throughput signal processing. The embedded processors can be configured as a pipeline, or as a processor farm. The latter architecture is often implemented by connecting all processors (host and DSPs) to a shared bus (e.g., PCI) or network (e.g., Ethernet). We refer to this setup as an embedded multi-processor system. In such a system all processors can communicate with each other, which greatly enhances flexibility.

To ease application development for embedded multi-processor systems it is crucial to develop compiler and runtime-system support to handle difficult and error-prone tasks like processor synchronization and data transport between (heterogeneous) processors (e.g., host to DSP). Many parallel programming systems exist ranging from library-based systems (PVM [16], MPI [10]) to language-based systems (HPF [6], CC++ [5], Orca [2]). The latter systems are easy to program because of the parallelizing compiler that hides the complex interface to the parallel hardware. The performance, however, is often compromised because of the layered approach (OS/runtime-system/compiler) taken to achieve portability. The alternative of writing explicitly parallel programs using communication libraries is not very appealing because of the large penalty in development costs -- parallel programming at such a low level is difficult and error-prone.

Our approach consists of building a complete tool chain for application development targeting embedded multi-processors. We combine ease-of-programming (compiler) and application performance (efficient DMA-based communication). We support data parallel (SPMD) programming through the Spar/Java language, which is a Java derivative with explicit support for scientific computations [14, 15]. The Spar/Java compiler recognizes annotations for data and code placement, and automatically generates a parallel program with explicit communication. On embedded systems communication is handled by the ENSEMBLE layer, which has been explicitly designed to take advantage of the hardware capabilities (i.e., DMA engines) present in embedded multi-processors. Furthermore we tightly integrated the compiler and ENSEMBLE to overcome the performance penalties (e.g., buffer copies) associated with portable communication libraries.

In this paper we describe the ENSEMBLE communication layer, and evaluate its performance on an Athlon/TriMedia system.

2. ENSEMBLE

Most embedded processors are capable of initiating asynchronous DMA transfers, so that communication and processing can be overlapped. In ENSEMBLE we exploit this feature by overlapping simultaneous DMA transfers and buffer packing and unpacking as much as possible. The communication performed by data-parallel Spar/Java programs is implicit; whenever a processor references data that resides on another processor, the compiler generates a communication event. The performance of so-called element-wise communication is poor. Therefore, the compiler uses message aggregation to send multiple data elements in a single message. With ENSEMBLE we are able to overlap message aggregation (computation) and communication. This is not
final int SZ = 300;
int A[] =
    <S>on=Lambda (i) P[(local 0)]> new int[SZ];
int B[] =
    <S>on=Lambda (i) P[(cyclic i)]> new int[SZ];
int C[] =
    <S>on=Lambda (i) P[(cyclic i)]> new int[SZ];
</S> independent $\$ foreach ( i := 0:A.length ) {
    A[i] = B[i] * C[i];
}

Figure 1: Spar/Java code fragment for inproduct.

MESSAGE = MSGBUF;

// all processors: pack outgoing message
for (int i=proco;i<A.length;i+=P) {
    MESSAGE++ = B[i] * C[i];
}

// all processors: send message to owner
send(MSGBUF,owner(A));

// owner: receive and unpack incoming messages
if (proco==owner(A)) {
    for (int pr=0;pr<P;pr++) {
        receive(MSGBUF,pr);
        MESSAGE=MSGBUF;
        for (i=pr;i<A.length;i+=P) {
            A[i] = *MESSAGE++;
        }
    }
}

Figure 2: Spar/Java-generated C++ code.

The Spar/Java compiler performs a sophisticated analysis to identify opportunities for message aggregation [13]. With
the inproduct code, the compiler infers that B and C are identically
distributed, so it can compute the expression
B[i] * C[i] at the owning processor and send the result
to processor 0. The individual results computed at one
processor are aggregated in a single message. Processor 0 pro-
cesses all incoming messages (including the message sent
by itself) by storing the result values in the array A.

Note that a message is completely assembled before being
sent; likewise, a message is first received before its contents
is processed. This setup rules out overlapping message
aggregation and communication by ENSEMBLE. Furthermore,
the Spar/Java compiler determines in which order the
messages are processed. If messages happen to arrive
in a different order at runtime, they cannot be processed
immediately, but must be buffered. To avoid unnecessary
waiting, we would like to process messages on a first-come
first-serve basis.

Overlapping message aggregation and communication re-
quires a tight integration: either the compiler must be
made communication aware (e.g., address fragmentation for
pipelining), or the message passing layer must provide a
higher-level interface (e.g., scatter-gather message vectors).
With ENSEMBLE we take the latter approach and operate on
(size-dimensional) data descriptors instead of contiguous
buffers. Furthermore, we require that all sends and receives
are registered before invoking ENSEMBLE to perform the ac-
tual data transfers. This registration-execution mechanism
provides ENSEMBLE with the opportunity to (re) schedule
data transfers dynamically to match availability of data
(buffers) at source (destination) processors.

We modified the Spar/Java compiler to generate
ENSEMBLE-style code as shown in Figure 3. At the send-
ing side nothing seems to have changed, but the actual
transmission of the message is delayed until emb_fence
is invoked. At the receiving side ENSEMBLE is instructed to
unpack messages directly into array A; the data must be
assigned by cycling through the array with stride P starting
at index pr. The net effect is that the data transmission
can be overlapped with the unpacking in any order that the
messages arrive.

2.1 Data descriptors

Both source and destination of a data transfer in
ENSEMBLE must be specified using ‘data descriptors’. These
allow a variety of often-occurring data access patterns to be
specified as either the source or destination of a transfer.

The data descriptors used in ENSEMBLE are based on the
concept of selecting the elements of a one-dimensional array
with index-values ranging from a lower-bound L up to an
upper-bound U; each time incrementing the index with a
stride S. Actually, the specification as used in ENSEMBLE
uses a slight modification of this scheme, using a fixed lower-
bound L=0; instead of the upper bound U the ‘element-
count’ C is used, for which the following holds:

\[
C = \left\lfloor \frac{U}{S} \right\rfloor + 1
\]

The (S,C) specifications can be nested to allow the
specification of complex data access patterns. The lack of
lower-bounds is compensated by having a ‘base-element
index) $B$, which designates the element in the source array that serves as the starting point for strided packing or unpacking. The advantage of this approach compared to using a full $(L,U,S)$ specification is that it suffices to have one base-index value $B$ per $(S,C)$-specification list, instead of having one lower-bound value per $(L,U,S)$ specification.

An obvious way in which nested $(S,C)$-specifications can be used is to traverse several dimensions of a multi-dimensional array. As an example, consider a 3-dimensional array, residing on one processor, that is to be cyclically distributed among three processors (Figure 4). The colors of the layers in the $z$-direction indicate the destination processors; the sender will need to specify three ‘send’ operations. Table 1 shows the three data-descriptors needed in specifying the ‘send’ operations: each of these consists of one first-element offset $B$, and three $(S,C)$ pairs. Note that the white processor will receive more data (4 vs. 3 planes) than the other processors resulting in different $C_0$ counts.

![Figure 4: 3D array to be cyclically distributed.](image)

Table 1: Data descriptors for the three targets.

<table>
<thead>
<tr>
<th></th>
<th>$B$</th>
<th>$(S_0, C_0)$</th>
<th>$(S_1, C_1)$</th>
<th>$(S_2, C_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>white</td>
<td>0</td>
<td>(1,20)</td>
<td>(20,8)</td>
<td>(480,4)</td>
</tr>
<tr>
<td>light-gray</td>
<td>160</td>
<td>(1,20)</td>
<td>(20,8)</td>
<td>(480,3)</td>
</tr>
<tr>
<td>dark-gray</td>
<td>320</td>
<td>(1,20)</td>
<td>(20,8)</td>
<td>(480,3)</td>
</tr>
</tbody>
</table>

Apart from specifying which elements to use, the base offset and $(S,C)$-pairs also specify an order in which the elements must be traversed. In ENSEMBLE, the first-specified $(S_0, C_0)$ pair denotes the inner loop. It is allowed for the source and destination data-descriptors to describe quite different array-traversals; only the total number of elements must be equal. This feature can be used in interesting ways, in which the ENSEMBLE layer is effectively used to perform a data-transforming operation. Suppose we have a square 100 x 100 matrix at processor 0, and that we need the transposed contents of this matrix on processor 1. Table 2 shows the way in which this can be accomplished using the ENSEMBLE data-descriptors.

![Table 2: 100x100 matrix transpose descriptors.](image)

<table>
<thead>
<tr>
<th></th>
<th>$B$</th>
<th>$(S_0, C_0)$</th>
<th>$(S_1, C_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sender</td>
<td>0</td>
<td>(1,100)</td>
<td>(100,100)</td>
</tr>
<tr>
<td>receiver</td>
<td>0</td>
<td>(100,100)</td>
<td>(1,100)</td>
</tr>
</tbody>
</table>

For many operations in which standard distributions such as 'block', 'cyclic', and 'block-cyclic' distributions are used, it is possible to express the send- and receive-operations with one or two data-descriptors.

int emb_init(int argc, char **argv, int nproc);
void emb_done(void);

int emb_PID(void);
int emb_HP(void);

void emb_barrier(void);

void emb_send(int dst, int elmSize, void *base, int offset, int nstrides, int *strides, int *counts);
void emb_sendblock(int dst, void *base, int sz);
void emb_recv(int src, int elmSz, void *base, int offset, int nstrides, int *strides, int *counts);
void emb_recvblock(int src, void *base, int sz);
void emb_fence(void);

Figure 5: ENSEMBLE interface.

2.2 API

This section describes the small set of primitives provided by the ENSEMBLE layer. The Application Programming Interface (API) comprises ten calls, see Figure 5.

The `emb_init()` call must be called once at startup, before using any other function of the ENSEMBLE layer. The nproc parameter specifies the number of processors that should be brought online when bootstrapping the multi-processor system. The value returned by the `emb_init()` function indicates the processor ID, ranging from 0 to (nproc-1), inclusive. When the `emb_init()` call returns, it is guaranteed that the parallel system is online, that the program is running on a processor that participates in the parallel system, and that other ENSEMBLE calls may be performed. The function cannot fail; if the bootstrapping process fails within the `emb_init()` call, ENSEMBLE will immediately abort program execution.

The `emb_barrier()` call implements a full barrier, synchronizing all processors. This means exactly the following: upon return from the i-th call to `emb_barrier()`, all processors are guaranteed to have at least entered the i-th call to `emb_barrier()`.

The `emb_send` and `emb_recv` functions are used to register 'send' and 'receive'-type operations, which will be executed upon the next call to `emb_fence()`. The dst and src arguments specify the 'peer processor' of the specified operation; a matching call must be executed on that processor. Send-to-self is allowed, and should also have a matching receive-from-self call. The `elmSz` argument specifies the element size of the array elements, while the base argument specifies the array pointer. The offset argument specifies the element relative to which all strides will be performed; this is the $B$ value that was discussed in Section 2.1. The `nstrides`, `strides`, and `counts` parameters specify the number of nested strides, and the values $S$ and $C$, respectively. These are processed as soon as the registering call is made; there is no need to preserve these arrays until the next `emb_fence()` call.

The `emb_sendblock()` and `emb_recvblock()` calls are shorthand functions for sending and receiving contiguous
buffers. These are treated exactly like the corresponding `emb_send()` and `emb_recv()` calls, and may be freely mixed.

Invoking the `emb_fence()` starts the actual data transfers involved in the sends and receives registered since the last call completed. The order in which the sends (and receives) are handled is left unspecified, except that sends to the same destination are handled in FIFO order. This provides ENSEMBLE with the freedom to schedule a send as soon as its destination is ready to accept the incoming data (i.e., entered the corresponding `emb_fence()` call). The `emb_fence()` call returns as soon as all pending sends and receives are completed; in case no transfer operations are specified (which is allowed), it will simply fall through.

### 2.3 Streaming

The goal of ensemble is to overlap message aggregation (computation) and DMA transfers (communication). This calls for a pipelined approach with three stages: packing (at source processor), transmitting (DMA engine at source), and unpacking (at destination processor). To reduce startup costs we use relatively small buffers that are passed down the pipeline; each DMA across the PCI bus transfers 4 KB (or less for the last fragment) of data. These transfer buffers are also the basic unit of flow control between sender and receiver. Flow control is implemented per sender/receiver pair: each node allocates a small number (4) of buffers per possible sender `(4 x nproc-1)` in total. Whenever a buffer is emptied (unpacked) at the destination it is returned to the sender immediately; the receiving processor updates the administration at the sender node. The arrival of a buffer at the receiver is similarly indicated by the sender writing in the receiver’s administration. (By appending an ‘available’ flag at the end of the transfer buffer, the signalling can be piggybacked nearly for free onto the DMA transfer.)

#### 2.3.1 Zero copying

Before describing the (un)packing of data elements into and out of transfer buffers it is important to note that in some cases it is better to avoid the complete pipeline altogether. When the source data elements are stored at consecutive memory locations (e.g., a matrix row) and the locations of the elements at the destination are consecutive also, then the most efficient communication method is a single DMA transfer avoiding two intermediate copies. To exploit zero-copying ENSEMBLE performs a handshake between sender and receiver at the start of each communication to check if both source and destination are contiguous buffers. If so, a single DMA transfer is requested streaming data directly from source memory to destination memory. If only one buffer is contiguous, one copy is saved by directly DMA-ing the from a transfer buffer. Otherwise, the complete pack-transmit-unpack pipeline is started (two copies).

#### 2.3.2 Dimension reduction

When `emb_send()` or `emb_recv()` calls are performed, the client program (generated by the Spar/Java compiler) passes a list of a stride/count specifiers. To enhance performance ENSEMBLE processes this data description to identify consecutive elements (and dimensions). For example, a matrix row specified as a sequence of `cnt` elements (stride 1, size `sz`), can be regarded as a single buffer (size `cnt x sz`). Likewise, a sequence of rows can be collapsed to a single buffer. The steps performed when translating a data descriptor to an internal ‘stride-copy state’ descriptor recognize such cases. This process is called ‘dimension reduction’ as it may result in a stride specification with less dimensions than the original specification. The dimension-reduction process as performed in ENSEMBLE comprises the following steps:

1. The ‘element size’ is included as the very first dimension of the stride-copy descriptor.
2. Next, all other dimensions are added. If possible, an added dimension is combined with the previous dimension. This is possible if the stride `S_{i+1}` of a dimension to be added is equal to the count `C_i` of the previous dimension, multiplied by the stride `S_i` of the previous dimension.
3. When all dimensions have been processed, we drop the inner dimension and use its size as the element size of the resulting descriptor.

These steps ensure that the minimal number of dimensions is specified needed to visit the same array elements as the original specification, in the same order. If the number of dimensions is reduced to zero, the data is available as one contiguous buffer (with length ‘element size’), indicating that we might engage a zero-copying transfer.

#### 2.3.3 Data (un)packing

During the execution of an `emb_fence` operation, the actual message aggregation takes place. The stride-copy descriptor is used to copy data from the source array into the next transfer buffer. The stride-pack routine can handle arbitrary specifications of memory traversal; however, much effort was put into the efficient operation for the most important case in which the packing code can be written as operations on aligned 4-byte elements (holding primitive types like float and int). The stride-pack code contains a highly optimized unrolled loop for this case achieving a throughput of 150 MB/s on our embedded TriMedia target. To use this highly-optimized code in as many cases as possible we check if the element size (`s`) is a multiple of four. If so, we effectively introduce a new inner dimension with count `s/4`.

### 2.4 Communication scheduling

The `emb_fence` routine will perform all registered send and receive operations in the correct (FIFO) order. This is achieved internally by providing each processor with both an incoming and an outgoing queue of registered operations with each processor in the system, including itself. Thus, `nproc x 2` queues must be managed at each processor. The queues are filled with stride-copy descriptors by the `emb_send()` and `emb_recv()` calls. The communication scheduler loops over all queue trying to advance the pending communication action at the head of each queue until all registered sends and receives have been performed. We constructed the scheduler such that each action on some queue is non-blocking and involves a small amount of work (‘pack the next buffer’ being the most expensive). This avoids deadlock and ensures fair progress (round robin scheduling); multiple pipelined communications are effectively interleaved.

With each queue we associate a state machine that records where execution will continue on the next round of the scheduler. A communication is initiated by the receiver who posts a request at the sender. The sender polls for
the request. When it arrives the sender checks if a zero-copy transfer can be used. If so, it initiates the DMA transfer, and modifies the queue state to check for completion. When a future action detects the completion of the DMA, it notifies the receiver and removes the strided-copy descriptor from the head of the queue. A buffered transfer is handled by checking if a free buffer is available. If not, we yield control back to the scheduler. Eventually a buffer will become available, it is filled by strided-pack, and a DMA transfer is initiated. (The strided-pack routine records in the queue state where to proceed for the next fragment.) When detecting the completion of the DMA in a future action, we do not need to explicitly signal the receiver because of the ‘available’ flag appended to the buffer, but may continue with allocating, filling and transmitting the next fragment. When the DMA of the last fragment completes we remove the strided-copy descriptor from the head of the queue. When all queues are emptied, emb_fence returns control to the caller.

3. IMPLEMENTATION

We implemented ENSEMBLE (and the Spar/Java compiler) on a heterogeneous multi-processor system consisting of one host CPU (AMD Athlon [1]) and three multimedia DSPs (Philips TriMedia [12]) connected by a PCI bus. The Spar/Java compiler generates C++ code, which is then compiled for the Athlon and cross-compiled for the TriMedia. The Athlon executable downloads the TriMedia code on the three embedded processors and initiates execution. The three TriMedia processors perform the actual parallel computation and return their output to the Athlon.

The AMD Athlon host processor is a high performance, x86-compatible microprocessor. In our system it runs at 700 MHz and is equipped with a 64 KB level-1 data cache, a 512 KB L2-cache, and 256 MB of SDRAM. The specifications of the embedded TriMedia TM1000 processor are more modest: 100 MHz processor, 32 KB instruction cache, 16 KB data cache, and 8 MB of memory.

The Athlon processor is under control of the Linux operating system. Linux supports virtual memory, which complicates the ENSEMBLE implementation for two reasons: 1) a TriMedia addressing Athlon memory uses physical addresses, 2) to avoid Linux from swapping out pages, DMA-like areas must be “pinned”. Since the user has no control over the relation between (consecutive) virtual addresses and hardware pages, the DMA-like area is limited to 4 KB (1 page). This prevents ENSEMBLE from using zero-copy transfers between the Athlon and a TriMedia.

The TriMedia boards are equipped with a DMA engine that operates independently and can be controlled from the embedded ENSEMBLE software running on the TM1000 processor. The caveat, however, is that the DMA engine directly operates on SDRAM without consulting the (copy-back) data cache. Therefore, ENSEMBLE code is cluttered with cache-copyback instructions to update memory with cache values, and cache-invalid instructions to avoid stale data in the cache when memory has been updated by the DMA engine.

Another characteristic of our system that has caused much grief is that for some reason DMA transfers and PIO (programmed I/O) can not be mixed freely. Handshaking by directly writing to another processor’s memory occasionally fails because the PIO write does not get through to the destination’s memory when contending with concurrent DMA transfers for the PCI bus. This results in data loss and, consequently, in deadlock. We now use DMA for all accesses to remote memory at the expense of additional latency.

Figure 6 shows the transfer rates for different message sizes when using DMA. The rates are measured with a hand-coded program issuing DMA-send/receive requests in a tight loop. Note that ‘get-type’ operations are invariably faster than ‘put-type’ operations. When performing a ‘get-type’ operation, the requester must wait for it to finish, thereby introducing a synchronization effect with the remote hardware. ‘Put-type’ operations on the other hand can normally proceed at full-speed, using buffers between the sender and receiver and requiring no end-to-end synchronization. ENSEMBLE therefore uses sender-initiated communications, except for Athlon-to-TriMedia transfers, because the DMA engine on the Athlon has too restricted capabilities: it can only address ISA (< 16 MB) devices. Additional measurements, including Athlon-to-TriMedia and concurrent DMA transfers, are reported in [4].

4. RESULTS

To demonstrate the performance of ENSEMBLE (supporting Spar/Java) we present two benchmarks capturing the often occurring peer-to-peer and all-to-all communication patterns. Other patterns, like gather and scatter, show intermediate behavior since peer-to-peer (min) and all-to-all (max) are the two extremes in terms of data rate.

4.1 Peer-to-peer

Figure 7 shows the DMA throughput achieved on our hardware for peer-to-peer communication. Processor 1 transmits a number of matrix elements (4-byte floats) to processor 0. For reference the basic DMA put performance from Figure 6 is plotted (top most curve). We measured the performance of ENSEMBLE under the most favorable conditions: the matrix elements are stored consecutively. In this case ENSEMBLE issues a zero-copy DMA transfer. The ‘ENSEMBLE zero copy’ curve shows that for small data sizes the ENSEMBLE overhead is significant, for example, when sending 64 consecutive floats (256 bytes) ENSEMBLE achieves a throughput of just 11.4 MB/s vs. 46.8 MB/s that is obtained with hand-coded DMA transfers. The overhead is caused by two factors: 1) the preprocessing
of data descriptors (dimension reduction, etc.), and 2) the handshaking between sender and receiver infer the possibility for a zero-copy transfer. The relative impact of the ENSEMBLE overhead rapidly decreases as the data size is increased; beyond 4 KB it is insignificant. Many programs repeat the same communication action many times inside a (nested) loop. A straightforward optimization moves the data descriptor preprocessing out of the loop to amortize the overhead over all iterations. This will reduce the ENSEMBLE overhead for small transfer sizes. Finally, we measured the performance of Spar/Java on top of ENSEMBLE. Spar/Java adds almost no overhead in this simple case, so the curve closely follows the ‘Embedded zero copy’; for clarity it is not shown.

To study the performance of ENSEMBLE in less favorable conditions, we forced it to execute the (buffered) pack-transmit-unpack pipeline. The ‘ENSEMBLE buffered’ curve shows the throughput across all data sizes. With 64 elements the performance starts out at 63% of the zero-copy case, reduces to just 45% for 1024 elements, and then gradually increases to 77% for 256K elements. This behavior is a consequence of ENSEMBLE using 4 KB buffers for (un)packing up until 1024 floats (= 4 KB) pipelining is not effective because only one buffer is used. For large data sizes one would expect the pipeline startup costs (pack and unpack of one buffer) to become negligible. This is clearly not the case. The reason is that for large data sizes the pack performance reduces to 70 MB/s due to the need to load (flush) data in (from) the cache; the 150 MB/s rate reported in Section 2.3.3 was obtained for a single 4 KB buffer that together with the source data exactly fits in the data cache. Thus, for large data sizes the bottleneck in the pack-transmit-unpack pipeline is the pack phase (70 MB/s), not the DMA speed (93 MB/s).

Again we studied the impact of Spar/Java. We measured the performance of a simple test program that assigns a cyclically distributed array (residing on both processors) to an array that is completely local to processor 0. The resulting curve is labeled ‘Spar buffered’. Note that the throughput is considerably lower than the corresponding ‘ENSEMBLE buffered’ curve. The Spar/Java compiler is not the cause of this performance drop (i.e., it does not introduce additional copies), but again the performance of the stride-based routine. Since the data is cyclically distributed, the array elements at processor 1 are laid out in memory with stride 1 (4 bytes). When packing the elements only 50% of the data in each cache line is used. The additional loads to fill the cache degrade the packing rate from 70 MB/s to 51 MB/s.

To measure the benefits of pipelining we ran the same Spar/Java test program on ENSEMBLE in synchronous mode. ENSEMBLE waits for each DMA to complete before continuing with other work (e.g., packing). This effectively simulates the Spar/Java compiler doing all message aggregation before invoking a traditional communication subsystem. Comparing the ‘Spar synch’ and ‘Spar buffered’ curves in Figure 7 shows that pipelining is effective. For example, when communicating 64 KB elements pipelining raises the throughput from 35.5 to 49.9 MB/s, an increase of 39%. Again we can see the effect of using 4 KB buffers: up until 1024 floats the complete data set fits into a single buffer so no performance is gained.

4.2 All-to-all

We repeated the measurements for the all-to-all communication pattern, in which each processor sends data to all others (including itself). We do not broadcast the data, but send separate messages to individual processors. The performance results are shown in Figure 8. For each data size, six messages of that size are transferred across the bus.

In comparison to the peer-to-peer pattern, the throughput obtained for all-to-all communications is higher (except for ‘Spar buffered’ with large data sets). ENSEMBLE (both zero copy and buffered) even exceeds the raw DMA speed. The reason is that concurrent DMA transfers issued by multiple processors better utilize the capacity of the PCI bus (132 MB/s). The highest throughput is achieved by ‘ENSEMBLE zero copy’: 107.4 MB/s. The performance of ‘ENSEMBLE buffered’ (104.2 MB/s) approaches the zero-copy throughput closely for large data sizes. This shows that the packing speed (70 MB/s) is not the bottleneck in the pipeline as it was in the peer-to-peer case. Again this is a consequence of performing multiple transfers in parallel: $3 \times 70\ MB/s > 107.4\ MB/s$.

At the Spar/Java level the performance improvement over peer-to-peer communication is only observed for small data sizes (< 4 K floats). For large data sizes the performance of ‘Spar buffered’ even decreases below the 49.2 MB/s achieved
with a data size of 1024 elements. The reason is (again) the data layout. The Spar/Java all-to-all test program assigns a cyclically distributed array to a replicated array. Since three processors are involved (against two for peer-to-peer) the cache lines holding the source data are used even less efficiently (33%) than with peer-to-peer (50%). The end result is that the buffered all-to-all throughput for 256K elements (437 MB/s) is less than the buffered peer-to-peer throughput (507 MB/s). In the non-pipelining case ('Spar synch') all-to-all performs slightly better than peer-to-peer: 373 MB/s versus 360 MB/s.

The effects of overlapping message aggregation and data transfer ('Spar buffered' versus 'Spar synch') for all-to-all are different than for peer-to-peer. With peer-to-peer no difference was seen for small data sizes because of the 4 KB transfer buffers. With all-to-all, however, pipelining pays off for all data sizes. The reason is that even if the data fits into one transfer buffer, the time waited for completion of the DMA transfer can be used to pack a message destined for another processor. The largest gain is obtained for 1204 floats: 'Spar buffered' achieves a 34% increase over 'Spar synch'. For large data sizes the poor cache utilization decrease the benefit of pipelining: just a 17% increase for 256K elements.

We plan to enhance our compiler to store all local data of a distributed array in one contiguous buffer. This reduces the memory footprint of a distributed array, and allows the cache to be used more efficiently. This will raise packing speeds considerably and increase the relative impact of pipelining. A potential disadvantage is that every array access requires an additional global-to-local offset translation, but as we describe in [13] this calculation can usually be lifted out of loops.

5. RELATED WORK

Overlapping computation with communication is a well-known concept; many message passing systems, including MPI, provide asynchronous send (and receive) primitives. Making effective use of these primitives is often the task of the programmer, whereas we use a compiler-based approach. When sending long messages, manual fragmentation and assembly to implement a pipeline is cumbersome. Therefore, thin message passing layers for fast networks like Fast Messages for Myrinet [8], provide a streaming interface where a message may be presented as many small parts each part is written into the stream using a separate call. Although this efficiently supports scatter/gather message vectors used in many protocol stacks, the function call overhead is prohibitive for sending strided data like Spar/Java requires.

Implementing message streaming on a shared bus architecture raises the issue of scheduling competing data transfers. ENSEMBLE uses a two-stage approach: round-robin scheduling within a node, and PCI bus arbitration between nodes. This simple scheme performs well since all communication in Spar/Java is part of a collective operation that ends after all data transfers are completed. Hence, the order does not matter. More advanced schemes may be used for non-collective communication. In [7], for example, an approach is described that monitors the system and dynamically predicts the importance of individual data transfers and assigns communication priorities accordingly.

In the way it is used, ENSEMBLE resembles irregular communication libraries such as CHAOS [11]. In both cases a list of communications that must be done is prepared, and upon execution of this list the library is free to choose the optimal approach for the particular communication pattern. However, ENSEMBLE was designed to implement communication of regularly strided blocks efficiently, not irregularly distributed single elements. Moreover, libraries such as CHAOS are usually implemented using standard communication libraries, and therefore do not exploit the possibility of overlapping DMA and message gather and scatter to the extent that ENSEMBLE does.

The embedded multi-processor systems that ENSEMBLE targets bear great resemblance with modern SMPs, which include programmable network interfaces with DMA functionality. Making DMA available to the user in the context of general-purpose SMPs is difficult because of the kernel barrier. Several techniques have been proposed to deal with this issue [3, 9]. Since embedded processors do not support multi-tasking, no true kernel is needed making the design of ENSEMBLE much simpler.

6. CONCLUSIONS

Embedded systems supporting multi-media must be flexible to support different applications and high-performance to support the signal processing involved. An embedded multi-processor system consisting of a general-purpose CPU and a number of dedicated DSPs connected by a shared bus is a suitable hardware architecture. To ease application development for embedded multi-processor systems we have developed a compiler (for Spar/Java) and communication layer (ENSEMBLE) that automatically take care of difficult and error-prone tasks like processor synchronization and data transport between heterogeneous processors.

The interface between the Spar/Java compiler and ENSEMBLE is designed such that the hardware capabilities of embedded systems can be exploited. In particular, ENSEMBLE exploits DMA engines to overlap message aggregation (computation) and communication. Spar/Java supports the SPMD computational model and generates rendez-vous style communication with explicit send and receive primitives. In contrast to traditional communication libraries, ENSEMBLE operates on n-dimensional data descriptors that can be used to specify often-occurring data access patterns in n-dimensional arrays. This allows ENSEMBLE to setup a three-stage pack-transfer-unpack pipeline, effectively overlapping message aggregation and DMA transfers. If source and/or destination elements are laid out contiguously in memory, ENSEMBLE (partly) slaps the pipeline to bypass intermediate buffers (zero-copy transfer) increasing throughput rates.

We implemented Spar/Java and ENSEMBLE on an embedded multi-processor system consisting of an Athlon host processor and three THMedia TM1000 multimedia processors. Performance measurements show that ENSEMBLE adds little overhead to the raw DMA speed. When supporting Spar/Java, the actual communication speed largely depends on the layout of the data elements involved; the pack and unpack routines are sensitive to how well the cache handles strided accesses. We determined that overlapping message aggregation and communication increases performance up to 30% for peer-to-peer communication, and up to 34% for all-to-all communication. We anticipate an even larger benefit when the Spar/Java compiler will implement shrinking to store distributed arrays compactly.
Acknowledgements

The design and implementation of ENSEMBLE was carried out within the JOSES project funded by Espirit (LTR Project #28198). We thank the anonymous reviewers for their comments on the draft version of this paper.

7. REFERENCES