6 Impact on New Programming Languages

The versatility and flexibility of Active Messages come fully into play in two implementations of new parallel programming languages presented in this Chapter. The implementations of Split-C and of Id90 on the CM-5 integrate Active Message communication into the compilation process and demonstrate that Active Messages is a natural compilation target which enables new optimizations in the compilation process to take communication into account and which offer performance at the hardware's limit.

The Split-C language is a simple parallel extension to C and offers a structured global address space with a set of novel split-phase memory access primitives. The discussion of Split-C in Section 6.1 focuses on the role of Active Messages in the language implementation. The flexibility of Active Messages allows Split-C's remote memory access primitives to be implemented with a set of simple handlers which allow multiple outstanding remote memory references and use simple counters to detect their completion. This set of handlers in fact forms a run-time library, called libsplit-c, which is used by the compiler to implement the language constructs. The libsplit-c run-time substrate could be used in other language systems and illustrates the layering model proposed in Subsection 2.1.2. The discussion of Split-C also complements the discussion of traditional communication architectures in Chapter 5. It shows the benefit of decoupling the construction of a global address space from the communication required to access remote locations (two notions which are combined in shared memory) which allows Split-C to offer a structured address space with novel access operations. The discussion also shows the performance advantages that a global address space offers over the disjoint local address spaces of message passing: Split-C's remote memory accesses can transfer bulk data more efficiently than message passing because no memory allocation for message queueing is necessary, no tag matching is involved, and processors do not need to block, waiting for each other.

The presentation of the Id90/TAM compilation system extends the discussion of message driven architectures in Section 5.2 to show that Active Messages allows the languages for which these architectures were originally developed to be implemented efficiently on conventional multiprocessors—in this case the CM-5. The Threaded Abstract Machine (TAM) was developed specifically to target Active Messages and uses the fact that communication and scheduling are exposed to the compiler (unlike in message driven architectures) to apply powerful optimizations reducing the frequency and cost of dynamic scheduling. TAM also pays careful attention to the storage hierarchy of conventional microprocessors and achieves computational performance superior to specialized message driven hardware. The discussion illustrates how Active Messages facilitates the tight coupling between computation and communication that is required for the efficient implementation of such dynamic parallel languages. The Id90 compiler generates a custom Active Message handler for every type of message used in a program. Each handler stores the message data directly into a variable in the proper activation frame and enables the appropriate computation. While the code in all these handlers is very similar, the data movement and especially the synchronization in each handler is heavily specialized to each situation.

6.1 Split-C

Split-C is a parallel extension of the C programming language primarily intended for distributed memory multiprocessors that support an Active Messages communication architecture. It is designed around two objectives. The first is to capture certain useful elements of shared memory, message passing, and data parallel programming in a familiar context, while eliminating the primary deficiencies of each paradigm. The second is to provide efficient access to the underlying machine with no surprises. (This is similar to the original motivation of C—to provide a direct and obvious mapping from high-level programming constructs to low-level machine instructions.) Split-C does not try to obscure the inherent performance characteristics of the machine through sophisticated transformations. This combination of generality and transparency of the language give the algorithm or library designer a concrete optimization target.

The role of this section is four-fold:

1. it illustrates the layering model described in Subsection 2.1.2 in the context of a high-level language, in particular, it describes the run-time substrate used to implement the communication model underlying Split-C,
2. it uses the versatility of Active Messages to develop novel remote memory access operators that offer efficient access to global data structures and that can be extended easily,
3. it shows the role of Active Messages in implementing shared memory like communication models, in particular, the separation of the address translation and the actual communication aspects of shared memory, and
4. it demonstrates that a global address space (combined with the flexibility of Active Message handlers) allows large data blocks to be transferred more efficiently than with regular message passing.

The current Split-C language system consists of three distinct parts: the Split-C compiler, the run-time library *libsplit-c* and Active Messages. The compiler translates the C language extensions into calls to the library which uses Active Messages to implement the structured global address space fundamental to Split-C. This section concentrates on the language developer's perspective of Split-C and describes the impact of Active Messages on the language development and on its implementation. The Split-C language itself is not covered in detail here; Subsection 6.1.1 summarizes the language features relevant to this discussion and the reader is referred to the language tutorial [CDG+] and to "Parallel Programming in Split-C" [CDG+93] for an in-depth discussion of the language itself.

The heart of Split-C lies in the run-time library that implements the two-dimensional global address space and provides a number of innovative accessors within that address space. Subsection 6.1.2 describes libsplit-c as providing an abstract multiprocessor, in some sense a NUMA machine—static in that each processor has one thread of control and global in the sense that memory can be accessed globally. Active Messages play and important role in libsplit-c: the novel forms of read and write to remote memory locations that libsplit-c provides are implemented using custom Active Message handlers to perform the access at the remote end and to synchronize the responses with the ongoing computation.

The Split-C language itself is a thin veneer over the run-time library as is implemented as part of the compiler front-end. The Split-C syntax for global pointers, global memory accesses and split-phase assignments is expanded into inline function calls to the underlying library. Subsection 6.1.3 summarizes performance results obtained with Split-C and contrasts libsplit-c to traditional shared memory, message passing, and data parallel programming models. Split-C offers a global address space similar to shared memory hardware at directly comparable performance, but with a clear cost model for the programmer and without the complexity of the hardware implementations.

6.1.1 Split-C language summary

The Split-C language extensions address two fundamental concerns introduced by large-scale multiprocessors: there are multiple active threads of control, one on each processor, and there is a new level of the storage hierarchy which involves access to remote memory modules via an interconnection net-

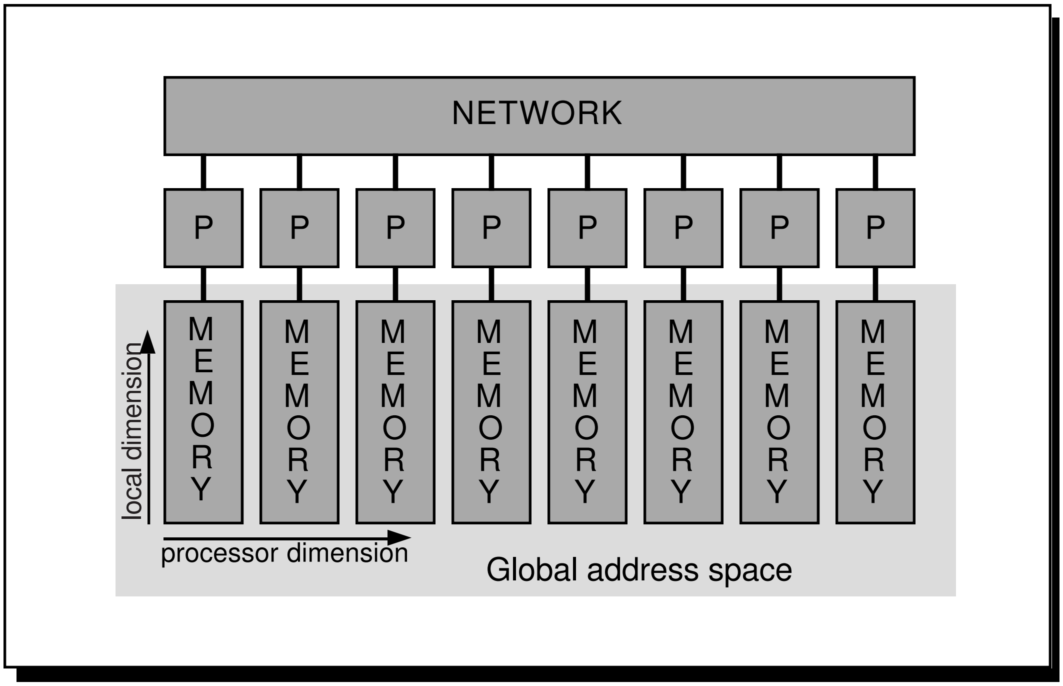


Figure 6-1: 2D global address space underlying Split-C.

work. Split-C assumes that the programmer must think about these issues in designing effective data structures and algorithms and desires a reasonable means of expressing the results of the design effort. The underlying machine model is a collection of processors operating in a common global address space, which is expected to be implemented as a physically distributed collection of memories. As illustrated in Figure 6-1, the global address space is two-dimensional from the viewpoint of address arithmetic on global data structures and from a performance viewpoint, in that each processor has efficient access to a portion of the address space—the local portion of the global space. Split-C provides access to global objects in a manner that reflects the access characteristics of the inter-processor level of the storage hierarchy.

Split-C provides a range of access methods to the global address space, but encourages a "mostly local" programming style. It is anticipated that different architectures will provide varying degrees of support for direct access to remote memory. Finally, it is expected that global objects will often be shared and this requires an added degree of control in how they are accessed. While Split-C requires the programmer to manage the synchronization issues resulting from split-phase accesses into the global address space explicitly, other, higher-level, language implementations are planned and expected to use more sophisticated compiler technology to relieve the programmer from many of the low-level concerns.

The Split-C language consists of ANSI-C augmented with a small set of extensions described in the following paragraphs. Each extension corresponds directly to a set of primitives provided by the run-time library which is described in Subsection 6.1.2 and evaluated in Subsection 6.1.3.

Multiple persistent threads

A Split-C program is parallel *ab initio*. From program begin to program end there are `PROCS` threads of control within the same program image. Each thread has a unique number given by a special variable `MYPROC` that ranges from 0 to `PROCS`–1. Generally, in Split-C the term processor refers to the thread of control on that processor. A variety of convenient parallel control structures can be built on this substrate and several are provided as C preprocessor (cpp) macros, but the basic language definition does not prescribe dynamic thread manipulation or task scheduling. A small family of global synchronization operations are provided to coordinate the entire collection of threads, e.g., `barrier`. No specific programming paradigm, such as data parallel, data driven, or message passing, is imposed by the language. However, these programming paradigms can be supported as a matter of convention.

2-D global address space

One of the fundamental notions in Split-C is the two-dimensional address space which allows every processor to reference all memory in the machine. Figure 6-1 shows the canonical Split-C view of a distributed memory multiprocessor in which the memories on all nodes are seen as one large address space indexed by processor number and by local addresses.

Split-C distinguishes between normal pointers referencing objects local to the processor and global pointers into the 2-D address space in order to expose their differing properties to the programmer. A global pointer refers to an arbitrary object of the associated type anywhere in the machine. An object referenced by a global pointer is entirely owned by a single processor. A new type qualifier `global` is introduced to define a pointer as meaningful to all processors. Global pointers can be dereferenced in the same manner as standard C pointers, although the time to dereference a global pointer is considerably greater than that for a local pointer. The language provides support for allocating global objects, constructing global pointers from local counterparts, and destructuring global pointers into its processor and local components.

A pointer in C references a particular object, but also defines a sequence of objects that can be referenced by arithmetic operations on the pointer. In Split-C the sequence of objects referenced by a standard pointer are entirely local to the processor. Address arithmetic on a global pointer has the same meaning as arithmetic on a standard pointer by the processor that owns the object. Hence, all the objects referenced relative to a global pointer are associated with one processor.

A second form of global pointer is provided which defines a sequence of objects that are distributed or *spread* across the processors. The keyword `spread` is used as the qualifier to declare this form of global pointer[1]. Consecutive objects referenced by a spread pointer are "wrapped" in a helical fashion through the global address space with the processor dimension varying fastest. Each object is entirely owned by a single processor, but the consecutive element, (i.e., that referenced by ++) is on the next processor. The arithmetic on both, global and spread pointers is illustrated in Figure 6-2.

Split-phase assignment

Split-phase assignments provide support to overlap communication and computation in Split-C. With a new assignment operator, `:=`, the initiation of a global access can be split from the completion of the access. This allows the processor to initiate remote accesses, do some local computation, and then wait for the outstanding accesses to complete using a `sync` statement. In contrast, standard assignments stall the issuing processor until the assignment is complete to guarantee that reads and writes occur in program order. However, there are restrictions on the use of split assignments: the split assignment operator specifies either to *get* the contents of a single global reference into a local one or to *put* the contents of a local reference into a global one. This is in contrast to the standard assignment operator which allows multiple reads and one write.

The `:=` assignment initiates a transfer, but does not wait for its completion. A `sync` operation joins the preceding split assignments with the thread of control, i.e., it blocks until all outstanding gets and puts complete. A local variable assigned by a get (similarly, a global variable assigned by a put) is guaranteed to have its new value only after the following sync statement. The value of the variable prior to the sync is not defined. Variables appearing in split assignments should not be modified (either directly or through aliases) between the assignment and the following sync, and variables on the left hand side should not be read during that time. The order in which puts take effect is only constrained by sync boundaries; between those boundaries the puts may be reordered. No limit is placed on the number of outstanding assignments.

---

1. The term global pointer is used when discussing the properties common to both, `global` and `spread` pointers.

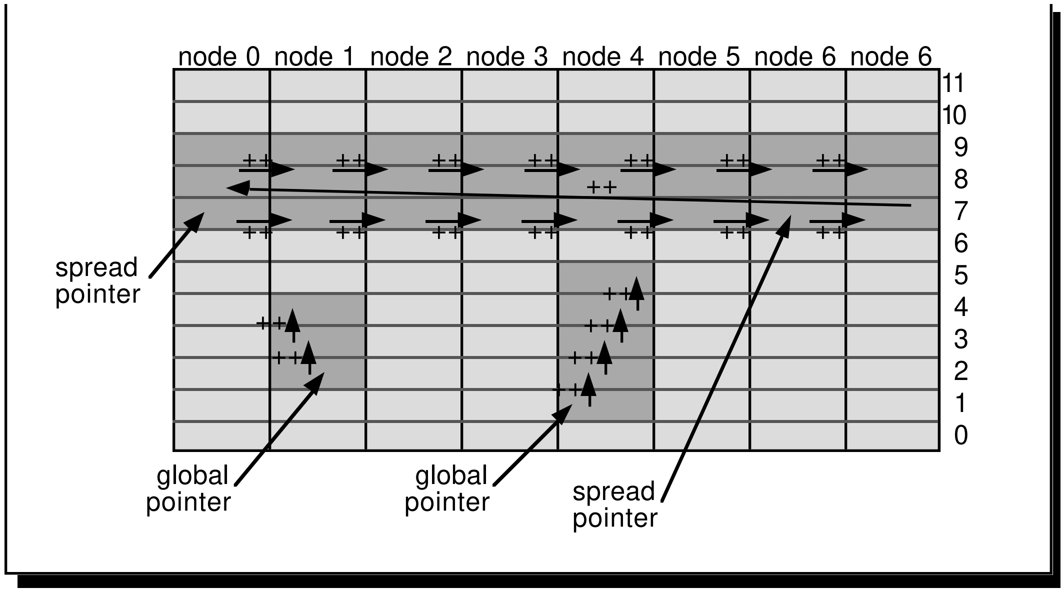


Figure 6-2: Split-C pointer arithmetic.

Signaling assignment

A weaker form of assignment, called *store* and denoted `:-`, is provided to allow efficient one-way data pushing and global operations. Store updates a global location, but does not provide any acknowledgment of its completion to the issuing processor. Completion of a collection of such stores is detected globally using `all_store_sync`, executed by all processors. For global data rearrangement, in which all processors are cooperating to move data, a set of stores by the processors are followed by an `all_store_sync`. In addition, the recipient of store can determine if certain number of stores to it have completed using `store_sync`, which takes the expected number of stores and waits until they have completed. This is useful for data driven execution with predictable communication patterns (e.g., not requiring arbitrary dynamic scheduling of computation).

Bulk assignment

Transfers of complete objects are supported through the assignment operators and library routines. The library operations allow for bulk transfers, which reflect the view that, in managing a storage hierarchy, the unit of transfer should increase with the access time. Moreover, bulk transfers enhance the utility of split-phase operations. A single word get is essentially a binding prefetch. The ability to prefetch an entire object or block often allows the prefetch operation to be moved out of the inner loop and increases the distance between the time where the get is issued and the time where the result is needed. The assignment and split-assignment operators transfer arbitrary data types or structs, as with the standard C assignment. However, because C does not provide operators for copying entire arrays, library functions must be used for this purpose.

6.1.2 Libsplit-c implementation using CM-5 Active Messages

Originally libsplit-c was developed as a true library to be called from the user program. The Split-C language was added later after the important concepts had settled and it became clear that a simple veneer could greatly improve the ease of use without compromising the performance. Due to the inevitable constraints of language syntax and semantics the Split-C language does not fully exploit all capabilities of the library whose functionality is also expected to grow as other languages use it as a run-time substrate. As illustrated in Figure 6-3, libsplit-c is not wedded to Active Messages and implementation on

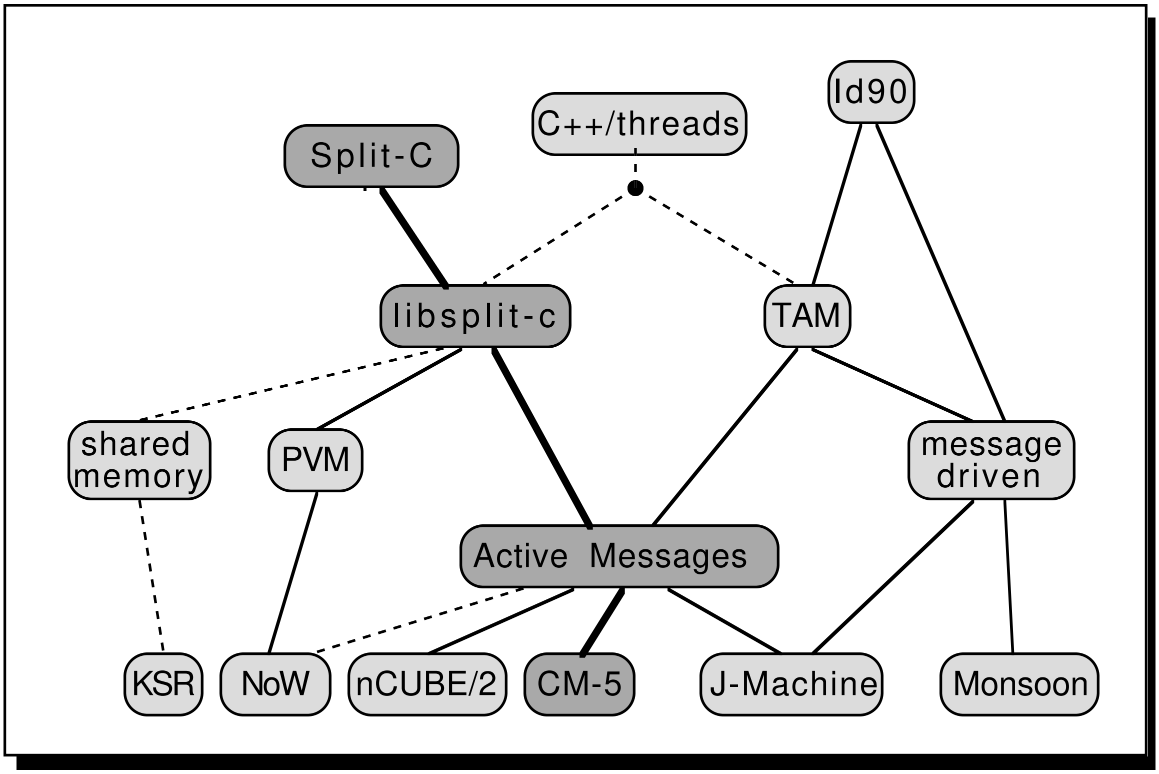


Figure 6-3: Software layers related to Split-C.

structures involving dynamic scheduling and requiring a closer coupling of computation and communication will require more of the power of Active Messages, however, and will be less amenable to implementation on other substrates.

The Split-C run-time substrate is structured around the definition of global pointers forming the Split-C two-dimensional address space. A number of functions provide remote memory accessors and a set of collective operations such as reductions and scans. The implementation of each one of these operations uses custom Active Message handlers. In the case of Split-C, the compiler translates the language extensions into inlined calls to the run-time substrate as if they were macro-assembly instructions.

Multiple persistent threads

At program start-up, libsplit-c broadcasts the conventional C program arguments (`argv`) to all processors and starts the main function, `splitc_main`, on each one. The broadcast is implemented using Active Messages to propagate the arguments along a static broadcast tree.

2-D global address space

Libsplit-c represents global addresses as 64-bit tuples with a processor and a local-address part:

- addresses are in the range $<0\ldots P-1, 0\ldots N-1>$, and
- address $<p, l>$ $(0 \le p < P$ and $0 \le l < N)$ refers to location l on node p, where P is the number of processors and N is the highest memory address.

The definition of global pointers is shown in Figure 6-6 together with functions to construct and destruct global pointers. Note that special care must be taken to ensure that a unique representation of global NULL pointers is used (line 8).

```
# Global and spread pointer definition
typedef struct {
  int   proc;
  void  *addr;
} gptr;
# Constructing a global pointer
inline gptr toGlobal(int proc, void *addr)
{
  gptr  g;
  g.proc = addr == 0 ? 0 : proc;  unique representation of NULL pointer
  g.addr = addr;
  return g;
}
# Taking a global pointer apart
inline int toProc(gptr g)
{
  return g.proc;
}
inline void *toLocal(gptr g)
{
  return g.addr;
}
```

Figure 6-4: Representation of global pointers in libsplit-c's 2-D address space.

In the 2-D address space a data structure is kept local to a given processor by indexing into it using arithmetic on the local part of the global address such that, for example, the i-th element of a localized integer array allocated at address $<a, p>$ is at $<p, a + i \cdot \mathrm{sizeof(int)}>$. To spread a data structure across nodes arithmetic is performed on both the node and the local part. The i-th element of an integer array allocated at address a and spread element-wise across all nodes would be at $<i \bmod P, a + (i \bmod P)\ \mathrm{sizeof(int)}>$.

The use of two forms of data structure layout requires support from the memory allocator which allocates spread structures in a special memory segment which is kept aligned on all processors. Local and global data structures are allocated in the normal heap which grows and shrinks independently on all processors. Figure 6-5 illustrates the use of memory segments to implement the Split-C global address space.

Pointer arithmetic

A number of pointer arithmetic operators are provided to support the global and spread forms of data layout. As illustrated in Figure 6-6, global pointer arithmetic is local to a given node and defined such that

$$<p, l> + i = <p, l + iS> \quad ,$$

where S is the size of the pointer base type. Spread pointer arithmetic wraps across all nodes, i.e., it is defined such that

$$<p, l> + i = <p + i \bmod P, l + ((p + i) \operatorname{div} P)\ S>.$$

When a global pointer is dereferenced, a simple check, shown in Figure 6-6, determines whether the access is local or whether it is remote. Local accesses simply dereference the address part of the global pointer. This means that using global pointers to point to local objects is only slightly more expensive

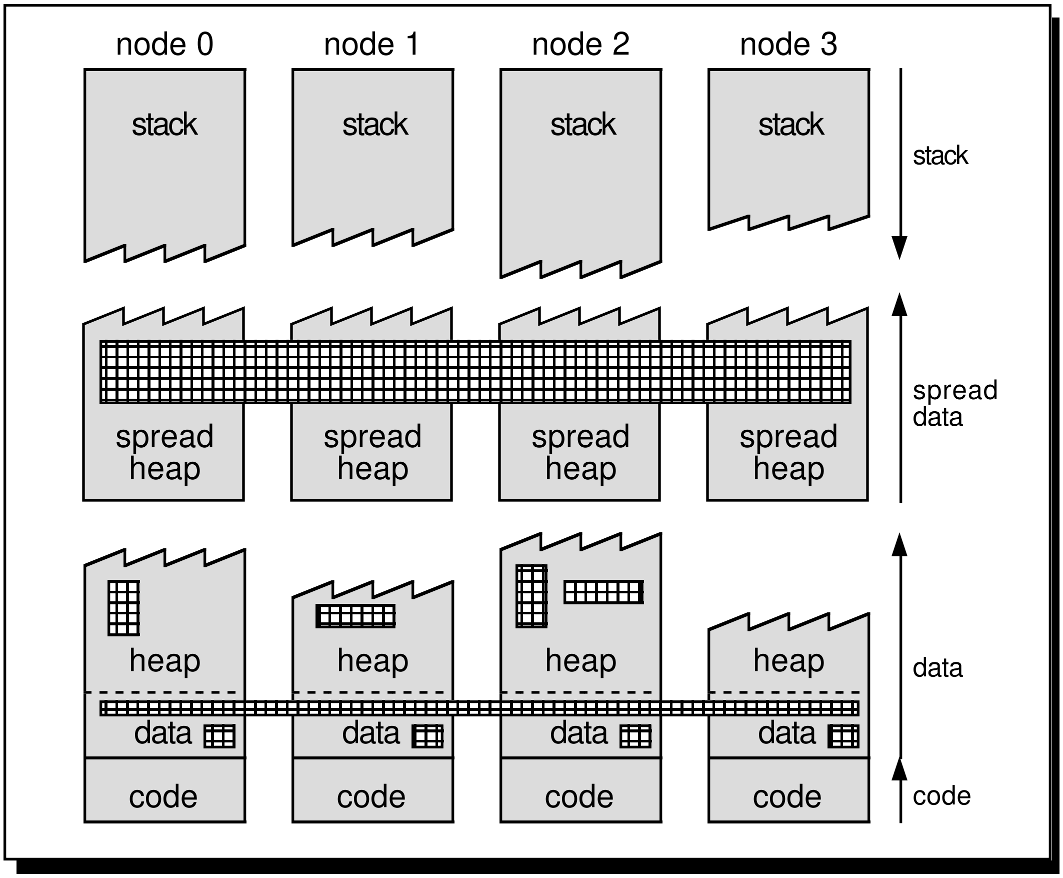


Figure 6-5: Usage of memory segments in Split-C.

The use of two forms of data structure layout requires support from the memory allocator. Local and global data structures are allocated in the normal heap which grows and shrinks independently on all processors. Allocating spread structures, however, requires coordination among processors as a spread structure requires memory on each node at the same local addresses. The spread allocator uses a memory segment which remains synchronized among all processors; growing or shrinking it is a global operation. On each processor the virtual-memory mapping maintains four memory segments which each start at the same local address on all nodes. The code segment is identical on all nodes as is typical for the SPMD programming model of Split-C. The data and stack segments are distinct in that they grow and shrink independently in response to local operations. The spread data segment, however, is special: it has the same size on all nodes and growing or shrinking it requires coordination of all processors. As illustrated, the initialized data region of the heap segment and the parallel segment are used to allocate global data structures (statically, respectively, dynamically).

than using regular pointers[2]. On 32-bit machines global pointers are twice the size of local pointers, though, which bears an indirect cost, increasing the register pressure and requiring double-word loads and stores.

Remote memory access

The two primitives for remote memory access are split-phase puts and gets with programmer-managed synchronization counters. The implementation is shown in Figure 6-7[3] and uses Active Message request and reply messages for the round-trip protocol. A counter on the initiating node is used to keep

[2] The current CM-5 implementation polls every time a global pointer is dereferenced locally. This raises the cost of accesses to local variables via global pointers substantially.

[3] The figure shows the implementation of the remote access of integers. The library provides a full complement for all data types.

```
# Global pointer arithmetic
inline gptr addToGlobal(gptr p, int i, int elem_sz)
{
  p.addr += i * elem_sz;
  return p;
}
# Spread pointer arithmetic
inline sptr addToSpread(gptr p, int i, int elem_sz)
{
# ifndef PROCS_POWER_OF_TWO        defined if PROCS is power of two
    p.proc = (p.proc + i) % PROCS;
    p.addr = p.addr + ((p.proc + i) / PROCS) * elem_sz;
# else
    p.proc = (p.proc + i) & (PROCS-1); LOG_PROCS = log2(PROCS)
    p.addr = p.addr + ((p.proc + i) >> LOG_PROCS);
# endif
  return p;
}
```

```
# Global pointer dereference
int *global gp;
...
x = *gp;
...
```

```
gptr gp;
...
x = (gp.proc == MYPROC)
      ? *(int *)gp.addr
      : read_int(gp);
...
```

Figure 6-6: Address arithmetic in the two-dimensional address space.

track of the number of outstanding requests; it is incremented before issuing the request Active Message and decremented by the Active Message reply handler. The handler at the remote node simply fetches (respectively stores, in the case of a write or put) the value and sends the appropriate reply back. The outstanding request counters allow a simple form of synchronization between the communication and the issuing computation. Synchronization operations provided in the library take a counter as argument and wait until it reaches zero.

As Figure 6-7 shows, the implementation of get with Active Messages is straightforward. The `getCtr` function increments the counter of outstanding requests and formats the request Active Message. The compiler typically inlines this function with the result that register allocation can ensure that the values to be sent in the message are produced in the correct registers such that they are transferred directly into the network interface (in the `AM` function) without intermediate copy. The request handler consists of a single statement sending the reply back. Again, the data value fetched from the remote memory is loaded into the correct register and never copied before it is stored into the network interface. The reply handler disposes of the data as appropriate and synchronizes the communication with the computation by decrementing a counter. The implementation of put is similar to that of get with the difference that the data value to be stored is carried in the request message and that only an acknowledgment is returned.

Split-C implements blocking remote reads and writes using these primitives. The functions implementing read and write declare a local synchronization counter and pass that to the split-phase put or get, similar to the blocking fetch&add shown in Figure 4-18. The implementation of Split-C's puts and gets simply uses one statically allocated set of synchronization counters per processor.

The full libsplit-c flexibility of using multiple synchronization counters for simultaneously outstanding remote accesses is not accessible from Split-C (except by calling the library primitives directly). One reason for restricting the language is to be able to use prefetch buffers or counters in the network inter-

Get primitive with explicit counter

```
 # Synchronization
typedef int Ctr;
void syncCtr(Ctr *ctr)
{ while(*ctr) ; }
 # Issue get (assume local/remote check done)
void getCtr
  (gptr g, int *l, Ctr *ctr)
{
  (*ctr)++;
  AM(g.proc, getCtr_h,
    MYPROC, l, ctr, g.addr);
}
 # Handle get request
void getCtr_h
  (int p, int *l, Ctr *c,
   int *a)
{
  AM_reply(p, getCtr_rh,
    l, c, *a);
}
 # Handle get reply
void getCtr_rh
  (int *l, Ctr *ctr, int v)
{
  *l = v;
  (*ctr)--;
}
```

Split-C get with implicit counter

```
 # Synchronization
int numSplit=0;
void sync(void)
{ syncCtr(&numSplit); }
 # Split-C get
void get(gptr g, int *l)
{
  getCtr(g, l, &numSplit);
}
```

Split-C read

```
 # Split-C read
int read(gptr g)
{
  Ctr *ctr;
  int val;
  getCtr(g, val, &ctr);
  syncCtr(&ctr);
  return val;
}
```

Figure 6-7: Split-phase remote memory accessors.

face on machines which provide this type of hardware support. A second reason is that syntax for associating a synchronization counter with each assignment statement was deemed to cumbersome. Associating different synchronization counters statically with assignment statement is mainly useful when building communication libraries in order to avoid interference between the primitive and the enclosing scope. The use of multiple counters is also useful with a task model and allows split-phase accesses of one task to complete while the next one starts executing. This might not be of great importance for single-word accesses, but can become significant when bulk transfers are used.

Signaling store

For stores, the library also provides a primitive with explicit synchronization counters. The store counters keep track of the number of bytes sent and the number received, rather than counting outstanding operations. This supports two forms of synchronization: waiting until a number of bytes have been received on a given processor and waiting until all stores have completed on all processors. Figure 6-8 shows the implementation of the synchronization operations, of the store primitive, and of the Split-C store which uses a statically allocated set of counters. The implementation is again similar to that of get but uses only one-way communication.

Using multiple counters with store has a number of interesting uses. In order to perform message passing style communication one can associate a set of counters with each data structure that is sent, or if the communication patterns highly structured, with each communication direction, e.g., north-south-east-west in 2-D grid communication. In a systolic communication mode (used, for example, in repeti-

Primitives for synchronization of stores

```
# Synchronization counters
typedef struct {
  int  send, recv;
} StCtr;

# Local synchronization
void store_syncCtr
   (StCtr *ctr, int num)
{
  while(ctr->recv < num) ;
  ctr->recv -= num;
  ctr->send -= num;
}

# Global synchronization
void all_store_syncCtr
   (StCtr *ctr)
{
  int d, all_d;
  barrier();
  d = 0;
  do {
    d += ctr->send-ctr->recv;
    ctr->recv = ctr->send = 0;
    all_d = reduce(d);
  } while(all_d != 0);
}
```

Primitives for store with explicit counter

```
# Issue store
void storeCtr
   (gptr g, int val,
    StCtr *ctr)
{
  AM(g.proc, store_h,
     g.addr, val, ctr);
  ctr->send += sizeof(int);
}

# Handle store
void store_h
   (int *a, int val, StCtr *c)
{
  *a = val;
  c->recv += sizeof(int);
}
```

Split-C store with implicit counter

```
# Synchronization counter
StCtr stores = {0,0};

# Issue store
void store(gptr g, int val)
{
  storeCtr(g, val, stores);
}

# Synchronize
void store_sync(int num)
{
  store_syncCtr(num, stores);
}
void all_store_sync(void)
{
  all_store_syncCtr(stores);
}
```

Figure 6-8: Signalling stores.

tive broadcasts, scans, and reductions) multiple sets of counters can be used in order to separate one wave from the next.

Bulk Assignment

The bulk get, put, and store allow the transfer of large blocks of data with high efficiency. Because all the operations specify both a source and a destination memory address no storage allocation needs to occur at the destination. Furthermore, because the synchronization is explicit and the semantics of the operations prescribe that data must not be changed before the operation completes the data can be sent directly from the source data structure and received directly into the destination memory locations. No intermediate copies need to be made in order to hide the fact that transmission is not instantaneous.

Due to the fact that typical Active Message implementations limit the message size challenging implementation issues arise for the transfer of blocks larger than the maximal message. For example, the CM-5 implementation of bulk get is straightforward, but bulk put is complicated by the potential for message reordering in the network which makes accounting of all messages forming a single block transfer difficult. In the case of get, a single request message is sent to the remote processor with the remote address, block length, and return address. The request handler sends the data back using multiple Active Messages which the local reply handler counts. When the last message arrives its reply handler decrements the number of outstanding gets. In the case of a block put, however, detecting the arrival of all messages is more challenging. A simple solution is to acknowledge every put message and thereby count the messages at the originating node in the same manner as for gets. On the CM-5, the additional cost of the acknowledgment for every message is significant and therefore the actual implementation starts the transfer with a round-trip message to allocate a synchronization variable at the remote node. The remote handler can thereby detect the arrival of all messages and send back a single acknowledgment at the end. Note that block puts of intermediate size, e.g., larger than a single message but too small to amortize the round-trip start-up, use the simple scheme.

The difficulty in detecting the arrival of multiple messages on the CM-5 leads to the situation where block puts (and thus also block writes) are more expensive than block reads. Note that this is an artifact of the combination of small fixed-size messages and the possibility for message reordering. On the nCUBE/2 block writes have the same cost as block reads because messages arrive in order and the remote node can simply send the acknowledgment when the last message arrives. On both machines it is important that Active Messages exposes the hardware constraints to allow the run-time substrate, libsplit-c in this case, to implement the best possible communication protocol. This means that the library implementation is not directly portable from one machine to the other, but the functionality provided by the library and seen by the user is portable and the general structure of the library is portable as well.

Collective operations

Libsplit-c implements a number of collective communication operations which involve all processors. These include barrier, broadcasts, reductions, and scans. The reductions and scans provided support the usual set of integer and floating point arithmetic and logic operations but can be easily extended to arbitrary function.

The basic structure of all collective operations is identical. All processors are organized into a binary tree rooted at processor 0 and values are propagated up and/or down the tree. The collective operations are implemented in terms of stores. At each node, when propagating up the tree the processor waits for two stores from the child nodes, combines the two values, and stores the result onto the parent node. The down-propagation is similar with each node waiting for the store from the parent and storing onto the two child nodes.

6.1.3 Split-C evaluation

This subsection evaluates Split-C from an absolute performance point of view and in comparison to other programming models. § 6.1.3.1 presents detailed Split-C micro-benchmark measurements, compares them with the costs of the underlying architecture, and summarizes the performance of a few larger Split-C programs. § 6.1.3.2 contrasts the Split-C programming language with shared memory to show the advantages and disadvantages of the explicit 2D address space. § 6.1.3.3 relates Split-C's store to messages passing and discusses the performance advantages gained by the global address space which allows block transfers to occur without any intermediate buffering.

6.1.3.1 Performance

Table 6-1 shows micro-benchmark performance measurements for individual Split-C remote access operations. The table compares the measurements obtained for the Split-C operations with those of the underlying Active Message primitives (as measured in Subsection 4.3.4). The results show that, in good C tradition, the cost of the Split-C operations directly reflects the possibilities of the underlying communication architecture.

The use of Split-C in a message passing style is nicely illustrated in a study of fast parallel sorting algorithms [CDMS93] which implements four sorting algorithms in Split-C and carefully measures their performance on a CM-5. The goal of the study is to develop sorting algorithms that sort a billion large

operation	transfer size	overhead [µs] Split-C	overhead [µs] Active Messages	latency [µs] Split-C	latency [µs] Active Messages	bandwidth [Mb/s] Split-C	bandwidth [Mb/s] Active Messages
read	32-bit	12.2	11.2	12.2	11.2	0.3	0.4
	64-bit	12.2	11.2	12.2	11.2	0.6	0.7
	bulk	20.3	18.2	20.3	18.2	10.1	10.0
write	32-bit	11.8	11.2	11.8	11.2	0.3	0.4
	64-bit	11.8	11.2	11.8	11.2	0.6	0.7
	bulk	31.2	29.4	31.2	29.4	9.1	9.3
get	32-bit	4.0	3.1	11.9	11.2	1.0	1.3
	64-bit	4.0	3.1	11.9	11.2	2.0	2.6
	bulk	11.0	9.8	20.4	18.2	9.8	10.0
put	32-bit	3.9	3.1	11.9	11.2	1.0	1.3
	64-bit	3.9	3.1	11.9	11.2	2.0	2.6
	bulk	22.8	21.2	31.4	29.4	9.8	9.3
store	32-bit	2.2	1.6	7.6	5.6	1.0	2.5
	64-bit	2.2	1.6	7.6	5.6	1.0	5.0
	bulk	20.7	19.5	24.2	23.5	10.0	10.0
sync		0.6	0.7	n/a	n/a	n/a	n/a

Table 6-1. Split-C micro-benchmark performance results

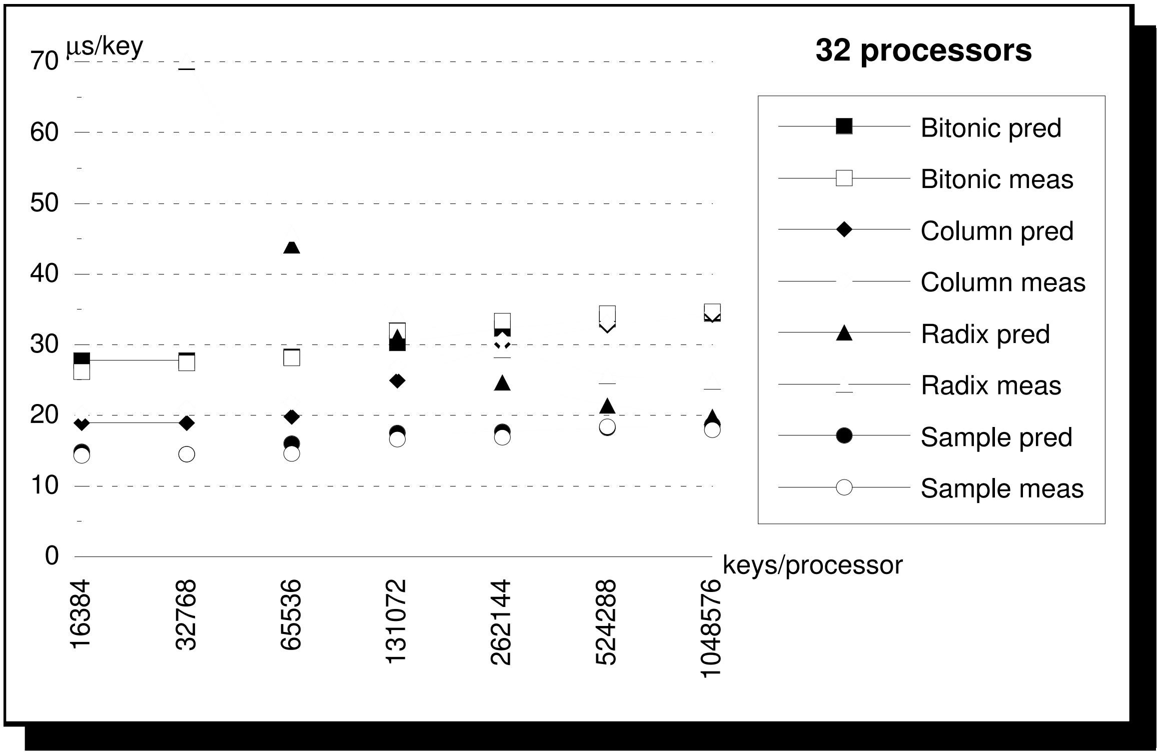
The measurements for the various Split-C primitives (shown in the "Split-C" columns) is compared to a prediction obtained by summing the costs of the underlying Active Messages (shown in the "Active Messages" columns).

Overhead measures the time spent by the processor initiating the remote access, latency measures the time from the access initiation to its acknowledgment (assuming the access is to a neighbor node, note that this is not LogP's latency L), and bandwidth measures the peak data transfer rate achievable.

keys on a thousand processors in less than a minute. More precisely, the sorting algorithms are designed to sort a billion 31-bit keys of arbitrary distribution. The input data set is spread evenly in a blocked layout across the processors and the output is sorted such that each processor ends up with 1/P-th of the keys. The algorithms studied are bitonic sort, column sort, radix sort, and sample sort. The performance of each algorithm is predicted using the LogP model of parallel computation and the experimental results are carefully matched to the model to ascertain that the behavior of the implementation is well understood.

The structure of all four sorting algorithms is very similar and consists of an alternation of local computation or sorting phases and global communication phases in which the keys are permuted. The implementations exclusively use Split-C stores for the communication as all data can be "pushed" and store offers a low overhead in this case. Figure 6-9 compares the LogP prediction for the four sorting algorithms with the results. The close agreement between the two demonstrates that the performance obtained in the micro-benchmarks—on which the LogP parameter values are based—translates to the real program behavior.

In addition to observing the absolute performance achieved by the Split-C implementations, a report on a similar radix sort implementation on the CM-5 [TS91] allows a comparison of Split-C with a hand-tuned "assembly language" version produced at Thinking Machines Corp. (TMC). The TMC version was written in C and uses assembly language routines to access the network interface which were hand coded specifically for the radix sort. Table 6-2 shows the timings for large runs of the two implementations. It is surprising to see that in the communication-intensive permutation phase the



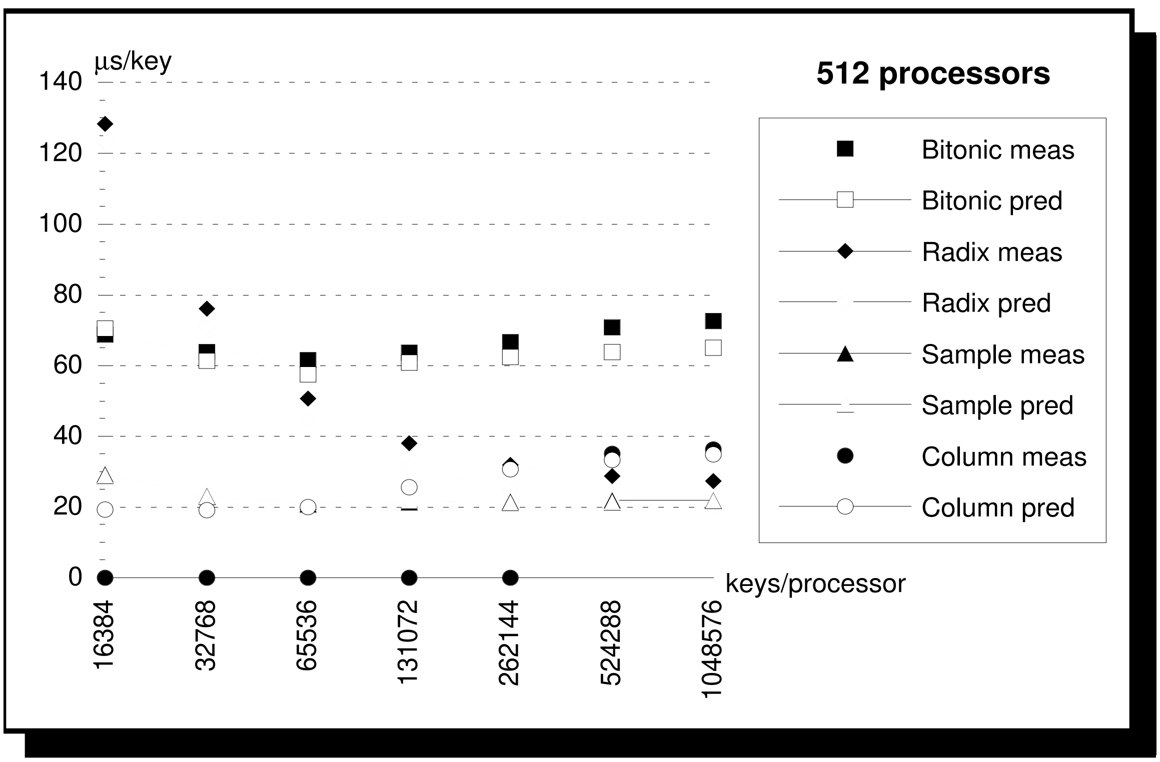


Figure 6-9: Predicted and measured performance for four Split-C sorting algorithms

Four fast parallel sorting algorithms were implemented in Split-C: bitonic sort, sample sort, radix sort, and column sort [CDMS93]. The performance of each algorithm was predicted using the LogP model and measured on a CM-5 using from 32 to 512 processors.

Radix sort phase	µs/(key/proc)	
	Split-C†	TMC‡
histogramming	2.41	3.11
scanning	1.61	1.21
permuting	23.26	13.59
total	27.28	17.91

†. From [CDMS93]
‡. From [TS91]

Table 6-2. Comparison of Split-C and hand-coded radix sort on CM-5

The benchmarks consists of sorting one billion uniformly distributed 31-bit keys on a 1024-processor CM-5. The table shows the time each processor spends per key in the sort. The Split-C run sorted 512M 31-bit keys on 512 processors while Thinking Machines' run sorted 1G 31-bit keys on 1024 processors. Experiments show that radix sort scales almost linearly at this size problem.

The implementations use a 16-bit radix and perform two passes over the keys. Each pass consists of three phases: (i) a local histogram of key values is built on each processor, (ii) a global histogram is formed by performing a prefix sum for each histogram bucket across all processors, and (iii) all keys are permuted to their new position based on the global histogram.

Split-C version performs only 70% worse than TMC's version given that the Split-C version distributes the keys using a simple loop containing a `:-` store for every key whereas TMC's version includes a special loop which sets-up registers to address the network interface once, tightly integrates sending and receiving, and uses a simpler message format consisting of the key destination address and the key value.

6.1.3.2 Shared memory style programming in Split-C

At first, it may seem that Split-C is a software implementation of NUMA shared memory with a few peculiar assignment statements added. In particular, in most NUMA shared memory machines each processor has two types of memory addressing for global data as well: memory segments local to a given processor (but mapped into all processor's address spaces) have the same arithmetic as Split-C's global pointers and the address translation hardware for memory segments interleaved across all processors is similar to spread pointer indexing. Closer inspection, however, reveals that the two differ in two important aspects:

- the interleaving unit in spread pointer arithmetic is the size of the elements pointed-to, which means that each data structure element resides on a single memory module and consecutive elements reside on consecutive modules while in NUMA shared memory the interleaving unit is determined by the hardware[4] and is the same for the entire memory segment, and
- global and spread pointers can refer to the same data structures which allows the local elements of a spread data structure to be traversed easily while the memory segments in shared memory architectures are disjoint.

The syntactic separation of global pointers from local pointers is simultaneously Split-C's main advantage and main drawback over cache-coherent shared memory programming models. In Split-C, the programmer is presented with a simple explicit cost model which encourages reasoning about the location of shared data structures and making local copies of frequently accessed data. From an implementation point of view, the separation allows the address translation required for global accesses to be decoupled from the communication proper. The frequency of global pointer dereferences is low enough that the address translation can be performed in software with the advantage that the spreading of data structures across nodes can respect access patterns and element sizes. Using Active Messages for the

4. Some systems allow the interleave unit to be changed on a per segment basis [Pfi85 ACC+90], while in others it is fixed [BGWW91].

communication gives enough flexibility to implement novel access methods which allow remote accesses to be pipelined and to overlap computation. Requiring the programmer to explicitly copy remote data into a separate local variable for frequent use is clearly the main disadvantage of Split-C. The automatic renaming that occurs in the cache of a CC-NUMA architecture allows the program to reference the local copy with the same address as the remote one.

6.1.3.3 Message passing style programming in Split-C

Algorithms developed for message passing models can often be implemented in Split-C with more ease and with higher efficiency. The global address space simplifies the access to global data structures, but more importantly, the combination of the global address space with the explicit synchronization of transfers allows data to be moved more efficiently than in message passing models where a combination of message queueing, data copying, and tag matching bears significant costs.

In Split-C the global address space allows the initiator of a data transfer to specify the source and destination addresses such that the data can be moved directly from application data structures into the network and vice-versa. The novel assignment operators provide more flexible synchronization between the two processors involved than the various blocking and non-blocking versions of send&receive. Unlike shared memory, Split-C puts and gets allow the overlap of computation and communication and stores support the efficient one-way communication that is typically associated with message passing (but, as discussed in Section 5.1, that only few message passing implementations actually achieve).

For example, the sample Fortran-D code in § 5.1.5.1 can be translated to Split-C by replacing every message send by a Split-C bulk store. The Split-C version allows the compiler to specify the destination address to Active Messages with the result that the data is transferred directly into the application data structure at the destination. In addition, Split-C allows a different compilation strategy in which the recipient requests the data using gets.

A further advantage of Split-C over message passing is that the synchronization is exposed: the source data for a put is not guaranteed to be sent instantaneously (which often requires copying and message queueing), instead the semantics of put require the data to remain unchanged until the next synchronization point (e.g., the next sync statement).

6.1.4 Summary

The Split-C language is built around the libsplit-c communication substrate which abstracts the new level of the storage hierarchy represented by remote memory into a global address space with a collection of innovative communication operations. Split-phase remote memory accessors support shared memory style programming and allows syntactically transparent access to remote data structures in Split-C while exposing the data layout and remote access costs. The split-phase remote memory access primitives with explicit synchronization are conducive to efficient implementation and allow the compiler to build blocking reads and writes in addition to the split-phase gets and puts. The Split-C language does not exploit the full capabilities of libsplit-c due to inevitable syntactic restrictions. A more sophisticated compilation approach could relieve the programmer from some of the low-level details and overlap remote accesses more aggressively.

The use of Active Messages has been a catalyzing element in the development of libsplit-c (and Split-C itself). The flexibility offered by the Active Message handlers allowed experimentation with a variety of remote memory accessors before the ones available today emerged. While Split-C's puts, gets, and stores are extremely simple and easily implemented in the context of shared memory hardware, no other shared memory framework supports similar operations. In addition, the current set of accessors explores only a fraction of the design space: a library of atomic operations on remote data structure elements is in development and uses the capability to perform simple atomic memory accesses (such as fetch and add, or enqueue) in the handler more extensively. In this context, the Active Message handlers allow atomic remote memory accessors to be tailored to each data structure.

Split-C and its associated run-time substrate illustrate the success of the layering model proposed in Subsection 2.1.2: the libsplit-c run-time builds communication primitives suited to the Split-C compilation process using the Active Messages communication architecture without compromising on performance. The communication performance observed in Split-C programs is directly comparable to the costs of the Active Message primitives.

6.2 TAM/Id90

Highly dynamic parallel languages such as Id90, Multilisp, and CST pose a serious challenge to the language implementor and the computer architect. If compiled in a straight-forward manner, these languages require execution models with very short threads—on the same order as basic blocks in sequential languages—and arbitrary context switches from one thread to the next. Attempts to support such execution models in hardware lead to dataflow and message driven architectures which, as discussed in Section 5.2, are not suited to efficient implementation. Mapping these execution models directly onto conventional hardware is not efficient either because of the high frequency of dynamic scheduling.

The Threaded Abstract Machine (TAM) approach described in this section uses powerful compilation techniques [SCvE91] to reduce the scheduling overhead and pay careful attention to the storage hierarchy such that Id90 can be mapped efficiently to conventional multiprocessors. The TAM compilation approach relies heavily on Active Messages to integrate communication into the code generation process[5]: the compiler generates message sends in-line and produces a custom handler for every message type to store the arriving data directly into a local variable and enable the appropriate computation.

Active Messages place a number of responsibilities in the compilation system that other communication models handle within the architecture: memory allocation for messages, scheduling of communication and computation, deadlock/livelock avoidance, and address translation. The Id90/TAM approach demonstrates not only that the compilation system can handle these responsibilities, but that the compiler can exploit and optimize special cases to the point where specialized hardware support in the form provided in message driven architectures is actually counter-productive. In TAM all storage resources required for communication are allocated in sizable chunks before they are required (e.g., no dynamic memory allocation occurs on message reception), the run-time substrate maintains a scheduling hierarchy adapted to the language, the restriction to request/reply communication patterns is observed, and pointers to global data structures are integrated into the type system.

The following subsection introduces the storage and scheduling hierarchies which are fundamental to TAM, relates them to the compilation challenge, and briefly describes the TAM execution model. Subsection 6.2.2 evaluates the use of Active Messages in Id90/TAM. In particular, the cost of communication, the influence of tight integration of computation and communication on the dynamic scheduling cost, and the pre-allocation of storage are discussed. Further information on TAM can be found in [CGSvE93], the compilation of Id90 to TAM is discussed in detail in [Sch94], and the mapping to the CM-5 in [Gol93].

6.2.1 The TAM/Id90 compilation system

The compilation algorithms used in mapping Id90 onto TAM extend the compiler technology developed in the dataflow community. In particular, Traub's "compilation as partitioning" framework [Tra88] and Iannucci's thread generation for the hybrid architecture [Ian88] demonstrated that it is possible to reduce the amount of dynamic scheduling required during the execution of such languages. The Id90/TAM compiler addresses two additional issues as well. First, communication and scheduling are explicit and subject to compile-time optimization rather than implicit in the execution model. Second, storage allocation is explicit and occurs in large chunks, namely activation frames and heap data structures, and the compiler is responsible for storage management.

6.2.1.1 Storage and scheduling hierarchies

In order to involve the compiler with communication and scheduling, TAM exposes the scheduling data structures together with a simple cost model for optimizing the relationships between scheduling, communication, and computation. The TAM storage hierarchy, shown in Figure 6-10, is composed of the processor registers, the activation frame of the currently executing function, and the heap. At the

5. In truth, implementing TAM on the nCUBE/2 was the initial motivation for developing Active Messages.

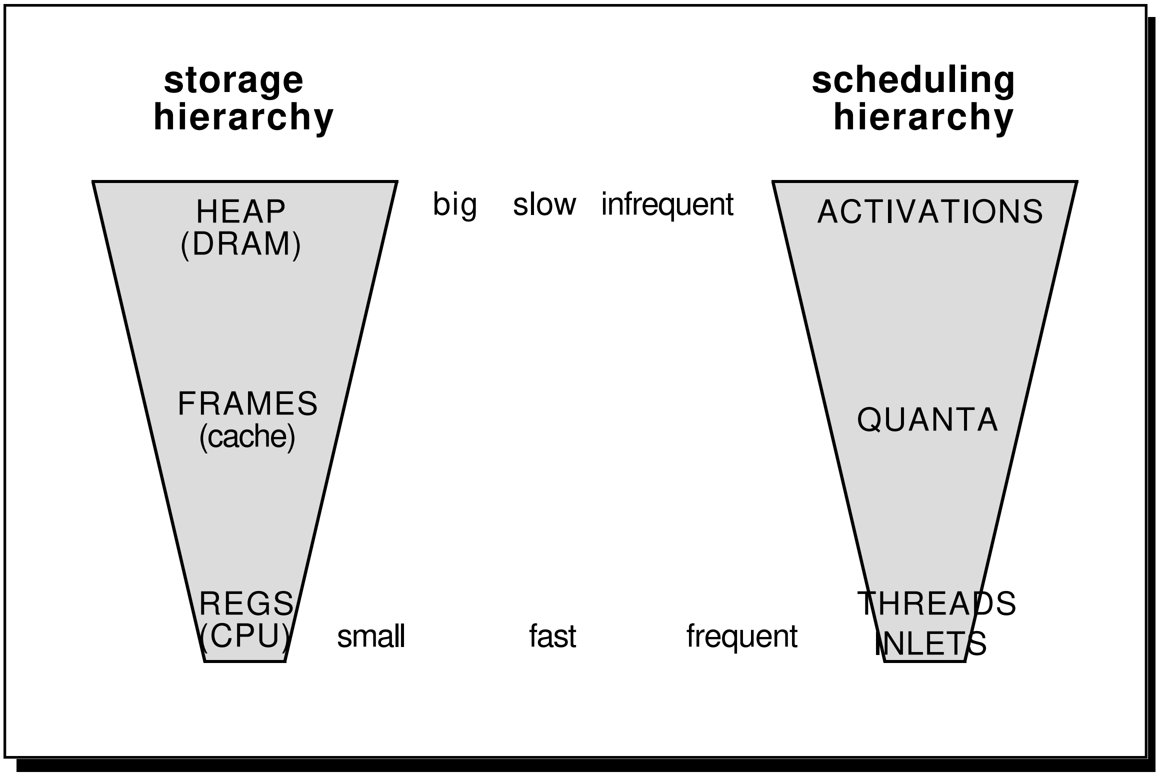


Figure 6-10: TAM storage and scheduling hierarchies.

bottom of the storage hierarchy the amount of state available is small, accesses are fast and can be frequent; at the top, the state is large, but accesses are slow and should occur in larger transfer units at lower frequency.

The structure of TAM's scheduling hierarchy parallels the storage hierarchy. At the bottom *threads* and *inlets* represent units of computation and of communication, respectively. Each thread and each inlet is small but executes quickly and switching from one to the next is cheap. At the next level, groups of threads that are part of the same function invocation and that execute together form *quanta*. Thus quanta are larger but require a heavier-weight scheduling mechanism and switching is more expensive. At the top, the invocation of activations requires dynamic storage allocation and load balancing.

TAM is based on the principle that operations in the scheduling hierarchy strongly affect the storage hierarchy. Thus, the optimization challenge posed to the compiler is to coordinate the two such that the constraints of the language model and of the communication architecture are met with the least movements in the storage hierarchy. The mechanisms available to the compiler are described in the following discussion of the execution model.

6.2.1.2 Execution model

A TAM program consists of a collection of *code-blocks*, which typically represent functions or loops in the source program. Each code-block consists of a collection of *threads*, which correspond roughly to basic blocks, and of *inlets*, which are compiler generated Active Message handlers. Two instructions appear in the same thread only if they can be statically ordered and if no operation whose latency is unbounded occurs between them (i.e., threads are straight-line code sequences). Inlets handle all messages received by the codeblock invocations, transferring data directly into the frame and enabling the threads using the data.

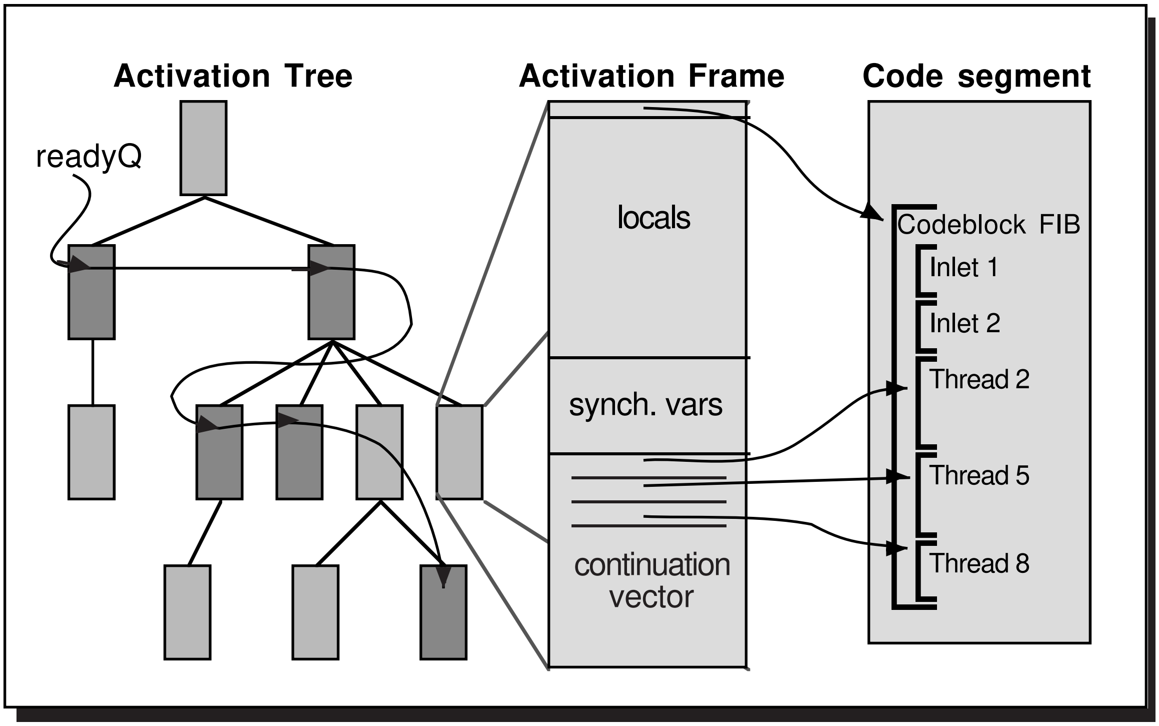


Figure 6-11: TAM activation tree.

The TAM execution model centers on the *activation frame*, shown in Figure 6-11, which is the analog of a stack frame for parallel calls. To invoke a code-block, a frame is allocated on a processor and initialized, and arguments are sent to the frame. In addition to the local variables typically found in stack frames the TAM activation frame also holds the scheduling queue and the synchronization variables controlling low-level scheduling. A portion of each frame is used to hold a stack of instruction pointers, called the *continuation vector*, representing the enabled threads for the corresponding function invocation. The compiler can determine the maximum size of this region in a manner much like register allocation. Each processor maintains a scheduling queue of local activations with enabled threads by linking the frames together.

TAM threads are straight-line code, with synchronization occurring only at thread entry. The basic control operation is to *fork* a thread; the fork may be conditional (on a boolean variable) or unconditional and the thread may be synchronizing (i.e., require multiple forks) or not. Thread synchronization is handled using synchronization counters. Each synchronizing thread has an associated counter which is initialized to the thread entry count, i.e., the number of forks required before the thread can execute. (The counter is omitted for *unsynchronizing* threads.) Each fork decrements the counter and the thread is only enabled if the counter reaches zero.

Each TAM processor contains a frame-pointer register that refers to the currently active frame on the processor and an instruction-pointer register that refers to the current instruction of the current thread of the code-block associated with the current frame. In addition, there is a set of general purpose registers. Distinguishing registers from frame slots allows the compiler to exercise control over the local processor storage hierarchy. Registers are not implicitly saved or invalidated when a thread switch or frame swap occurs; they are explicitly managed by the compiler: registers can be allocated across thread boundaries and the compiler generates code to save and restore live registers at frame swap boundaries. TAM instructions are generally of three-address form, with operands drawn from frame slots, processor registers, or immediates.

Activations communicate arguments and results using explicit messages. A message is formed and issued with the `send` instruction, which sends a number of data values to an inlet of a potentially remote frame. The inlet is a stylized compiler generated Active Message handler that receives data from the network and stores it into local variables in the destination frame. It will also *post* a thread into that frame, if the entry count is satisfied. Post is analogous to fork, except that it enables a thread for the frame associated with the message, whether it is currently running, idle, or ready, while fork enables a thread for the currently running frame. If the activation is idle when the post occurs, it is placed on the scheduling queue. If it is ready, i.e., already on the queue, it simply accumulates another thread in the continuation vector. If the activation is currently running, the inlet may take some special action to put the data directly in registers or it may simply fork the thread.

To minimize the interaction between inlets and the scheduling events within threads, each activation's continuation vector is split into two parts, a remote continuation vector (RCV) in the frame and a local continuation vector (LCV) that is part of the processor, like registers.[6] Forks push thread pointers onto the LCV, whereas posts in inlets push thread pointers onto the RCV. Maintaining the continuation vector in the frames provides a natural two-level scheduling hierarchy which enhances the locality of reference within the processor. When a frame is scheduled, the *remote* continuation vector (RCV) is copied into the *local* continuation vector (LCV), from which enabled threads are executed until the LCV is empty. The set of threads that run during this time is called a *quantum*.

Global data structures in TAM provide synchronization on a per-element basis to support Id's I-structure and M-structure semantics [ANP87]. In particular, reads of empty elements are deferred until the corresponding write occurs. Accesses to the data structures are split-phase and are performed via special instructions: *ifetch* reads an element by sending a message to the processor containing the data which returns the value to an inlet, *istore* writes a value to an element, resuming any deferred readers, and *ialloc* and *ifree* allocate and deallocate I-structures. Access to I-structure elements is performed by a standard inlet on the processor that is local to the accessed element. It reads the element and its presence bits, performs the state transition, and replies to the program inlet.

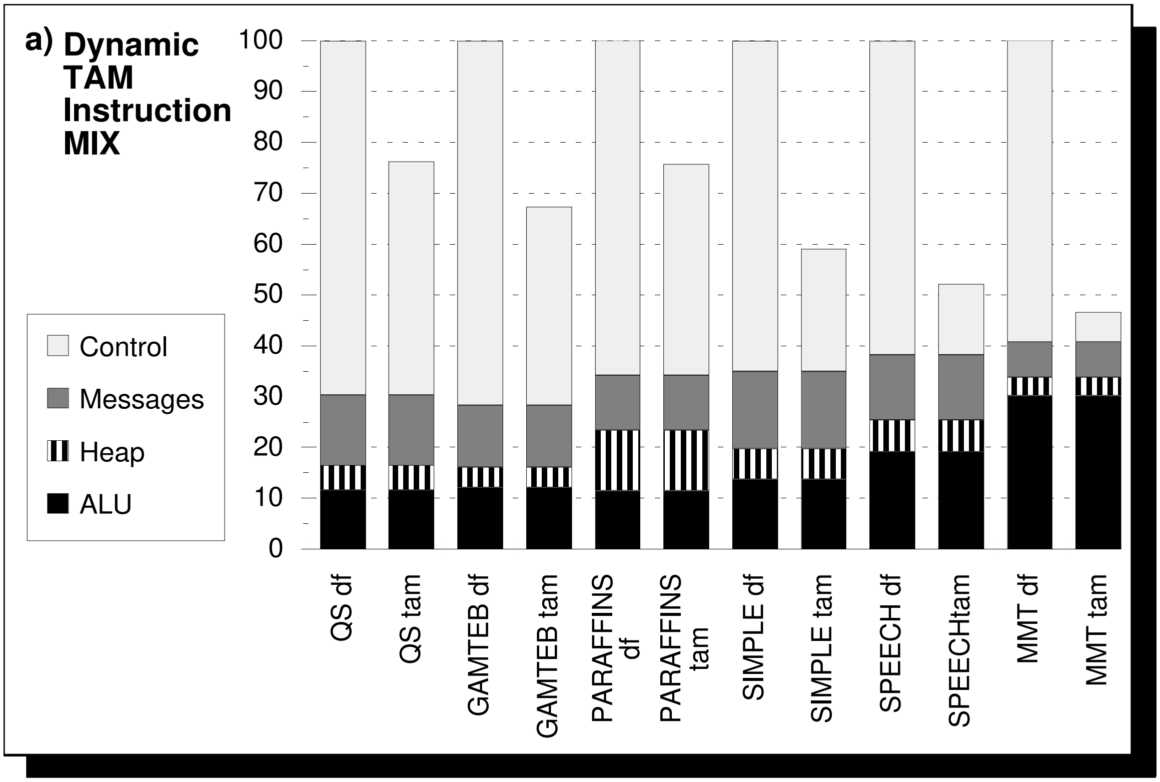
6.2.2 Evaluation

This subsection analyses statistics on dynamic scheduling and communication to evaluate the effectiveness of the Id90/TAM compilation system for conventional multiprocessors. The programs are compiled to TAM instructions by the Id90 compiler and then translated by the TAM back-end to Sparc assembly language on the CM-5. TAM uses a variation on the CMAM communication architecture and generates the code to send and receive messages directly in-line. Although the message formats are the same as in CMAM and the code sequences are similar, TAM does not use the Sparc register windows nor the standard calling sequence. Instead, it uses only a single window as a flat 32-register file and reserves eight registers for inlets (e.g., Active Message handlers).

The results presented here are adapted from a comparison of mechanisms for fine-grained parallel programs in the J-Machine and the CM-5 [SGS+93] and stem from six benchmark programs described below. TAM-level dynamic instruction distributions are collected by running an instrumented version of each program on the CM-5. The translator inserts in-line code to record roughly a hundred specific statistics on each processor, which are combined at the end of the program. These are grouped into the basic instruction categories in Figure 6-12a. *ALU* includes integer and floating-point arithmetic, *messages* includes instructions executed to send and handle messages, *heap* includes global I-structure and M-structure accesses, and *control* represents all control-flow instructions including moves to initialize synchronization counters.

The frequencies of the various operations comprising each of the programs are then used to weight the cycle cost of each of the operations. The result is a graph which shows the relative contribution of each operation category to the overall execution time of the programs. This statistical method is similar to

6. Alternatively, inlets may execute on a separate processor that is closely coupled to the thread processor. Still it is valuable to minimize the interactions between the two.



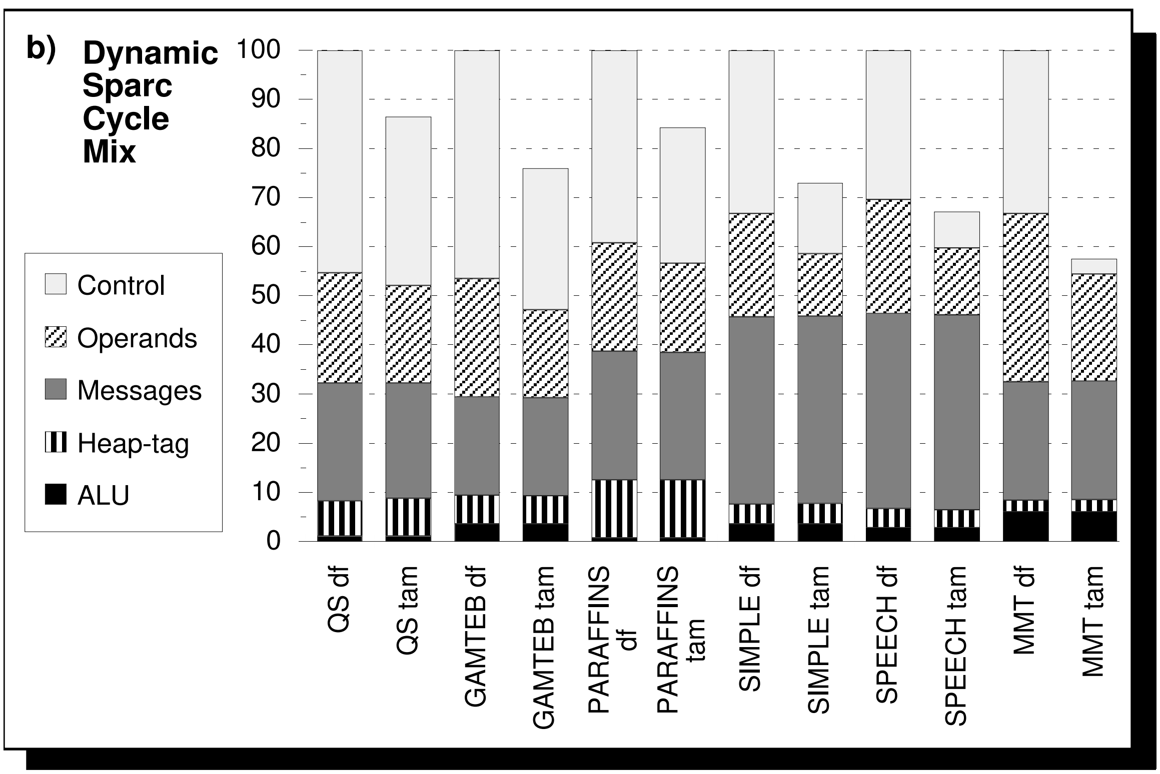


Figure 6-12: Dynamic program execution statistics

The top bar graph shows the TAM instruction mix for two versions of each program: the "df" version uses dataflow compilation techniques while the "tam" version includes TAM-specific compiler optimizations. The bottom bar graph shows the Sparc cycle count breakdown for the same programs.

narrow spectrum benchmarking [SB92] and has the advantage that different—even hypothetical—machines can be compared: for each new machine only a cost vector capturing the average cycle cost for executing each primitive needs to be developed. Weighting this cost vector by each program's statistics yields a prediction of the execution time on the new machine.

This methodology is used below to evaluate the cost of dynamic scheduling and of communication. Comparing the normal CM-5 to a hypothetical CM-5/J-Machine hybrid machine gives an indication of the improvements possible with specialized hardware. The cost vectors for the hybrid machine are adapted from [SGS+93] where it is developed by modifying the J-Machine to include Sparc features such as an on-chip cache, an on-chip floating-point unit, and 32 integer registers. Adding these "mundane" architectural features to the J-Machine creates a machine equivalent to a CM-5 with an on-chip network interface and message handler dispatch in hardware.

The six benchmark programs range from 50 to 1,100 lines. *QS* is a simple quick-sort using accumulation lists. The input is a list of random numbers. *Gamteb* is a Monte Carlo neutron transport code [BCS+89]. It is highly recursive with many conditionals. *Paraffins* [AHN88] enumerates the distinct isomers of paraffins. *Simple* is a hydrodynamics and heat conduction code widely used as an application benchmark, rewritten in Id90 [CHR78, AE88]. *Speech* determines cepstral coefficients for speech processing. *MMT* is a simple matrix operation test using 4x4 blocks; two double precision identity matrices are created, multiplied, and subtracted from a third. The programs toward the left of Figure 6-12a and b represent fine-grained parallelism. They are control intensive and make frequent remote references, as opposed to the blocked matrix multiply which is dominated by arithmetic. The Id90 implementations of comparable programs take about twice as long on a single processor as implementations in standard languages like C or Fortran [CSS+91]. Some of this overhead (mostly seen in the amount of control in Figure 6-12) is inherent in any parallel implementation and not specific to Id90.

The evaluation of TAM using Active Messages proceeds in three parts. The dynamic scheduling cost demonstrates the effectiveness of the compiler optimizations in reduce the overall cost of scheduling, the storage allocation scheme avoids memory allocation on message arrival, and the communication cost highlights the success in integrating communication into the compilation process as well as the value of hardware support for communication.

6.2.2.1 Dynamic scheduling cost

The cost of dynamic scheduling is reflected in the frequency of control instructions which handle basic control flow among threads (*fork*, *switch*, *stop*, and *swap* instructions) as well as asynchronous events due to message arrival (*post* instructions). In TAM the compiler attempts to reduce the dynamic scheduling cost by producing as large threads as possible (thereby lowering the frequency of thread switches) and by specializing each scheduling operation to its context of use (to avoid the expensive general case).

The effect of compiler optimizations[7] to produce longer threads is demonstrated in Figure 6-12 which shows dynamic TAM instruction and Sparc cycle distributions with and without optimizations: the version of each program labeled "df" uses standard dataflow thread generation while the version labeled "tam" is optimized for TAM's scheduling. The TAM instruction mix in Figure 6-12a shows that the number of arithmetic instructions, heap accesses, and messages is not affected by the optimization but the number of control instructions is reduced by 60% on average. Figure 6-12b shows the number of Sparc cycles executed for each of the programs. A new cycle category, *operands*, represents the cycles spent accessing operands in the activation frame. The cycle counts show that the reduction in TAM instructions translates into fewer clock cycles for both, control and data access. The overall reduction in execution time by 25% demonstrates that the new optimizations enabled by exposing the scheduling to the compiler result in significant savings.

Beyond reducing the frequency of control operations, the compiler also specializes each scheduling operation. Table 6-3 shows the costs of the many variants on each of the control operations in terms of execution cycles on the CM-5 and on the hypothetical CM-5/J-Machine hybrid. Dynamic scheduling

7. The details of the compiler optimization is the subject of Klaus Schauser's Ph.D. thesis [Sch94].

	TAM operation	specialization	clock cycles CM-5	CM-5/J-Machine hybrid
scheduling within the computation	fork	fall through	0	0
		branch to thread		
		unsynchronizing	1	1
		successful sync.	4	4
		unsuccessful sync.	13	13
		push onto LCV		
		unsynchronizing	5	5
		successful sync.	10	10
		unsuccessful sync.	7	7
	switch	(same as for fork)	2+fork	2+fork
	stop	pop from LCV	5	5
	init sync		4	4
scheduling on message arrival	—	poll	18	0
	post	idle frame		
		unsynchronizing	19	19
		successful sync.	23	24
		unsuccessful sync.	7	11
		ready frame		
		unsynchronizing	14	15
		successful sync.	19	20
		unsuccessful sync.	7	11
		running frame		
		unsynchronizing	8	31
		successful sync.	12	31
		unsuccessful sync.	7	27

Table 6-3. Cost of TAM synchronization and scheduling instructions

For most operations several different versions reflect the various compiler optimizations or run-time conditions, such as whether a fork can be combined with a stop into branch, whether the target thread is synchronizing, and whether the synchronization was successful or not.

The CM-5/Hybrid is a hypothetical machine consisting of a CM-5 augmented with the J-Machine on-chip network interface and automatic handler dispatch on message arrival. (The numbers are adapted from a comparative study of the J-Machine and the CM-5 [SGS+93].)

among threads within the same function is cheap. To fork a new thread, the address of the thread is simply pushed onto the LCV. When a thread stops, a pop-jump removes the next thread from the LCV and starts it. Coordination among threads uses the synchronization counters which are decremented by forks to synchronizing threads. Synchronization counters are implemented as locations in the frame which must be initialized to the entry count of their associated thread. This initialization incurs the cost of a memory write.

The compiler specializes most forks into fall-throughs and branches, which eliminates the corresponding stop at the end of the thread. The remaining forks translate into a push onto the LCV. A pointer to the top of the LCV is kept in a register. The top portion of Table 6-4 shows the cost of the specific code sequences, depending on the position of the fork in the thread and whether the destination thread is synchronizing or not. Taking the dynamic frequencies of these cases into account, the average cost of a fork is about 8 cycles on the CM-5. Nonetheless, forks account for roughly half of the cost of control operations.

The instruction sequence executed for post depends not only on whether the target thread is synchronizing or not but also on the state of the activation frame. When the frame is idle or ready, the pointer to the top of the RCV must be used; it is contained in the frame, not a register. Also, if the frame is idle it must be placed on the ready queue. However, if a thread is enabled (e.g., the synchronization succeeds) the thread is being posted to the running frame, it can simply be pushed onto the LCV, like a fork.

An important consideration in the implementations of fork, post, and stop is atomicity. Inlets and threads interact in that they share the synchronization counters, the continuation vector, and the frame ready queue. Thus, the implementation must guarantee that all scheduling operations (e.g., fork, stop, and swap) are atomic with respect to inlet execution, in particular with respect to the post instructions. On the CM-5, the compiler inserts polls into the threads to allow for message reception which incurs a cost of 18 cycles for each poll but ensures that fork, stop, and swap run atomically relative to post. Eliminating the cost of polling by implementing an interrupt-based approach on the CM-5 is impractical for fine-grained parallelism. This is due to the high cost of an interrupt, about the same as 10 polls.

On the CM-5/J-Machine hybrid no polls are necessary due to the automatic dispatch on message arrival; however, atomicity becomes a problem. Rendering fork and stop atomic relative to posts would require bracketing each expansion with interrupt disable/enable instructions. This would incur a four cycle cost on every thread scheduling operation! Instead, the expansion of post in the case the corresponding activation is running (i.e., in the case atomicity is an issue) is modified to push the thread onto the RCV instead of the LCV. When the LCV is empty, instead of switching to the next activation, the RCV is double checked to pick-up threads posted during the quantum.

The CM-5 implementation of post is straightforward, as described above. The net result of specializing the post instruction is that the most common case, posting to a running frame, costs only 2 cycles more than a fork instruction. For example, in Gamteb the posts to a running frame account for 51% of the posts. The cost of a post is reduced further in inlets that handle an ifetch response. For the case that the element being fetched is local and present, the response inlet is inlined directly into the thread. This eliminates all the inlet overhead and as a result turns the post into a fork. The result of all the specializations is that the average post instruction takes between 9 and 13 cycles depending on the application program.

Control in fine-grained parallel programs involves the integration of asynchronous events with the control flow internal to the computation. In TAM, the two are closely related as the compiler generates the code for both scheduling levels. With this kind of coupling, the compiler can partition the available registers by convention; hardware partitioning into distinct contexts can restrict how registers can be used and can prevent certain optimizations. The close relationship between the two levels also requires that asynchronous scheduling be complemented by efficient atomic operations on shared resources. Polling for messages avoids this issue and is acceptable for fine-grained parallelism. For instance, in our benchmark programs, polling accounts, on average, for 4% of execution time. It may, however, constitute unnecessary overhead for coarser-grained computation.

6.2.2.2 Communication cost

Communication is handled using explicit send instructions and inlets to handle message arrival. Table 6-4 shows the cost of sending and receiving a message using synchronous reception on the CM-5 and asynchronous reception on the CM-5/J-Machine hybrid. The start-up cost for sending a message to a remote processor is 3.5 times as high on the CM-5 as on the hybrid, mainly due to the status regis-

TAM operation	CM-5 Cycles		CM-5/J-Machine hybrid cycles
	synchronous reception		asynchronous reception
	remote messages	local messages	all message
Send N-word message	$25 + 4\left\lceil \frac{N}{2} \right\rceil$	$4 + N$	$7 + N$
Receive N-word message	$13 + 12\left\lceil \frac{N}{2} \right\rceil$	$4 + 4\left\lceil \frac{N}{2} \right\rceil$	$5 + 7\left\lceil \frac{N}{2} \right\rceil$
Poll	9	—	—
Ifetch message	93	16 or 4†	24
Istore message	60	4 or 2	15

†. Cost when the element is present for an ifetch, or empty for a istore.

Table 6-4. Cost for sending and receiving messages.

ter check. The per-word cost is twice as high on the CM-5, due to the cost of stores across the MBUS. Periodic polling is required in the synchronous reception models. The start-up cost for receiving a message includes the dispatch and the return to the interrupted computation. It is roughly twice as high on the CM-5 with synchronous reception. However, the extremely high receive start-up cost for the asynchronous reception on the CM-5 shows that the user-level interrupt which comes with the hardware dispatch is a dramatic improvement. The 100 cycles for the kernel trap on the CM-5 is an approximation and assumes that, on average, two to three messages are received back-to-back per interrupt. Notice that the asynchronous model also increases the cost of local messages, in that local messages must be atomic with respect to remote messages. A remote ifetch involves two messages (request and reply), and a remote istore involves a single message.

Given the high message cost on the CM-5, it is advantageous to treat local messages as a special case in software if their frequency is non-negligible, i.e., to exploit send to local frames and ifetch and istore to local structures. The required destination check costs two cycles per message. The "local messages" column of Table 6-4 shows the resulting costs for local messages. The local ifetch and istore entries show that optimizing for local messages can enable further optimizations: the access portion of the ifetch is performed directly in the thread and if the data is present the threads which use it can be enabled cheaply.

The implementation costs highlight three key points: the J-Machine integration of the network interface close to the processor reduces the communication costs substantially, receiving is more expensive than sending on both machines, and optimizing for local messages can be valuable. On Simple, where 93% of the messages are remote, the CM-5 spends three times as much time on messages as the CM-5/J-Machine hybrid. However, on Gamteb where 30% of the messages are local the CM-5 spends only twice as much time. Note, however, that the improvement on the CM-5/J-Machine hybrid is mostly due to the on-chip network interface and less to the special communication instructions.

Several factors cause message reception to be more expensive than sending. Sending is synchronous to the computation whereas reception is logically asynchronous. Thus values to be sent are typically available in registers, whereas values received must be stored in memory until the waiting computation can be scheduled[8]. Reception involves a dispatch, while sending does not. Finally, stores to the CM-5 network interface (to send a message) can take advantage of the write buffer, while loads (to receive) see the full network interface access latency.

8. Moving computation into the inlet, as in a message-driven model, does not help because it replaces memory accesses for storing message data with memory accesses for retrieving local operands.

6.2.2.3 Storage allocation

The Id90/TAM implementation pays careful attention to storage allocation to ensure that memory allocation on message arrival is not required. This is mainly achieved by preallocating storage within each activation frame to hold all data destined for that activation.

Each function receives its arguments via messages sent to a set of inlets and the results returned by child invocations arrive via messages as well. In both cases, storage for the data is pre-allocated in the frame in the same manner as for ordinary local variables. The values returned by remote memory fetches are handled similarly.

There are two situations that are more difficult to handle: deferred ifetches and remote frame allocation. When an ifetch request is received, the ifetch handler inspects the I-structure element tags and must enqueue the request if the element is empty. Enqueuing the request clearly requires storage which cannot be foreseen. Fortunately, all enqueued requests require the same amount of storage: four 32-bit words holding the requesting processor number, the frame pointer, the inlet number, and a link to the next enqueued request. The strategy adopted in the TAM implementation is to allocate a large number of deferred ifetch descriptors at a time such that only every few hundred-th deferred ifetch incurs the memory allocation cost.

Activation frames are typically allocated on a remote processor in order to balance the load. The current implementation performs the memory allocation within the message handler. The frame allocator used maintains separate bins for frequently used frame sizes such that in most cases the allocation is very cheap. In order to guard against memory overflow problems, the compiler generates two distinct inlets to which the remote allocator can respond. One of the inlets indicates success and receives the pointer to the new frame while the other indicates failure and retries the allocation on another node. The overall frequency of frame allocation messages is low enough that the time taken for the memory allocation in the handler is not of great concern.

6.2.3 Conclusions

Dataflow and message driven architectures have been proposed to allow the efficient execution of highly dynamic parallel languages such as Id90. The Id90/TAM compilation system described in this section takes a different approach and uses Active Messages on conventional multiprocessors to expose communication and scheduling to the compiler. This allows the entire compilation process, in particular code generation, to take communication and scheduling into account and to optimize each special case. The result is that the frequency of dynamic scheduling is reduced dramatically and the cost of each specialized scheduling event is far lower than the general case implemented in message driven hardware.

The comparison of communication costs on the CM-5 with the J-Machine (in the form of a hypothetical CM-5/J-Machine hybrid) show that bringing the network interface close to the processor is highly beneficial, but that the J-Machines's novel hardware scheduling mechanisms are not satisfactory. This is mostly due to insufficient attention to atomicity issues: the tight integration of computation and communication which results from the optimized compilation strategy requires the scheduling of computation to be atomic relative to message arrival. The frequency of these scheduling operations makes the cost of forming an atomic section critical and polling, especially if well implemented, appears as an attractive solution.

The TAM approach relies heavily on the versatility and flexibility of Active Messages in all aspects of the compilation. The compiler generates message handlers which efficiently couple communication and computation. Each handler contains specialized code sequences to store the message data into the right variables in the activation frame and to enable (post) the appropriate thread. In particular, the code sequence for the post takes the type of thread (synchronizing or not) into account and performs different actions depending on the state of the activation frame (running, ready, or idle). The combination of the Active Messages efficiency and of the optimized scheduling constructs brings the cost of dynamic scheduling down to a point where specialized hardware support is questionable.

The Id90/TAM approach also demonstrates that the responsibilities Active Messages places on the compilation system can be taken into account. All communication is one-way or round-trip such that deadlock and livelock is avoided, and storage is preallocated in sizable chunks such that no memory allocation is necessary on message arrival (while a few exceptional cases requiring allocation remain, their frequency is kept low).