7 Conclusions and Prospect

Active Messages achieves an order of magnitude performance improvement over message passing and supports a variety of programming models, including shared memory and dataflow, efficiently on the same hardware platform. A departure from previous attempts at developing efficient communication architectures is key to this success: the entire system has to be taken into account, from the micro-architectural level up to the language system. The current state of languages and compilers makes it possible to predicate the architectural design on the use of high-level parallel languages. Exposing the hardware and its performance characteristics to the compiler in a rational manner allows the key communications issues to be addressed in a specialized form in the run-time substrate of each language instead of requiring the architecture to implement a general solution. However, determining the right level at which to address each issue requires complex trade-offs and the various layers of abstraction must be designed carefully to avoid unnecessary overheads.

This dissertation develops a conceptual framework in which communication can be integrated into the compilation process and trade-offs can be made across all hardware and software layers of the system. The notion of a communication architecture which extends the sequential instruction set architecture forms the central focus of the framework. It allows the RISC design philosophy to be leveraged and the criteria that are used for ISA design and evaluation to be extended to communication.

The most important attributes of a communication architecture are versatility, efficiency, and incrementality: a communication architecture must be versatile to support a variety of parallel programming models and emerging parallel languages, it must be efficient in the mapping from language constructs to communication primitives and in the mapping of these primitives onto the processor and network hardware structures. Because modern parallel machines are essentially a collection of workstation-class computers, the communication architecture must also be incremental and complement the sequential instruction set architecture without disturbing its versatility or efficiency.

Active Messages is proposed as an efficient, versatile, and incremental communication architecture. It addresses the four key communications issues (data transfer, send failure, synchronization of computation and communication, and network virtualization) at carefully chosen levels of the system:

- On the sending side the computation transfers data directly into the network interface when sending a message. On the receiving side, the Active Messages handler extracts the data from the network and places it into the locations expected by the computation. The mechanisms for transferring data from the application data structures into the network and vice-versa are kept implementation dependent and are expected to vary from one machine to another, just as the addressing modes and instruction encodings vary.
- Each message names its handler allowing the communication architecture to accept incoming messages anytime without buffering or queueing by running the handler. To prevent uncontrolled nesting of handlers, communication patterns are restricted to requests and replies

which allows the communication architecture to accept only incoming replies when a reply attempt fails.

- The execution of Active Message handlers is atomic relative to other handlers (except when attempting to reply) and cheap critical sections in the computation must be supported, although the specific mechanism remains implementation dependent. This allows the compiler to generate handlers which interact with the scheduling of computation by manipulating the appropriate data structures atomically.
- Virtualizing the network when using Active Messages requires checking the destination node and process of a message, as well as translating the handler address at the destination. Because a user-level handler must be executed on message arrival it is best to coordinate the scheduling of processes on all nodes running a parallel application, although this coordination can be relaxed with the appropriate hardware support in the network interface.

The effectiveness of the Active Messages approach is demonstrated via implementations on the nCUBE/2 and CM-5 multiprocessors. On the CM-5, Active Messages reduces the communication overhead by an order of magnitude over traditional message passing models and enables competitive implementations of message passing, shared memory, and message driven programming languages on a single hardware platform.

The improvement over message passing is achieved by eliminating expensive general purpose memory allocation and synchronization from the communication architecture and placing the responsibilities onto the run-time system where they can be handled more efficiently. The analysis of various send&receive message passing implementations shows that, among other problems, the lack of a global address space prevents these models from achieving high performance because the sender cannot name pre-allocated resources at the destination to handle the message.

The reduction in overhead brings the latency of short round-trip messages to remote nodes into the same order of magnitude than found in shared memory multiprocessors which expend significant hardware support for communication. As shown in the case of Split-C, Active Messages allows access to remote memory locations with the same latency but with more flexible access methods than just read and write. This flexibility is exploited in Split-C to implement split-phase memory accesses and one-way stores. These offer high throughput and good overall system utilization because multiple remote references and local computation can all be overlapped.

In the case of fine-grained dynamic parallel languages, Active Messages enables new compiler optimizations allowing stock hardware to compete with custom dataflow and message passing processors. The compilation strategy employed for Id90 integrates communication in the form of Active Messages deep into the optimization process and illustrates that the compiler can specialize communication, memory allocation, and synchronization to the point that the expensive general mechanisms embodied in the custom architectures are counter-productive.

The communication performance demonstrated on the CM-5 with Active Message indicates that it is indeed possible to build general-purpose multiprocessors which support a large variety of parallel programming models efficiently. The CM-5 itself comes close to that goal and supports each of existing programming models as well as hardware tailored to a specific programming model (perhaps with the exception of cache-coherent shared memory). With respect to cache coherent shared memory, Active Messages supports the communication requirements well but intentionally separates the address translation issues from the communication and therefore requires additional hardware support to match the efficiency of the automatic address renaming occurring in cache-coherent shared memory hardware[1].

The research leading to this dissertation began with a strong hardware focus on the design of powerful network interfaces for multiprocessors. The study of existing systems quickly revealed that, not hardware per se, but the total system integration was the problem. If progress was to be made, it had to

1. Incidentally, the Wisconsin Windtunnel project managed to use the virtual memory hardware and the memory error correcting codes (ECC) to perform the renaming with considerable success.

build on a solid understanding of the entire hardware and software system and develop out of a conceptual framework for integrating communication into the compilation process[2]. With the foundation laid in this dissertation, it is now possible to close the circle and design communication micro-architectures which support a general-purpose communication architecture such as Active Message more efficiently than today's multiprocessors. In fact, this dissertation has in several occasions scratched the surface of possible improvements by analyzing the timing of Active Messages in detail on the CM-5 and the nCUBE/2 to uncover a number of simple hardware changes that would improve the performance dramatically. The fact that the implementations of Id90 on the CM-5 and the J-Machine use the same intermediate Threaded Abstract Machine (TAM) further allowed a direct evaluation of the on-chip communication features built into the J-Machine MDP processor. Section 7.1 below summarizes these findings and presents a number of network interface improvements proposed in related research projects. The large number of options underlines that Active Messages is in fact contributing to the development of higher-performance network interfaces than are available today.

2. Dataflow systems do integrate communication into the compilation process, but at the expense of the computational performance.

7.1 Impact on Communication Micro-Architectures

The simplicity and versatility of Active Messages provide an attractive optimization goal: Active Messages supports a wide variety of parallel programming models and thus new hardware features that improve the performance of Active Messages benefit all programming models alike. This is in contrast to existing approaches at improving performance, such as shared memory hardware, which improves a particular programming model, often at the expense of others.

The simple implementation of Active Messages in itself helps identifying possibilities for improvement. The two dozen instructions required to send or handle an Active Message allow a careful cycle-by-cycle accounting of the costs and in many cases the first-order bottlenecks are simple to alleviate, the nCUBE/2 being a particularly good example. This comes at no surprise: with the send&receive software overheads for buffering and synchronization, the cost of the few instructions accessing the network interface is insignificant and was never optimized.

Three subsections illustrate the possibilities for improvement offered by Active Messages: improving the network interface to reduce the access cost, hardware support for accelerating handlers, in particular, improving the dispatch on message arrival, and support for building a global address space.

7.1.1 Improving Network Interfaces

The most obvious, and probably also most important, improvement concerns the placement of the network interface in the node architecture. This is evidenced by the 50% of all cycles in CM-5 Active Messages that are spent in load and store instructions accessing the network interface across the MBUS. On the nCUBE/2 the problem is similar: the DMA instructions are executed on-chip but by the instruction fetch unit instead of the execution (or a separate DMA) unit with the effect that instruction prefetch stops for a dozen clock cycles.

The biggest obstacle to bringing the network interface closer to the processor is not necessarily of technical nature. Market forces dictate that the processor and closely linked support functions be off the shelf components which are designed and produced for the uniprocessor market. This limits the entry point at which a multiprocessor designer can insert the network interface in the overall architecture. For example, on the CM-5 it would be easy to place the network interface onto the cache bus instead of the memory bus, thereby cutting the access time to the NI in less than half. It would as well be feasible to implement the network interface as a coprocessor. Both options, however have only a short lifetime: the next generation processor (superSPARC) does not allow either option with the result that the network interface would have to be redesigned and that its performance would not scale with the computational improvement.

A number of research projects have explored closer coupling of the processor and network interface. The J-Machine (described in § 5.2.2.1) provides instructions to send directly from the register set and to receive into on-chip memory. The results in Section 6.2 suggest that adding a similar network interface to a CM-5 could reduce the execution time of Id90 programs by roughly 15%.

The iWarp processor [Bor88, Bor90] integrates the network interface (called communication agent) onto the processor chip as well. Careful attention is paid to enable the streaming of data, both, into the computation and into memory. A special set of registers (called gates) serve to access to outgoing and incoming FIFOs and can be used as operands in arithmetic instructions. Thus, in a single cycle, iWarp can access the message data and perform an arithmetic operation. iWarp provides multiple DMA engines (called spooling gates) to transfer messages directly between the memory and the network. Unlike on the nCUBE/2, the receiving DMA can be set-up after the head of the message has been inspected such that the data can be transferred directly into application data structures.

Another study [HJ92] adds a network interface to a Motorola MC88110 processor in three different places: off-chip on the second level cache bus, on-chip on the first-level cache bus, and on-chip as part of the register set. The results reported for two Id90 programs similar to the ones used in Section 6.2 show little benefit (under 5%) in bringing the network interface on-chip unless it is integrated into the instruction and register sets at which point overall improvements on the order of 20% are seen. Unfortunately the study does not compare the three variants to a machine with a network interface attached to the memory bus, such as the CM-5. The cycle counts for sending and receiving a message indicates a significant improvement over the CM-5 in all cases: the three variants take from 3 to 6 cycles to send a message and 5 to 8 cycles to receive! The dual issue capability of the MC88110 is found to improve sending and receiving of messages, however, computation benefits even more and the base machine with a network interface on the second-level cache bus spends over 50% of its time in the communication.

The MC88110/MP project [PBGB93] uses an MC88110 and adds an on-chip network interface as an additional function unit next to the integer, floating-point, and load/store units. The network interface consists of a number of register sets in which messages are composed and received. Special instructions move data between the general-purpose registers and the network registers and, using the dual-issue capability, allow 16 bytes to be added to a message in a single clock cycle. Receiving messages is limited to 8 bytes per cycle due to the smaller number of write ports into the general-purpose register files.

Beyond just bringing the network interface on-chip, the above projects also optimize the mechanics of message composition and reception. The use of registers to hold messages instead of a FIFO has several benefits. Foremost, in the case of send failure, the message is still available in the registers and can be re-sent without requiring the data to be pushed into the FIFO anew. This means that in an environment which integrates message send into the code generation (i.e., moves data directly from variables allocated to general-purpose registers into the network interface) a generic retry code sequence can be used. Several optimizations proposed in [HJ92] use registers to accelerate message composition: the return address contained in a request message can be moved to the head of the reply message within the network interface itself, and in consecutive messages with identical data or identical address only the new fields need be set.

Additional support in the network interface for the most frequent message types may be beneficial as well. For example in *T [NPA92], issuing a global memory fetch takes a single store double instruction (the network interface is memory mapped in this case). The 64-bit data value is interpreted as a global address and expanded in the network interface into a node/local-address pair. For the return address a pointer to the current thread descriptor (the current frame pointer in the case of TAM) is cached in the network interface and the return handler address is calculated from the low-order bits of the store address.

7.1.2 Hardware support for message handlers

The analysis of the hypothetical CM-5/J-Machine hybrid in Section 6.2 highlights the difficulty in integrating the asynchronous message arrival into the computation. Even though the J-Machine network interface allows much faster message composition and consumption, a large part of the benefits are compromised by the cost of forming critical sections that execute atomically with respect to message arrival.

On current message passing machines the way to signal an asynchronous event is to take an interrupt. This flushes the pipeline and enters the kernel. Executing the Active Message handler requires a crawl-out to user-level and a trap back into the kernel in addition. Forming critical sections is similarly expensive given that disabling and re-enabling interrupts both require kernel traps. The MC88110/MP improves the latter aspect by providing a user-level interrupt enable bit. The kernel retains it own interrupt mask which overrides the user-level bit. Critical sections cost a single cycle (e.g., two instructions) on the MC88110/MP.

The alternative to interrupts is polling. The MC88110/MP proposes several efficient polling primitives. The simplest form checks for message arrival and, if successful, loads the message into the network registers a single instruction. More powerful variants incorporate message handler dispatch into polling. The key idea is to combine the traditional status register of the network interface with the handler dispatch on the first word of the message. Instead of returning a boolean result, a poll can check the network interface and return a pointer to the appropriate code sequence handling the situation: if the previous send failed a pointer to the retry code sequence is returned, if a message has arrived the first word of the message is returned, otherwise the source operand to the poll (which may point to the thread scheduler or to the next instruction) is returned. The poll can then be followed by an indirect jump with the result that the poll and the handler dispatch occurs in two instructions. The result of the optimizations proposed in the MC88110/MP is that sending an Active Message with 20 bytes of payload takes two clock cycles (assuming the payload comes from the general-purpose registers) and handling the same message costs three cycles of overhead.

An interesting observation made in the course of implementing TAM on the J-Machine is that there is a continuum of intermediate design points between polling and interrupts. The basic interrupt mindset is that an interrupt can occur between any two instructions. With the introduction of critical sections, interrupts cannot occur during certain, typically short, code sections. If the size of these code sections is increased, then at some point interrupts are disabled most of the time and only enabled during brief "windows". This is the technique used to implement polling on the J-Machine: a poll consists of an interrupt enable immediately followed by a disable. (Similar interrupt windows are used in real-time operating systems to allow high-priority interrupts during lengthy paths through the kernel.) The trade-off between polling and interrupts is then a matter of cost and of convenience. Experience with Split-C and TAM indicates that polling is simpler and cheaper if code generation takes communication into account, i.e., can insert the polls automatically. The main problem with polling lies in code sequences compiled by a standard uniprocessor compiler[3] which do not include any polls and, as a consequence, do not service the network. A simple solution to this problem is to add a timer to the network interface which starts when a message is received and signals an interrupt if it is not handled after a configurable amount of time.

The processor can be off-loaded from the burden of handling asynchronous events by using a coprocessor to handle messages. The Intel Paragon, the Meiko CS-2, and, in essence, all cache coherent shared memory multiprocessors follow this approach. The coprocessor can be fully dedicated to communication and can poll the network continuously. The advantage of using a general-purpose processor as message coprocessor is flexibility. In general the coprocessor can be simpler than the main processor (for example, message handling can be restricted not to involve floating-point operations) and allow a tighter integration of the network interface and the processor itself. The Meiko CS-2, for example, integrates a simple Sparc-compatible processor directly onto the network interface (the main processor on each node is a superSPARC). Simple requests, such as a remote write or remote read are handled by hardware state machines and more complicated ones can involve the coprocessor. The Paragon forgoes the opportunity to integrate the network interface and uses two (or more) identical processors. The distinction between the "main" processor and the message coprocessor is a pure software convention.

The difficulty with message coprocessors is the communication between the processor and the coprocessor. If a message arrives and mostly needs to interact with memory, e.g., a remote read or write request, then the coprocessor solution works great. If a small message, for example the reply to a remote fetch, arrives and needs to be integrated into the computation then the coordination between the coprocessor and the processor poses problems. Typically, careful engineering attempts to use cache line transfers between the two units to bring the data close to the processor efficiently.

Cache coherent shared memory multiprocessor, in essence, turn the memory controller into a coprocessor with a fixed set of handlers hard-coded in state machines. The coordination between the processor and the memory controller is kept simple by limiting the types of requests (e.g., reads and writes) and, more significantly, by having the processor block whenever a remote request involves a reply that needs to be passed back to the processor. None of the processors used in current shared memory multiprocessors supports multiple outstanding replies[4].

---

3. As suggested in Chapter 1, most multiprocessors use a standard uniprocessor on each node; being able to use the highly tuned libraries (such as matrix multiply) developed for the uniprocessor is very attractive.

4. While weak stores support multiple outstanding requests, at no time is more than a single reply to the processor outstanding. Most cache controllers support multiple outstanding prefetches, but they are independent of the processor itself (this is why on all machines values are prefetched into the cache and not into registers).

7.1.3 Support for a global address space

The experience with Split-C has demonstrated that a global address space not only simplifies programming but is also beneficial from a performance point of view. Global addresses allow the message sender to name resources needed at the remote end to handle the message. Building a global address space in software without hardware support is clearly feasible but incurs a substantial cost. Split-C places the burden of controlling this cost on the programmer by exposing the difference between the local pointers understood by the hardware and global pointers which require translation in software. An example of a more automated approach is the Wisconsin Windtunnel which uses memory ECC (error correcting code) bits to detect pointers to remote memory and trap to the translation software only when required.

The general strategy promoted by Active Messages is to separate the notion of a global address space from the communication proper. This is exemplified in Split-C where address arithmetic is used to build a two-dimensional address space while the accessors to objects in this global address space are implemented using Active Messages. This set of accessors can be extended to new data types, new forms of remote operations, as well as new synchronization between the communication and communication without any interference with the global address space itself.

The most costly operation related to Split-C's global address space is the local/remote check that occurs for every global pointer dereference. Performing this check in hardware is exceedingly simple and only requires comparing a bit field in the address with the local node number. The difficulty is in keeping the memory access cheap in case of local accesses. Even if the check occurs in a single cycle, a check-branch-load/store sequence remains significantly more expensive than a simple load/store, as a shared memory architecture would support.

For large data structures that are mapped across multiple nodes, the cost of the address arithmetic required for the layout is higher than that of the local/remote check. For example, indexing into a Split-C spread array involves the code sequence shown in Figure 7-1 which includes a division and modulo by the number of nodes in the system. On the CM-5, a power-of-two number of nodes is assumed such that a shift and a mask can be used instead of the divide and modulo. Nevertheless, while the local remote check costs only two cycles, the indexing costs five.

If global memory access randomization using hashing and low-order interleaving techniques is desired, the address arithmetic is further complicated. Memory controllers such as the one designed for the RP3 [Pfi85] apply a pseudo-random hashing function to the low-order address bits and rotate the result right in order to prevent memory references with a constant stride to cause contention on a small number of memory modules.

The final form of support desirable for building a global address space concerns data movement and replication. The essential support required to move the location of data transparently is renaming of addresses. In essence, the associative cache look-up in cache-coherent shared memory multiprocessors allows any global address to be transparently renamed into a local address (that of a cache line). The J-

```
int *global sPtr;
int index;
# ...
sPtr += index;
# ...
```

```
struct gaddr { int node; int *addr; };
struct gaddr sPtr;
int index, tmp;
# ...
tmp = sPtr.node + index;
sPtr.node = tmp % PROCS;
sPtr.addr += (tmp/PROCS)*sizeof(int);
# ...
```

Figure 7-1: Typical code sequence for Split-C spread array indexing

Machine attempts to provide a similar mechanism explicitly in the form of two instructions which access an on-chip associative translation cache. The `enter` instruction enters a value with the corresponding tag into the cache and the `xlate` instruction queries the cache for the value corresponding to a tag. However, this mechanism does not appear to be useful in practice [NWD93], primarily due to the fact that these instructions are not very fast on the J-Machine and because the cache the trap taken when `xlate` misses is too expensive. In a similar attempt, the Wisconsin Windtunnel uses the Sparc virtual memory system to control renaming of remote addresses to local memory and implements a variety of cache coherency protocols in software. The enabling factors for the Windtunnel are the explicit software control over the renaming of addresses and the separation of the address renaming from the communication itself. A major problem experienced in the Windtunnel project is the page-size granularity of the virtual to physical address mapping.

7.2 Impact on Operating Systems

The discussion of Active Messages in this dissertation expressly assumes a parallel computing setting in which the scheduling of all processors participating in the execution of a parallel program is coordinated. This assumption is realistic because the tight coupling of all parts of a parallel program require this type of coordination for performance considerations and, as a consequence, numerous studies have explored coordinated scheduling policies for multiprocessors [GMB88, Cro93, GTU91, ZM90].

With Active Messages the execution of the message handler at user-level in some sense requires the destination process of a message to be running at message arrival time. On the CM-5, the operating uses strict gang scheduling to ensure that all processors of a partition run the same parallel process at the same time. In addition, to cope with messages that are in flight at the time of a context switch, the network state is fully saved and restored at a quantum boundary. More sophisticated network interfaces, such as the one proposed for the MC88110/MP add a process ID automatically to every message sent. At the receiving node the network interface compares the message process ID with the currently running process and traps to the kernel if the two do not match. This allows the kernel to either buffer the message or to initiate a context switch. While such a mechanism principally allows user-level message reception without coordinated scheduling, its main purpose is to simplify context switches by allowing the context switch to occur even if a number of messages are still in transit.

The requirement for coordinated scheduling is, in some sense, analogous to the coordinated memory management (also known as maintaining TLB consistency) that is mandatory in shared memory multiprocessors [Tel90]. In a shared memory architecture, the node sending a remote memory reference translates the virtual address issued by the processor into a physical address. The request message itself contains the physical address only and is not specific to any user process. This means that whenever the virtual to physical address translation is changed on a processor the operating system must propagate the change to all other nodes as well.

7.3 Summary

The initial motivation for Active Messages was to simply give parallel programming language implementations access to the network hardware on message passing machines without the traditional layers of software overheads. In the process of achieving this goal, Active Messages emerged as a new methodology to view communication in a multiprocessor. This dissertation develops this methodology to rationalize the integration of communication into the node architecture of a multiprocessor. The focus is on understanding how the hardware and software layers of abstraction interact such that the key communications issues can be addressed at the right level. The analysis of traditional systems (e.g., message passing, shared memory, message driven, and dataflow) concludes that these are akin to the high-level language instruction set architectures of the 70's and that an approach, analogous to RISC, predicated on the use of high-level languages with sophisticated compiler technology is called for.

Active Messages implementations on existing hardware provide simple communication primitives appropriate for code generation and decouple storage allocation and sophisticated scheduling from communication proper. This enables powerful compiler optimizations and results in a more efficient system overall. On a single platform, the CM-5, Active Messages supports message passing, message driven, dataflow, and NUMA shared memory programming models today as efficiently as more specialized hardware. Active Messages also provides a communication substrate which can be coupled with hardware support for a global address space to implement cache-coherent shared memory. A number of ongoing hardware developments use Active Messages as the basis and promise a dramatic improvement in communication performance. To date, the Active Messages communication architecture offers the most versatile and efficient avenue towards multiprocessors which support a full spectrum of parallel programming languages.