In Proceedings of Supercomputing'93, Portland, Oregon, November 15-19, 1993.

Concert — Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware

Vijay Karamcheti and Andrew Chien
University of Illinois at Urbana-Champaign
1304 West Springfield Ave., Urbana, IL 61801
{vijayk,achien}@cs.uiuc.edu

Abstract

Inefficient implementations of global namespaces, message passing, and thread scheduling on stock multicomputers have prevented concurrent object-oriented programming (COOP) languages from gaining widespread acceptance. Recognizing that the architectures of stock multicomputers impose a hierarchy of costs for these operations, we have described a runtime system which provides different versions of each primitive, exposing performance distinctions for optimization. We confirm the advantages of a cost-hierarchy based runtime system organization by showing a variation of two orders of magnitude in version costs for a CM5 implementation. Frequency measurements based on COOP application programs demonstrate that a 39% invocation cost reduction is feasible by simply selecting cheaper versions of runtime operations.

1 Introduction

Concurrent object-oriented programming (COOP) languages are an attractive approach for programming massively-parallel computers. Object encapsulation can be used to hide details of concurrency control and implementation, providing much needed modularity in concurrent programs. Fine-grained COOP languages are particularly promising since they expose a great deal of concurrency by allowing its expression at the object level. Numerous researchers have been exploring fine-grained object-oriented approaches [1, 2, 3, 4, 5].

Implementations of COOP languages face not only the challenges of object-oriented languages, but also several additional challenges related to concurrency and distribution.[1] In particular, COOP language implementations must provide a global object namespace, implement the communication necessary for remote method invocation, and schedule method invocations on objects. One reason why COOP languages have not gained widespread acceptance is the perception that these features cannot be implemented efficiently on multicomputers built from conventional processors (stock hardware). This claim is reinforced by the use of custom hardware in the highest performance implementations of COOP languages. Implementations of CST on the J-machine [6] and ABCL on the EM-4 [7] exploit hardware support for message passing, method dispatch, global name translation, and thread scheduling to achieve high performance.

Such custom hardware efforts focus on minimizing the cost of all runtime operations by supporting a few general-purpose primitives efficiently. This reduces the cost difference between situations; e.g., making the cost of accessing a remote object comparable to that of accessing a local object. This design philosophy works well for custom hardware where fundamental cost distinctions between different cases are small[2]. On the other hand, in stock hardware where cost distinctions are more pronounced, general-purpose implementations cannot be fast. Thus, this approach implies slowing down the common cases.

Our approach focuses on a runtime design which distinguishes between the expensive and inexpensive cases, thereby complementing efforts to improve overall execution efficiency of COOP languages on stock hardware. The rationale for this approach is the recognition that some cost differences (such as the local/remote dichotomy) are fundamental to parallel machines, particularly those based on stock hardware. These cost distinctions must be recognized and effectively managed to achieve efficient execution of fine-grained COOP languages. In this paper, we present the design of the Concert runtime which embodies this design philosophy. Our design provides several versions of each runtime operation with varying cost and functionality, allowing the compiler or programmer to choose the most efficient version. The Concert runtime also increases opportunities to use low-cost versions by supporting spec-

[1] For the purposes of this paper, we only consider implementations on distributed memory parallel machines.

[2] inter-node operations are faster because of closely coupled processor-network interfaces, while intra-node operations are slower because of slower clock rates, shorter pipelines, and fewer registers.

ulative compilation and runtime mechanisms such as group scheduling of computations.

An example Figure 1 illustrates the benefits of static selection of the cheapest appropriate version of method invocation. In the general case, an invocation from **A** to **B** requires several steps, due to concurrency and distribution concerns (see Figure 1(a)): *translation* to obtain the target object location from a global name, message *creation* to marshal arguments for remote delivery, *transfer* to send them to the target, and *scheduling* to enforce concurrency constraints and support blocking invocations. In a general implementation, the overhead of these steps are required even if **B** is on the same node as **A**.

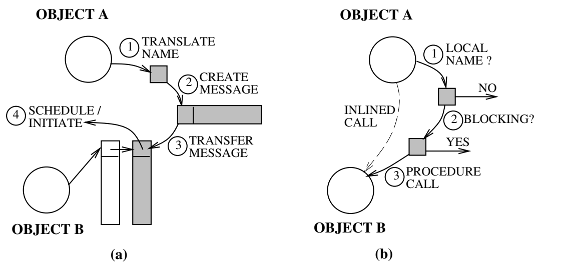

Figure 1: Invocations: (a) general-purpose, (b) with runtime checks and static selection.

The sequence shown with solid lines in Figure 1(b) uses runtime checks to reduce invocation cost. A check to ensure that **B** is located on the same node, followed by a check to ensure that the invocation is non-blocking allows the method to be invoked with a procedure call. A considerably cheaper invocation (dashed line in Figure 1(b)) is possible if the compiler knows about the location of **B** and the non-blocking nature of the handler. It could eliminate the runtime checks (which may have cost comparable to the actual invocation), and make use of global information not available at runtime (e.g., **B** will *always* remain on the same node) to statically inline the method code. Such static selection is particularly important for object-oriented languages which have high procedure call frequencies because it enables interprocedural optimizations.

The Concert runtime is part of the Concert project [8] whose goal is to achieve efficient and portable execution of concurrent object-oriented languages on a variety of hardware platforms. The Concert system focuses on execution grain size tuning relying on aggressive program analysis and careful information management at every stage from the compiler to the runtime system.

Overview The rest of the paper is organized as follows. Section 2 briefly describes a computational model for COOP languages and then describes the key components of the runtime interface. In Section 3, we use an example, method invocation, to illustrate the range of cost and functionality choices that are available to a compiler or programmer. Costs of runtime versions (based on a CM5 implementation) and analyses of optimization opportunities in application programs are presented in Section 4. Finally, Section 5 discusses the related work, and Section 6 summarizes the paper and the current status of the project.

2 The Concert runtime

This section surveys the design of the Concert runtime system. First, we briefly describe an abstract computation model which embodies characteristics common to many COOP languages proposed so far [2, 3, 4, 9, 5]. We then present the Concert runtime interface which provides the runtime operations required by the abstract model.

2.1 Computation model of COOP languages

COOP languages express computation as a set of autonomous communicating entities (objects). Generally, these objects reside in a globally shared object namespace and send method invocations to each other. Each invocation request involves the sending of a message containing the method-selector and invocation arguments to the target object. An invocation request is accepted and scheduled according to a language-specific policy which may depend upon local object state. Invocations accepted by the object result in method executions which may in turn send other invocation requests, create additional objects, and modify object state. Invocation requests can behave like remote procedure calls, requiring reply messages which are processed as invocations on the activation frame. A theoretical basis for reasoning about COOP languages can be found in [1]. A wide variety of COOP languages [2, 3, 4, 9, 5] can be modeled by specializing the message acceptance and scheduling policies of this abstract computation model.

2.2 Components of the Concert runtime interface

The dynamic nature of the abstract computation model and a distributed implementation make runtime support for the following operations essential.

- *name translation* to map the object global name to a physical memory location.
- *message delivery* to deliver the invocation requests,
- *type-dependent dispatch* to execute messages based on the object type and state,
- *storage management* to efficiently allocate and free storage for objects and messages, and
- *activation scheduling* to schedule the user computation for increased data locality.

Name translation is necessary for supporting object location and migration. Message delivery is essential for communicating method invocations between objects on different nodes. Type dependent dispatch is a staple of object-oriented languages, and storage management supports dynamic object creation. Activation scheduling can have a significant impact on locality and reuse of data in upper levels of the memory hierarchy.

The Concert runtime provides multiple versions of each runtime operation, exposing the fundamental performance distinctions for static selection of the version with the needed functionality. The runtime system modules and their usage interactions are shown in Figure 2. The overall structure has three levels: Level 1 provides a global namespace, communication mechanisms and local storage management; Level 2 provides resource management mechanisms and frameworks for speculative compilation; and Level 3 uses a combination of first level operations to provide a distributed object space, and support method invocation, object creation, and object migration.

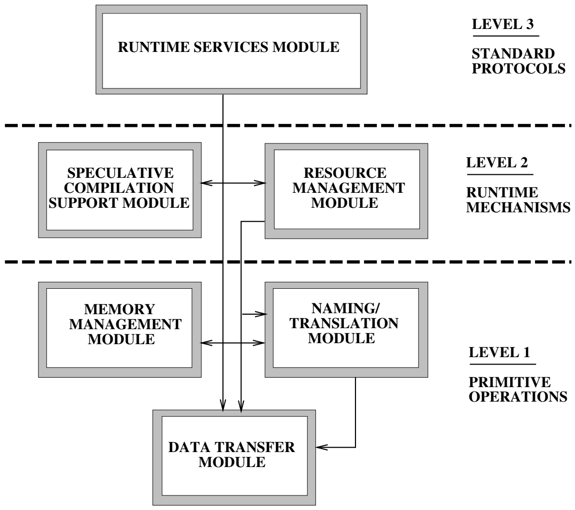

Figure 2: Components of the Runtime Interface.

The first level is composed of three modules:

- **Data Transfer Module** which delivers invocations,
- **Naming and Translation Module** which maps public names of objects to physical memory addresses, and
- **Memory Management Module** which allocates and frees local storage chunks.

The runtime provides several versions of each operation, each appropriate for a different situation. For example, the runtime provides two versions of translation, one for node-private names, the other involving public names which may require remote access. The versions expose the cost difference, allowing it to be managed intelligently.

Level 2 provides mechanisms whose goal is to utilize system resources more efficiently, and consists of

- **Resource Management Module** which provides mechanisms for load balancing and efficient utilization of local processor resources, and
- **Speculative Compilation Support Module** which provides runtime support for speculative compilation.

The speculative compilation support module provides three kinds of runtime primitives to facilitate speculative optimization. *Runtime checks* determine actual object properties at runtime, enabling the runtime selection of the cheapest version of an operation. *Assertions* extend the temporal and spatial scope of information about an object property and allow the compiler to generate code which tests for a desired object property (e.g., locality, which allows use of cheaper versions of runtime operations), assert that property, then execute a long sequence of code optimized based on the assertion (e.g., eliminating runtime checks, replacing general method invocations by procedure calls or inlined functions). Without assertions, to generate safe code, the compiler would have to assume that object properties could change at arbitrary times, limiting optimization. The runtime ensures that the assertion either remains true or if it becomes false, a special callback function is invoked to restore the code sequence to the general (and more costly) version. Our measurements show that assertions can payoff even for single method invocations. However, the cost of invalidating an assertion can be very high; thus, optimizations using assertions must be applied with discretion. *Hints* extend both checks and assertions by allowing the compiler to influence malleable object properties such as locality. Hints enable a compiler to communicate optimization assumptions to the runtime system by indicating that it would be beneficial if a particular property were true. Hints given to the runtime are non-binding; they typically guide runtime policies.

Level 3 implements a set of basic protocols for operations such as method invocation, object creation, and object migration. The **Runtime Services Module** provides several versions of these protocols by combining different versions of the Level 1 runtime operations.

The design of Concert runtime components described in this section is driven by cost hierarchies prevalent in stock multicomputers. The next section uses an example to illustrate this cost-hierarchy based design in more detail.

3 Cost and functionality choices for method invocation

In this section, we focus on a key example, *method invocation*, to illustrate the utility of providing several versions of runtime operators. Method invocation is the central operation in the execution of any object-oriented language. The basic steps *translate* the object public name to a physical memory address, *transfer* the method-selector and invocation arguments to the destination object, and *schedule and initiate* the computation by finding and invoking the appropriate handler. The overall cost depends upon several factors such as type and location of the target object, number of arguments, and the scheduling policy. In this section, we describe the key factors influencing cost of each step; each significant cost distinction is reflected in a new version of the operation.

3.1 Name translation

Name translation associates an object name with a $\langle$*node,address*$\rangle$ tuple giving location of the corresponding storage segment. The cost of translation depends on

- whether the object has a *public* or *private* name, and
- whether or not the object is *relocatable*.

Public names belong to a global namespace and can be used by other objects to send messages without knowing the actual object location, while *private* names are entries in a local namespace and must be qualified with node addressing information. Relocatable elements require translation which involves querying an associative structure; non-relocatable elements do not. Different translation versions (Figure 3) are available for each described situation.

Name translation is cheapest when no translation is required, intermediate cost when the translation is found locally, and maximum cost when a public name is looked up in a global associative table using one or more remote operations. Thus, the different versions correspond to the basic performance distinction between *local* and *remote* operations. This cost distinction is fundamental and arises from the difference in local and remote access times for multicomputer architectures which will persist and perhaps increase in the future.

The performance difference among the versions implies that an efficient implementation must exploit the cost hierarchy, using the cheapest appropriate names. For example, consider objects and activation frames (contexts). Typically, objects are globally accessible and relatively long-lived, requiring relocatability. On the other hand, activation frames are accessible to a small set of continuations and are relatively short-lived. Consequently, public names are

Situation	*Version*	*Description*
non-relocatable	`xlate_null`	–
relocatable, private name	`xlate_pvt`	private name $\rightarrow$ *address*
relocatable, public name	`xlate_pub`	public name $\rightarrow$ $\langle$*node,address*$\rangle$

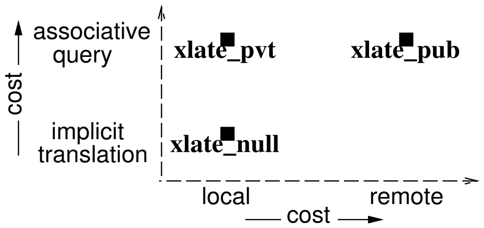

Figure 3: Versions of name translation.

appropriate for objects, while private names suffice for activation frames. As we will show in Section 4, such simple optimizations can give significant performance gains.

3.2 Data transfer

The data transfer operation delivers invocation arguments and reply values to target elements. The cost of data transfer depends on

- whether the transfer target is *local* or *remote*, and
- the *memory hierarchy level* containing the data at the source and destination.

If the data resides in different memory hierarchy levels at source and destination (e.g., the source data might be in registers but needs to reside in memory at the destination), additional work is required to move data up/down the memory hierarchy. Remote transfers cross processor boundaries and involve additional cost for marshalling local references in the outgoing data. Hardware communication primitives[3] may cause additional data movement up/down the memory hierarchy. Data transfer versions are listed in Figure 4.

Since most data transfers are accompanied by a handler invocation, all versions *integrate handler invocation with actual data transfer*. This integration reduces the overhead from data reception to dynamic method dispatching [12] and has been used in the J-machine [13] and the *active message* facility [14].

The different data transfer versions reflect two basic performance distinctions: *local* versus *remote* transfers, and *register* versus *memory* access costs. Remote transfers cost more than corresponding local transfers due to routing and network interface delays. Local memory transfers are more

[3]The CM5 [10] provides a register-to-register primitive, while the Paragon [11] provides a memory-to-memory transfer operation.

Situation	*Version*	*Description*
local destination	`lsend`	register → register
	`lsend_mem`	memory → memory
remote destination	`rsend`	register → register
	`rsend_mem`	memory → memory

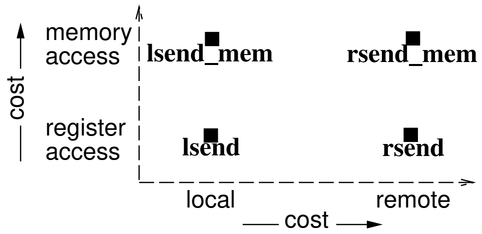

Figure 4: Versions of data transfer operation.

expensive than register movements because memory data must be moved to and from registers in load/store architectures. The cost difference between remote versions of register-to-register and memory-to-memory transfers depends on both access-cost differences and the hardware communication primitives. Based on technology trends, the above distinctions will be more pronounced in future systems: network latency will increase (in units of processor cycles) and memory hierarchies will get deeper with a larger cost to go from top to bottom.

The cost distinctions amongst different versions make it critical to choose an appropriate version for performance. For example, non-blocking invocations with few arguments should use `lsend` or `rsend`, since the small messages need not be buffered. On the other hand, blocking invocations should use memory-to-memory versions because the message must be buffered and integrating it with data transmission avoids having to copy arguments into a frame at the destination. Other examples include always preferring local versions over remote ones, and integrating handler invocation with data transfer to allow execution of a portion of the user method in the invoked handler. In Section 4, we show that the costs of different versions range over two orders of magnitude; thus, the above optimizations can yield sizeable performance gains.

3.3 Scheduling and initiation

The scheduling and initiation operations include all steps between data reception and execution of the user handler: *dispatch* binds selector to user code, *frame allocation* obtains an activation frame, and *scheduling* initiates the computation. The cost of these operations depends on

- the time of *binding* the selector to user code,
- whether or not the handler *blocks*, and
- the policy used for *scheduling* the computation.

Selectors can be bound to procedures at compile time or at runtime using a dispatch table based on type and object state information. A handler is potentially blocking if it may suspend awaiting the reply to an outstanding invocation. Non-blocking handler invocations can avoid the cost of frame allocation, using the system stack to execute immediately. Execution can be deferred and scheduled later later with a group of related computations, improving data locality but requiring argument storage and activation enqueuing. The different scheduling and initiation operations (shown in Figure 5) each capture a distinct situation.

Operation	*Version*	*Description*
dispatch	`s_bind`	static binding
	`d_bind`	delayed binding via dispatch table
frame allocation	`alloc_stack`	non-blocking method: stack allocation
	`alloc_frame`	blocking method: heap allocation
scheduling	`sched_now`	immediate execution
	`sched_ind`	delayed execution, indiv. scheduled
	`sched_grp`	delayed execution, group scheduled

Figure 5: Versions of scheduling and initiation.

The different runtime operations correspond to three basic performance distinctions: *compile-time* versus *runtime* binding, *stack* versus *heap* allocation, and *immediate* versus *delayed* execution. Moving work to compile time, as in static binding, reduces runtime cost. Stack allocating activation frames can reduce storage allocation costs. And, immediate execution avoids allocating storage to hold data until it can be processed. The cost of delayed execution points out a clear tradeoff between scheduling overhead and the efficient use of processor resources. As we will show in Section 4, the performance differences amongst the versions suggest that specialization is essential for high efficiency. Making the optimizations expressible in the runtime interface is a critical step in that direction.

Summary Method invocation in a distributed COOP implementation involves the composition of several runtime operations. In this section, we have described a range of invocation choices arising from fundamental cost distinctions, such as *local* and *remote* operations, *register* and *memory* accesses, *stack* and *heap* allocation, and *immediate* and *delayed* execution. In the next section, our measurements show the cost implications of different choices, and the performance advantages of making them explicit.

Operation	*Version*	*Implementation*	*Cost*	
			μs	*cyc.*
Name Translation	`xlate_null`	translation implicit in name.	0.03	1
	`xlate_pvt`	hashtable of associations.	0.91	30
	`xlate_pub`	distributed hashtable (with local caching). *min*	3.59	120
		max	16.07	536
		obs	11.70	390
Data Transfer (Local)	`lsend` reg→reg	register arguments of integrated invocation.	0.15	5
	`lsend_mem` mem→mem	argument copying into preallocated buffer.	2.88	96
Data Transfer (Remote)	`rsend` reg→reg	built with CMAM_4() from [15].	7.05	235
	`rsend_mem` mem→mem	built with CMAM() from [15].	15.65	520
Scheduling & Initiation	`s_bind`	known at compile time.	0.00	0
	`d_bind`	C++-like dispatch table.	0.09	3
	`alloc_stack`	modify stack pointer.	0.03	1
	`alloc_frame`	binned memory allocator.	3.72	124
	`sched_now`	frame setup and procedure call.	0.14	5
	`sched_ind`	computation stored in central scheduler queue.	3.27	108
	`sched_grp`	scheduling queue associated with object: all waiting computations scheduled together.	4.06	134

Table 1: Cost of different versions of runtime operations.

4 Implementation and performance results

The Concert runtime is currently being used as an execution environment for Concurrent Aggregates (CA) [3] programs on both the Thinking Machines CM5 [10] and a thread-based uniprocessor simulator. We use these implementations to measure runtime operation costs, and CA program statistics to conservatively estimate opportunities for using low cost versions of runtime operations. Performance measurements validate the existence of a cost hierarchy for each operation, justifying its reflection in the runtime interface. Dynamic statistics show that less expensive versions can be used in many cases, dramatically reducing effective cost of runtime operations. We are constructing a compiler which will automatically optimize execution by selecting cheaper versions of runtime operations [8].

4.1 Costs of primitive runtime operations

The costs of various runtime operations are given in Table 1. Operation latency in μseconds and CM5 processor cycles was measured using a synthetic program which repeatedly performed the runtime operation of interest. Accuracy was ensured by measuring over an interval 10^5 times larger than the 1μsecond timer resolution. The actual CM5 hardware used was a 33 Mhz clock, 32 Mbytes per node, 128 PN partition with the node network interface supporting five 32-bit word messages. Note that the cost of operations involving remote access is dominated by software overhead (the hardware network latency is small in all cases).

Based on the data in Table 1, we observe a clear cost hierarchy amongst versions for each operation. In the following sections, we make more specific observations.

Name translation `xlate_pub` is implemented using a globally distributed hash table with local caching. The three timings for `xlate_pub` correspond to the minimum cost of finding the translation in the local cache, the maximum cost when translation requires a remote access, and the observed cost based on CA application programs. Our performance data shows that translation cost spans several orders of magnitude from essentially free for non-relocatable private names to tens of μseconds for an access to the globally distributed hash table. Thus, using low cost versions for translation yields significant performance gains.

Data transfer All measurements for data transfer versions use a transfer size of four 32-bit words. Table 1 shows that for local invocations, register-to-register transfer is an order of magnitude cheaper than the memory-to-memory alternative; most of the difference is attributable to buffer management. For remote invocations, register-to-register transfer is two times cheaper than the memory-to-memory

Implementation	*Clock rate MHz*	*Send/Reply*		*Dispatch*		*Translation*	
		μs	*cyc.*	μs	*cyc.*	μs	*cyc.*
CST(on J-m/c)	28.3	6.36/7.8	180/220	0.6/1.6	18/46	0.9/2.6	26/74
ABCL(on EM-4)	12.5	9	112	0.24	3	–	–

Table 2: Min/Max costs of runtime operations on custom-hardware implementations.

alternative. Clearly, register-to-register versions give significant benefits for small transfers; however memory-to-memory transfers are required for large amounts of data which cannot be buffered in registers. In many machines, transfer cost increases with message size, and both the CM5 [10] and AP1000 [16] incorporate special mechanisms for sending small messages.

Scheduling and initiation Dynamic binding using C++ like dispatch tables [17] costs only a few instructions more than static binding. However, the cost of frame allocation for blocking invocations is significant implying that non-blocking invocations are worth dealing with specially. Both sophisticated forms of scheduling, individual and group, are much more expensive than the simple `sched_now` approach, suggesting that such approaches must give significant locality benefits to be worthwhile.

Our measurements of runtime operation costs present a different profile from that of COOP language implementations on custom hardware (see Table 2). The numbers show that such implementations provide relatively flat cost hierarchies, with no more than a factor of three difference in cost; thus, choosing the most general mechanism incurs only a modest performance penalty. In contrast, operation costs in the stock-hardware based Concert runtime span two orders of magnitude, making the choice of the cheapest appropriate mechanism essential for good performance.

4.2 Method invocation cost

To demonstrate the performance impact of choosing cheaper versions of runtime operations, we consider a range of implementations for the method invocation operation. Table 3 presents costs of some typical method invocation sequences. In all cases, the method invocation takes two arguments, a selector, and a continuation.[4] As in Section 4.1, for operations involving `xlate_pub`, three numbers corresponding to the minimum, maximum, and observed timings are presented. The table shows that the *cheapest invocation sequence (**4** cycles) is as inexpensive as a procedure call* in an imperative language. The most expensive sequence (**1275** cycles) is three orders of magnitudes more expensive, and corresponds to a naive implementation of a fine-grained COOP language. The different situations presented in Table 3 bring out the cost-functionality trade-offs associated with providing features like heap-allocated activation frames (increases cost by **150** cycles), sophisticated scheduling (incurs cost of **120** cycles), and relocatable namespaces (cost ranges from **140** to **450** cycles).

Table 3 also shows the costs of corresponding operations in the ABCL/onAP1000 implementation [18] which focuses on improving the worst-case efficiency of runtime primitives. The numbers show the Concert runtime to be as fast as the ABCL/onAP1000 implementation for comparable invocation sequences, implying that the observed cost-hierarchy is not merely an artifact of poor implementation of the worst-case version. The order of magnitude cost difference for the cheapest invocation sequence (**4** cycles versus **58** cycles) validates our claim about traditional runtime designs charging for unnecessary functionality. Some Concert operations are more expensive only because they provide additional functionality such as relocatable namespaces, larger message sizes, and flexible scheduling.

4.3 Opportunities for optimization

Higher performance by capitalizing on low-cost versions of runtime operations is possible only if there are opportunities to use the cheaper versions. To demonstrate the extent to which the cost distinctions are exploitable, we present frequency measurements of optimization opportunities for a set of CA applications [3], and characterize the effective reduction in costs of runtime operations. The applications include **pcbroute** which uses concurrent A^* search to route nets around obstacles on a board, **logicsim** which is an event-driven logic simulation, and **multigrid** which implements the multigrid relaxation technique.

In the following discussion, effective reduction in an operation's cost is computed by weighting it with the fraction of times an optimizable event occurs. Percentage reductions in the effective cost of an operation are presented relative to the most general (expensive) version of the operation in Tables 1 and 3. For operations involving `xlate_pub`, the baseline used is the observed cost.

Low-cost versions of name translation Global names and remote translation are unnecessary in many cases. To estimate opportunities for using cheaper versions, Table 4 presents the message target statistics for each application.

[4] While this number of arguments is small, it allows us to contrast invocation sequences with and without hardware supported communication. The new CM5 network interface supports larger messages.

Description of Sequence	*Versions Used*	*Cost*			
		Concert		ABCL/AP1000	
		µs	*cyc.*	*µs*	*cyc.*
non-susp. local invoke, reg→reg transfer, static binding, scheduled immed.	`xlate_null, lsend, s_bind, alloc_stack, sched_now`	0.1	4	2.3	58
susp. local invoke, reg→reg transfer, dynamic binding, scheduled later	`xlate_null, lsend, d_bind, alloc_frame, sched_ind`	8.4	277	9.6	240
non-susp. remote invoke, reg→reg transfer, dynamic binding, scheduled immed.	`xlate_null, rsend, d_bind, alloc_stack, sched_now`	7.7	252	8.9	223
		min–max (obs)	*min–max (obs)*		
remote translation, non-susp. remote invoke, reg→reg transfer, dynamic binding, scheduled immed.	`xlate_pub, rsend, d_bind, alloc_stack, sched_now`	11.9–27.1 (21.4)	397–904 (713)		
remote translation, susp. remote invoke, reg→reg transfer, dynamic binding, scheduled immed.	`xlate_pub, rsend, d_bind, alloc_frame, sched_now`	16.3–31.8 (25.9)	543–1053 (865)		
remote translation, susp. remote invoke, mem→mem transfer, dynamic binding, scheduled later	`xlate_pub, rsend_mem, d_bind, alloc_frame, sched_ind`	28.6–44.2 (38.3)	941–1456 (1275)		

Table 3: A wide range of method invocation costs.

Two kinds of optimizations reduce the cost of translation. The first optimization associates private names with activation frames (see Section 3.1) allowing `xlate_pvt` to be used for all reply messages. Replies constitute over **40%** of all messages; consequently, this reduction from the baseline `xlate_pub` (11.7μs) to `xlate_pvt` (0.91μs) will reduce translation cost by **37%**.

program name	*total msgs*	*reply msgs*	*invoke msgs to i^{th} neighbor*		
			1	*2*	*3*
`pcbroute`	116,883	47%	14%	5%	3%
`logicsim`	219,788	43%	24%	15%	7%
`multigrid`	582,913	41%	19%	11%	6%

Table 4: Message target statistics in CA programs.

The second optimization reduces the cost seen by invocation messages. From Table 4, we observe that applications demonstrate significant invocation locality (i.e., objects tend to communicate with a small set of preferred objects) with **35%** of *all* messages being invocations from an object to one of its three most preferred neighbors. Translation costs can be reduced by exploiting this locality to cache translations to preferred neighbors. Since invocations are separate from replies, reducing the cost for `xlate_pub` from 11.70μs to its minimum possible (3.59μs) yields an additional translation cost reduction of **24%**.

Low-cost versions of data transfer On machines such as the CM5 which supports hardware transfer mechanisms for small messages, data transfer cost can be significantly reduced by using appropriate versions for such transfers. To evaluate the impact of this optimization, Table 5 presents the distribution of message sizes in each application. Approximately **50%** of the messages are four words or less, so they can benefit from the cheaper register-to-register versions. Even with the conservative assumption that all message destinations are remote (else `lsend` can be used), this optimization reduces the cost of data transfer by **28%**.

program name	*total msgs*	*msgs of size (words)*			
		1–4	*5–9*	*10–14*	*15+*
`pcbroute`	116,883	58%	27%	11%	4%
`logicsim`	219,788	47%	28%	13%	12%
`multigrid`	582,913	44%	41%	14%	1%

Table 5: Message size distribution in CA programs.

program name	*invocations*		
	total	*# non-blocking*	*% non-blocking*
`pcbroute`	61525	36289	59%
`logicsim`	18764	9756	52%
`multigrid`	34440	14012	41%

Table 6: Invocation statistics in CA programs.

Low-cost scheduling and initiation versions Table 6 presents the fraction of invocations which are non-blocking. Such invocations comprise **50%** of the total and can be implemented using (i) stack allocation (using `alloc_stack` instead of `alloc_frame`), and (ii) by executing a portion of the user computation in the integrated handler (using

Optimization	*Version Enabled*	*optimiz. cases*	*cost redn. (primitive)*	*cost redn. (invocation)*
reply msgs to activation frames	`xlate_pvt`	40%	37%	11.5%
invoke msgs to preferred objects	`xlate_pub`(min)	35%	24%	7.5%
short (*1–5* words) msg transfers	`rsend`	50%	28%	11.6%
non-suspending invocations	`alloc_stack`	50%	29%	5.0%
	`sched_now`	50%	19%	3.3%
Total reduction in cost of method invocation				38.9%

Table 7: Reductions in invocation cost by using lower-cost versions.

`sched_now`). Using the program stack for the activation frame reduces the scheduling and initiation cost by **29%**. A further reduction of **19%** is possible by starting user-handler execution immediately.

Table 7 summarizes the cost reductions described in this section by applying them to a composite method invocation operation. An overall improvement of at least **39%** is possible from optimizations which choose appropriately specialized versions of runtime operations. It is interesting to note that all the basic runtime operations contribute to the overall reduction, suggesting that hierarchical interfaces are necessary for all of them. Further performance gains are possible by exploiting resource-management mechanisms and support for speculative compilation to proactively structure the computation, increasing optimization opportunities. As an example, consider group scheduling of computations to benefit from state reuse. Table 8 presents a conservative estimate of the number of activations that can be deferred and group scheduled. This number has been obtained by processing application traces and counting the unordered activations on an object. This number is remarkably high, **40%** – **70%** of all invocations can be group scheduled, indicating the potential for large performance gains.

program name	*Scheduling Quanta*		
	# indiv. scheduled	*# group scheduled*	*% redn.*
`pcbroute`	119262	66692	44%
`logicsim`	36877	15460	58%
`multigrid`	53972	13816	74%

Table 8: Scheduling statistics in CA programs.

5 Related work

Our work is related to many efforts focusing on runtime support for efficiently executing irregular computations on stock hardware [19, 20]. It differs from runtime systems for coarse-grained object-oriented languages such as COOL [21] and Mentat [22], by focusing on fine-grained object-level concurrency.

The ABCL/onAP1000 implementation [18] is most similar with our work but adopts a traditional design, emphasizing techniques for reducing access latency and object-creation costs. The general versions in Concert are as efficient as in the ABCL system, while the cheaper versions cost significantly less. Our work complements the ABCL work by distinguishing low-cost operations from expensive ones, exposing the distinctions for optimization.

Our work is also similar to the Threaded Abstract Machine(TAM) [23]. Although TAM supports a different language model (extended hybrid dataflow), both systems enable optimization by providing the compiler with a range of mechanisms for managing synchronization, scheduling and storage and an associated cost hierarchy. The speculative compilation module in the Concert runtime reflects additional support not present in TAM, required because of the dynamic nature of object-oriented languages which limits the applicability of static analysis and optimization.

6 Conclusion

We have described a novel approach for efficiently executing COOP languages on stock multicomputers. The key is a runtime interface which exposes the essential performance distinctions between different versions of a runtime operation, allowing their use to be optimized. Since these cost distinctions driving the runtime design arise from basic architectural structures such as the local/remote dichotomy, and deep memory hierarchies, we believe them to be fundamental; consequently, the only path to higher efficiency is to choose cheap versions, decreasing the frequency of expensive ones.

Timing measurements on a CM5 based runtime implementation bear out our claim of cost distinctions between different versions of primitive operations in a method invocation. Expensive versions have cost comparable to previously published results, while inexpensive ones cost significantly less. The runtime differs from special-hardware implementations in providing versions with costs spanning two orders of magnitude; clearly, the ability to choose the cheapest version is essential for good performance. Frequency measurements on CA applications reveal ample opportunity for efficient exploitation of cheaper versions of runtime operations. Based on a few simple optimizations, effective invocation cost reductions of at least **39%** has been shown possible.

Current efforts focus on continued tuning of the implementation. We are integrating this runtime interface into the Concert system along with compilation techniques to exploit the available hierarchy of runtime operators.

Acknowledgements

The authors would like to thank John Plevyak for participating in several discussions and assisting with the runtime system implementation, and the anonymous reviewers for their helpful comments.

The Concert project and the authors of this paper are supported in part by NSF grant CCR-9209336, ONR grant N00014-92-J-1961, NASA grant NAG 1-613, and a special-purpose grant from the AT&T Foundation. Some of the work reported in this paper was done on a CM5 at the National Center for Supercomputing Applications.

References

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: MIT Press, 1986.

[2] P. America, "POOL-T: A parallel object-oriented language," in *Object-Oriented Concurrent Programming* (A. Yonezawa and M. Tokoro, eds.), pp. 199–220, MIT Press, 1987.

[3] A. A. Chien, *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. Cambridge, MA: MIT Press, 1993.

[4] W. Dally and A. Chien, "Object oriented concurrent programming in cst," in *Proceedings of the Third Conference on Hypercube Computers*, (Pasadena, California), pp. 434–9, SIAM, 1988.

[5] A. Yonezawa, ed., *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990. ISBN 0-262-24029-7.

[6] W. Horwat, "Concurrent smalltalk on the message-driven processor," Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1989.

[7] M. Yasugi, S. Matsuoka, and A. Yonezawa, "ABCL/onEM-4: A new software/hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer," in *Proceedings of the ACM Conference on Supercomputing '92*, 1992.

[8] A. Chien, V. Karamcheti, and J. Plevyak, "The concert system – compiler and runtime support for efficient fine-grained concurrent object-oriented programs," Tech. Rep. UIUCDCS-R-93-1815, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1993.

[9] C. Houck and G. Agha, "HAL: A high-level actor language and its distributed implementation," in *Proceedings of the 21st International Conference on Parallel Processing*, (St. Charles, IL), pp. 158–165, August 1992.

[10] Thinking Machines Corporation, Cambridge, Massachusetts, *Connection Machine CM-5, Technical Summary*, November 1992.

[11] Intel Corporation, *Paragon XP/S Product Overview*, 1991.

[12] T. Baba, et al., "A parallel object-oriented total architecture: A-NET," in *Proceedings of IEEE Supercomputing '90*, pp. 276–285, IEEE Computer Society, 1990.

[13] W. J. Dally, A. Chien, S. Fiske, W. Horwat, J. Keen, M. Larivee, R. Lethin, P. Nuth, S. Wills, P. Carrick, and G. Fyler, "The J-Machine: A fine-grain concurrent computer," in *Information Processing 89, Proceedings of the IFIP Congress*, pp. 1147–1153, Aug. 1989.

[14] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active Messages: a mechanism for integrated communication and computation," in *Proceedings of the International Symposium on Computer Architecture*, 1992.

[15] T. von Eicken and D. E. Culler, *Building Communication Paradigms with the CM-5 Active Message layer (CMAM)*. University of California, Berkeley, 2.4 ed., September 1992.

[16] T. Shimizu, T. Horie, and H. Ishihata, "Low-latency message communication support for the AP1000," in *International Symposium on Computer Architecture*, pp. 288–297, 1992.

[17] B. Stroustrup, *The C++ Programming Language*. Addison Wesley, second ed., 1991.

[18] K. Taura, S. Matsuoka, and A. Yonezawa, "An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers," in *Proceedings of the Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1993.

[19] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren, "Early experiences with olden," in *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Machines*, 1993.

[20] W. C. Hsieh, P. Wang, and W. E. Weihl, "Computation migration: Enhancing locality for distributed-memory parallel systems," in *Proceedings of the Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, pp. 239–248, 1993.

[21] R. Chandra, A. Gupta, and J. L. Hennessy, "Data locality and load balancing in COOL," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[22] A. Grimshaw, "Easy-to-use object-oriented parallel processing with Mentat," *IEEE Computer*, May 1993.

[23] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek, "Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages an Operating Systems*, pp. 164–75, 1991.