Performance Evaluation of the Sylvan Multiprocessor Architecture

F. J. Burkowski
C. L. A. Clarke
S. C. Cowan
G. J. Vreugdenhil

Multiprocessor Systems Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1

sylvan@watmsg.waterloo.edu

Abstract

Sylvan is a multiprocessor system that provides lightweight tasks with full address space protection. The primary design goal of the system is to make tasks viable as a scalable unit of concurrent abstraction. Lightweight tasks are supported by a tasking coprocessor that provides fast context switching and low-latency inter-task communication and synchronization. With fast context switching and communications available, programs can be designed using many concurrent tasks so they can be distributed across multiple processors, yet still run efficiently on a small number of processors. This paper describes the architecture of the Sylvan system and details performance measurements made on the existing single-node four-processor system. Based on these performance measurements, we discuss the value of the architectural features of the Sylvan system, and describe some additional architectural features which would further enhance performance. These architectural features are evaluated quantitatively where possible.

1 Introduction

A network of workstations contains considerable parallel processing power that is not presently being exploited. To address this, several distributed operating systems have been developed that provide an environment suitable to developing parallel distributed applications composed of communicating tasks [10, 21, 17]. Performance, particularly the latency of inter-task communications, is essential to efficent exploitation of the processing resources. These systems provide excellent remote communications latency, and may even be approaching the theoretical limits of their networks. Local communications performance, however, is correctly perceived by developers to be too slow relative to that which is possible in a shared memory environment, and so are dropped in favour of a shared memory model when parallel applications are being developed. Unfortunately, shared memory does not easily or efficently scale up to be implemented across multiple machines on a network.

To resolve this conflict between scalability and local performance, Sylvan provides hardware support for local inter-task communications, reducing the cost of a request to a server to approximately the cost of a subroutine call. With fast context switching and communications available, programs can be designed using many concurrent tasks so they can be distributed across multiple processors, yet still run efficiently on a small number of processors.

Sylvan is a multiprocessor system that provides hardware support for "tasking" operations, such as communications and scheduling, to provide tasks that are both "lightweight" and exist in separate address spaces. Tasks are considered to be "lightweight" if an inter-task communication cycle occurs with latency close to that of a procedure call. Lightweight tasking is realized by supplying a simple core of hardware services which extend standard CPU facilities to encompass task scheduling, context switching, communication, and synchronization. Operating-system services use this hardware core as the basis for their implementation.

A Sylvan task has its own context and thread of execution. When a task is created, it is assigned a unique Task IDentifier (TID) and a protected virtual address space. Tasks communicate among themselves by way of *synchronous message passing*[9]. Groups of tasks are used to provide operating system services using a technique known as *multiprocess structuring.*

Sylvan tasks are intended to be used as a unit of abstraction, hence the emphasis on low-latency tasking operations. Tasks are a concurrent abstraction; there are no concurrency issues within a task, all synchronization is done using the tasking primitives provided. Tasks are a scalable abstraction, because the semantics of the tasking primitives are independent of the proximity of the tasks. No special capabilities are granted to physically nearby tasks; all such optimizations are embodied within the tasking kernel.

The concepts of synchronous message passing and multiprocess structuring were first developed in the Thoth operating system [9], and have proven their value in numerous projects and systems [2, 10, 15, 21, 30, 17, 33]. These projects demonstrate that operating systems may be developed in a natural fashion using a simple tasking kernel as a basis. Synchronous message passing and multiprocess structuring have also been proven to scale for use in multiprocessor and distributed systems [5, 26, 8, 21, 17]. The simplicity and utility of these concepts make them an appropriate basis for the tasking model provided by the Sylvan system. Synchronous message passing has the further advantage of mapping directly onto the model of synchronization and communication used in many concurrent programming languages, including Ada [31], Concurrent C [13], and concurrent C++ [4]. Refer to Gentleman [14] or Cheriton et al. [9] for a detailed description of synchronous message passing and multiprocess structuring.

Sylvan uses a shared bus architecture, in which multiple microprocessors connect to the system bus. At the heart of the architecture is a tasking coprocessor referred to as the *Taskmaster* (or TM). The Taskmaster implements a simple kernel which performs task scheduling and message queuing, and directs all context switching and task interactions.

This hardware-based approach is similar to that taken by Roos [27], however Roos does not share our aim of supporting general purpose computing. Roos focussed specifically on the support of Ada tasking on a single processor, where address space protection is not needed. Ramachandran et al. [23] share our aim of supporting general-purpose computing, but they did not address the problems associated with providing address space protection and virtual memory; additionally, they failed to achieve comparable levels of performance.

The task model provided by the Sylvan system is described in the next section. Section 3 describes the Sylvan architecture and section 4 provides performance measurements. Section 5 discusses and quantifies the benefits of existing and proposed architectural features that enhance tasking performance. Section 6 compares the performance of Sylvan to similar systems, and section 7 presents our conclusions.

2 The Sylvan Task Model

A Sylvan task consists of a single thread of execution in a separate protected address space. The unique TID associated with each task allows the rapid direction of a message to a task. The separate protected address space provides inter-task protection, while the following communications primitives provide fast, safe inter-task communications.

There are three communication primitives which are fundamental to the system:

Send specifies a bounded length message to be transmitted to a specified receiving task. The sending task is blocked waiting for a reply message from the receiving task.

Receive is issued by a task wishing to accept a message. If a message has already been sent to this task, the message and the sender's identity are returned to the receiver. If no message has been transmitted, the receiver is blocked until a message is sent.

Reply is issued by the receiving task after a successful **Receive**. **Reply** transmits a reply message back to the sender and unblocks it. The replying task then continues its execution.

The **Send** primitive requires the sending task to specify a *send queue* number identifying one of 16 queues associated with the receiving task. A send queues effectively allows a type, or name, to be associated with an incoming messages. A mask may be specified with the **Receive** primitive which allows a task to receive messages from a subset of the queues. This facility, called *message screening*, allows a task to avoid receiving messages which are of a type which it is temporarily unable to handle. Incoming messages are queued in FIFO order at the specified send queue.

These primitives are similar to those that effect *rendezvous* in Ada. **Send**, **Receive**, and **Reply** correspond to Ada's entry call, **accept**, and **end accept** respectively. A send queue roughly corresponds to an entry of an Ada task. Unlike Ada, these primitives do not suffer from an inability to cope with local delay [18], because replies may be issued in arbitrary order.

A subset of a processor's general purpose registers is used for exchanging data with other tasks. A message to be transferred to another task is loaded into these registers and a request is issued to the Taskmaster. These same message registers are used when receiving incoming messages. Presently eight general purpose registers form this message register set, making the maximum message length 32 bytes. Experience with synchronous message passing has shown this to be sufficient for most interactions [10, 3].

For communicating larger blocks of data, sylvan provides the **Copy_To** and **Copy_From** primitives. When task A (say) has sent a message to task B, and B has received the message but not replied, then B may copy data either to or from A's address space. This facility is not provided by the Taskmaster, but by the memory management subsystem, which is itself a group of tasks.

Interrupt handling is easily integrated into this communication and synchronization model by mapping interrupts into messages. The **Install_Handler** primitive allows a task to specify a function which is to be called whenever a particular interrupt occurs. When called, the function has access to device registers but cannot access the task's address space. The function is responsible for removing the source of the interrupt and may return up to four bytes of data. This data is then translated into a message and sent to the task which installed the function.

The Sylvan task model does *not* include asynchronous message passing, task migration or shared memory. Asynchronous message passing and task migration are features that frequently lead to slower performance of the tasking primitives[28, 8], and thus threaten the use of tasks as a unit of abstraction. Shared memory among tasks is very fast when the tasks are co-resident, but very slow, or totally unavailable, when they are not. As such, programs that depend on shared memory become non-scalable, threatening the use of tasks as a scalable abstraction.

3 The Sylvan Architecture

3.1 Hardware Description

As illustrated in Figure 1, a Sylvan node contains one tasking coprocessor (the Taskmaster), and several processor complexes. In the existing implementation there are four processor complexes, each of which consists of a MC68030-based Motorola VM04 VERSABus module and a 4 MB VM13 memory module. The processors communicate with the Taskmaster through the MC68030 coprocessor interface, and invoke Sylvan primitives by executing coprocessor instructions that refer to the Taskmaster. The resulting VERSABus accesses are serialized by the bus arbiter. The VERSABus is only used for coprocessor invocations and message transfers between the processors. The more frequent instruction fetches and data accesses for each processor take place on a private bus (the RAMBus) that exists between each VM04 and its local VM13 memory. This assures a processor of fast access to its local memory. The VM13 memory is dual ported so that data may be transferred over the VERSABus when explicitly requested.

The Taskmaster is a bipolar processor based on the AMD 29116 register ALU and the AMD 2910 microsequencer, and was designed as a microcode engine to run the Sylvan kernel exclusively. The execution unit has a load/store architecture: There are 32 16-bit general purpose registers, and all operations except load and store take only registers as operands. Since it is not intended as a general purpose CPU, the Taskmaster has no interrupt or exception handling hardware.

The Taskmaster contains 16 KB of fast static RAM used for internal data structures. Analysis of the Taskmaster microcode [11] shows that two-thirds of the references to local RAM are to fields within a *task descriptor*. A task descriptor is a contiguous 16-word structure aligned to a 16-word boundary, and is used to contain data related to a single task. To allow fast access to fixed fields within a task descriptor, the Taskmaster has a register displacement addressing mode with 4-bit displacement. Since task descriptors are aligned, no addition is used to compute the effective address. Register displacement mode and a register indirect addressing mode are the only addressing modes supported by the Taskmaster.

3.2 Kernel Data Structures and Algorithms

The data structures necessary to perform task scheduling and inter-task communications are stored in task descriptors in the Taskmaster's local RAM. The data structures for context switching and message buffering (one message per task) are stored in *shadow task de-*

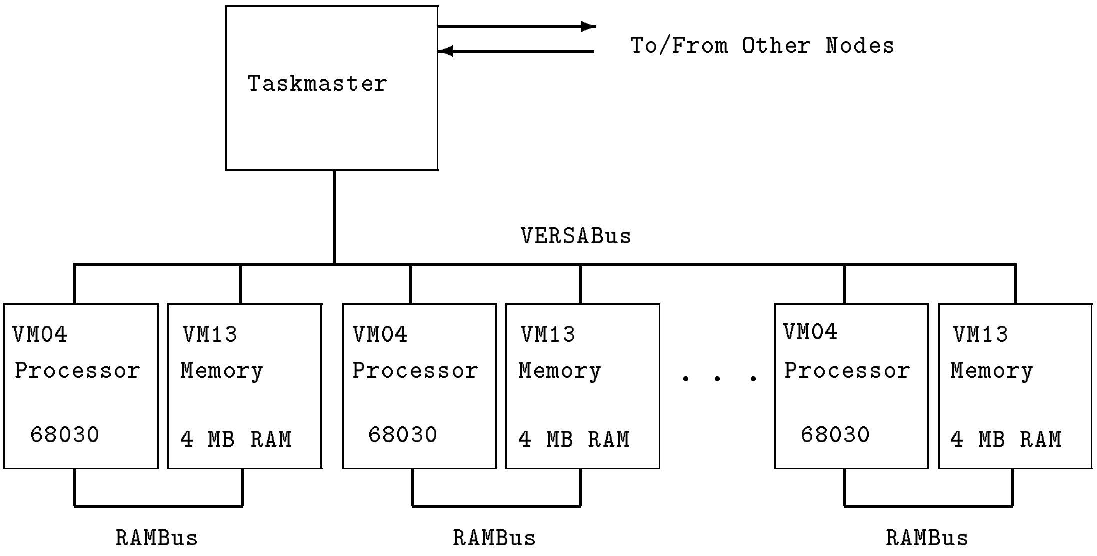

Figure 1: A Sylvan Node

scriptors [32] located in the memory of the processor complex where the task resides. Partitioning the data structures in this way lowers the cost of the message passing operations. Placing the scheduling information close to the scheduler, and the task context information close to the applications processor, means that hardware units have access to the appropriate data without bus contention.

The processing of a Sylvan primitive involves the following four phases:

1. Read the parameters from the processor registers.
2. Compute state transitions.
3. Return results to the processor registers.
4. Invoke a software *context handler* on the processor.

A set of these context handlers reside on each processor complex. During the fourth phase, the Taskmaster points the invoked context handler at one or two *shadow task descriptors*. These shadow task descriptors each contain a message buffer for their respective tasks, and a pointer to the address translation table for the virtual address space of the task. The handlers do not contain any conditional branches; all decisions are made ahead of time by the Taskmaster when it selects the appropriate handler. The handlers are small, stateless assembly language routines, and have no context to be loaded or saved. Consequently, the handlers are much faster than corresponding entries into a monolithic kernel (which requires a switch on both entry and exit), and are amenable to implementation in hardware.

Context handlers take the place of hardware support for fast context switching on the processors themselves. Such support could take the form of multiple context sets within the processor: multiple banks of registers, and TID-tagging on the cache and TLB entries. We will discuss these ideas further in section 5.

To achieve the smallest possible context switching time with the hardware available, we provide a large number of these handlers. The selection of a context handler specifies one **save** operation, one **action**, and one **load** operation, as follows:

- A **Save** operation specifies the location where the message registers are to be written; one of:

None	no saving necessary
Mine	save message in this task's shadow task descriptor
Other	save message in some other task's shadow task descriptor

- An **Action** specifies the type of action to perform; one of:

Switch	the current task has blocked, context switch to another task
Continue	continue executing the current task

- A **Load** operation designates the location from which to load the message registers; one of:

None	no loading necessary
Mine	load message from this task's shadow task descriptor
Other	load message from some other task's shadow task descriptor

In addition to the general context handlers described above, there are special context handlers that exploit scheduling opportunities:

Idle	there are no other ready tasks; idle the processor
Wake	if same task then resumes, no context switch required
Switch Direct	pass message directly in registers instead of through shadow task descriptors

.

The **Idle** and **Wake** handler are applicable only in lightly loaded configurations; the **Switch Direct** handler is frequently applicable.

4 Performance

4.1 Inter-task Communication Performance

In the algorithms that we have developed, the time overhead of inter-task communication is always bounded above by a constant [5]. The observed time, however, can actually vary by as much as a factor of three, depending on circumstances such as whether

```
Task A:
  ttid <- Get_TID(Task B)
  loop forever
    Send(ttid, message)
  endloop
endtask A

Task B:
  loop forever
    ttid <- Receive(message_buffer)
    Reply(ttid, reply_message)
  endloop
endtask B
```

Figure 2: A Pair of Communicating Tasks

the communicating tasks are on the same or different processors, and whether other tasks are ready to run. Here we examine a number of canonical situations that may arise in the execution of a Sylvan program, and detail the time overhead of the tasking operations used.

Measurements accurate to one tenth of a μsecond were made by tracing the bus transactions between the processors and the Taskmaster on the VERSAbus using a logic probe. Thus the precise time of initiation and completion of all Taskmaster interactions, and trans-processor message movements, can be measured. Behavior of context handlers internal to a single processor are not visible; their times are inferred from the latency to the next Taskmaster request.

To place the following tasking measurements in context, we examined the cost of a subroutine call and return. The subroutine had a 32-byte parameter, and returned a 32-byte result. The latency of the subroutine call and return was 18.1 μseconds.

We first examine a test that has two communicating tasks on a single processor with no other ready tasks. Table 1 presents a time analysis for this benchmark, and Figure 2 provides a pseudo-code listing of the test. Each entry in Table 1 gives the time taken by the specified primitive or context handler. Taskmaster primitive entries have been further broken down into times for reading input parameters, computing state changes, and writing out results to the processor.

Task A sends a message to task B, which invokes the first context handler to save the message from reg-

Processor(s)	Primitive	Handler	Cost μ seconds	% of Total
CPU & TM	A sends to B		14.3	10.51
CPU		save mine switch load none	46.0	33.82
CPU & TM	B receives from A		10.1	7.43
CPU		save none continue load other	19.6	14.41
CPU & TM	B replies to A		13.9	10.22
CPU		save none switch load none	32.1	23.60
		Total Cycle Time	136.0	100.00

Table 1: Communicating Tasks on the Same Processor

isters into A's shadow task descriptor. Task B then executes a **Receive** primitive, which accepts the message from a queue, and uses a context handler to read the message from A's shadow task descriptor into registers. Task B then replies to task A, invoking the **Switch Direct** special context handler. This handler switches directly to task A, leaving the message entirely in registers. Since the last context handler does no work other than to change context from one task to another, we can regard its run time of 32.1 μseconds as the time cost of a context switch. Similarly, we can use the difference in run time between the first and third handlers to derive the cost of saving a message to registers as 13.9 μseconds. Most importantly, we notice that 64.2 μseconds or 47% of the total time is spent changing contexts.

We next examine two communicating tasks on different processors. This test is identical to the first test, except that Task A and Task B execute on different processors. Table 2 presents the timing breakdown of this test. With two active processors, some operations take place concurrently. The figures under Critical Path are the contribution of each operation to the total latency overhead, while the figures under Unit Cost represent the latency of the operation on that processor. The Concurrent Operation entry represents the difference between the critical path time, and the sum of the unit costs (latencies) of the operations.

In this test, there is only one task on each processor, so when a task blocks the Taskmaster does not cause a context switch, but instead puts the processor into an idle state (using the **Idle** handler). When the task becomes ready again, the scheduler notices that it is the same task that was running previously, and so again avoids the cost of context switching (using the **Wake** handler). Because the handlers involved in this test never cause context switches, both their unit latency and critical path times are very low, making this the fastest test possible. Breaking down the latency of this test into categories, we find that 43.4 μseconds or 44% was spent by the Taskmaster scheduling tasks, while 54.5 μseconds or 56% was spent in handlers that do little more than transfer messages.

The third test is identical to the second, except that a lower-priority background task running an infinite loop is present on each processor. This illustrates the message passing cycle time in a heavily loaded setting (each processor is CPU-bound). Every time one task sends or replies to the other, the Taskmaster detects that a task with higher priority than the task

Processor(s)	Primitive	Handler	Critical Path μ seconds	Unit Cost μ seconds	% of Total
CPU 0 & TM	A sends to B		18.1	18.1	18.5
CPU 0		save other idle load none	16.0	16.0	16.3
CPU 1		save none wake load mine	12.0	28.0	12.3
CPU 1 & TM	B replies to A		16.3	16.3	16.6
CPU 1		save other continue load none	14.0	17.7	14.3
CPU 0		save none wake load mine	12.5	26.5	12.8
CPU 1 & TM	B receives and blocks		9.0	12.2	9.2
CPU 1		save none idle load none	0.0	16.0	0.0
		Cycle Time	97.9	150.8	100.0
		Concurrent Operation	52.9		54.0

Table 2: Communicating Tasks on Different Processors

currently running has become ready, and interrupts that processor. This interrupt causes the processor to execute a **Reschedule** primitive. The Taskmaster gives a destination context to the processor during a Reschedule, and a context switch is performed.

Table 3 illustrates the timing breakdown of this test. Again, with two active processors, many operations take place concurrently, so their full cost is not reflected in the Critical Path column. The latency of the cycle time has risen because of the need to interrupt and reschedule a processor on each **Send** and **Reply**. The cost of the context handlers has increased; 131.6 μseconds (50%) of the critical path is spent interrupting and rescheduling processors. This cost is an obvious byproduct of pre-emptive scheduling and involuntary context switching; however, during part of this time, the other processor is running the background task, achieving useful work. Under more realistic circumstances, the communicating tasks would do some local computation, giving the background task more time to compute before being interrupted, presumably furthering some other task, and preserving the overall throughput of the system.

4.2 Interrupt Response

The test shown in Figure 3 was used to measure the latency of translating interrupts into messages. This test is of particular significance, because it is this mechanism that can be used to respond to outside events such as user key strokes, or the arrival of a disk block. Interrupt processing in traditional systems is typically highly restricted, slow, or both. The Sylvan interrupt handling mechanism is both fast and fully integrated into our tasking model.

This test measures the amount of time between

Processor(s)	Primitive	Handler	Critical Path μ seconds	Unit Cost μ seconds	% of Total
CPU 0 & TM	A sends to B		21.8	21.8	8.4
CPU 0		save other switch load mine	0.0	22.8	0.0
CPU 1		Interrupt	40.3	40.3	15.4
CPU 1 & TM	Reschedule		16.2	16.2	6.2
CPU 1		save mine switch load mine	48.3	48.3	18.5
CPU 1 & TM	B replies to A		19.4	19.4	7.4
CPU 1		save other continue load none	0.0	28.9	0.0
CPU 0		Interrupt	39.1	39.1	15.0
CPU 1 & TM	B receives and blocks		8.7	13.3	3.3
CPU 0 & TM	Reschedule		14.2	14.2	5.4
CPU 0		save mine switch load mine	52.9	52.9	20.3
		Cycle Time	260.9	317.2	100.0
		Concurrent Operation	56.3		21.6

Table 3: Communicating Tasks on Different Processors with a Background Task

an interrupt event and the start of execution of the appropriate task. Delivery of the interrupt notification message unblocks and schedules this task. During the test, a background task is executing. Table 4 illustrates the chronological components of this test. Interrupt handling breaks down into 43.7 μseconds (39%) for actual interrupt processing, 32.5 μseconds (29%) to reschedule the processor, and 36.1 μseconds (32%) to context switch to the receiver task.

With a total interrupt translation time of 112.3 μseconds, a single processor can handle over 4000 interrupts per second, with more than 50% of CPU time still available for real work. This interrupt rate is sufficient to support 4 serial lines running at 9600 baud, or disk blocks arriving at 4000 blocks per second continuously.

5 Feature Analysis

Using a coprocessor to execute the tasking kernel provides two first order benefits:

- State save and restore on every entry and exit of the operating system is no longer needed.
- The kernel is the only program running on the Taskmaster so all data structures are ready in registers and local static RAM.

These benefits may be measured by comparing a tasking kernel that does not use a coprocessor to a kernel that does. Vadura [32] provides a description of a software only implementation of a kernel that does not provide separate address spaces, but is otherwise similar to the Sylvan kernel. In Burkowski et al. [5],

Processor(s)	Primitive	Handler	TM Phase	Cost μseconds	% of Primitive	% of Total
CPU		interrupt		43.7	100.00	38.9
CPU & TM	Reschedule		input	4.6	14.15	4.1
			compute	9.2	28.31	
			output	18.7	57.54	
			total	32.5	100.00	29.0
CPU		save mine switch load none		36.1	100.00	32.1
			Response Time	112.3		100.0

Table 4: Interrupt Response Timing

```
Task A:
  /* Install interrupt handling
     function. */
  Install_Handler(vector, handler)

  /* Receive interrupt messages. */
  loop forever
    itid <- Receive(message_buffer)
  endloop
endtask

/* Interrupt handling function. */
handler:
  return(data)
endfunction

/* Background task
   (runs at low priority). */
Task B:
  loop forever
endtask
```

Figure 3: Interrupt Response Test

a description is given of a prototype implementation of a Taskmaster-assisted Sylvan kernel that does not provide separate address spaces. Both were executed on Motorola VM04 processors.

The software only kernel of Vadura achieved a message passing cycle time of 232 μseconds; the Taskmaster-assisted kernel of Burkowski et al. achieved a message passing cycle of 65 μseconds. Not all of the improved performance, however, can be attributed simply to using a coprocessor, many other factors contribute. In the following sections, we detail these factors. We discuss the benefits of the architectural features included in the Sylvan system, and estimate the benefits of future features that may further enhance tasking performance.

5.1 Taskmaster-specific Features

The Taskmaster's instruction set architecture contains many standard features that contribute to its performance. These were detailed in section 3.1. Performance can be further enhanced by exploiting some simplifying assumptions that can be made.

Much of the difficulty in supporting long pipelines stems from the need for precise exceptions, low-latency interrupt response, and from instructions of different lengths[22, Chapter 6]. Since the Taskmaster is not a general purpose processor, we can exploit the following properties:

- The Taskmaster does not have to support precise exceptions because it never suffers any exceptions.

- The Taskmaster simply responds to requests in FIFO order, it need never be interrupted.
- No floating point is needed for the kernel; all instructions can be of uniform length.

Thus issuing pipelined and superscalar instructions is greatly simplified, and can be used to a much greater extent without checkpointing. The current Taskmaster implementation is not pipelined, but does have nearly uniform instruction lengths, and supports a limited form of superscalar instruction issue; the ALU and instruction sequencer can operate in parallel. The Taskmaster achieves a static clocks-per-instruction ratio of 0.916.

5.2 Context Handlers

It is difficult to measure the general improvements due to the use of the context handlers since there is no concrete piece of software with which to compare. Our context handlers are between 5 and 17 instructions long, and average 13 instructions, which is much shorter than the context switching path in conventional operating systems, and is as close as possible to the direct hardware support that we would like.

However, we can measure the benefits due to special handlers exploiting scheduling opportunities. The major difference between test 2 and test 3 is that the Taskmaster is prevented from using the **Idle** and **Wake** handlers described in section 3. Considering test 2 as an optimization of test 3, we see that the **Idle/Wake** handlers reduce the message cycle latency from 260.9 μseconds to 97.9 μseconds, an improvement of 166%. A similar comparison between test 1 and one in which the **Switch Direct** handler is defeated shows a performance improvement of 26%. Thus a decrease in context manipulation can result in a large improvement in overall performance.

5.3 Coprocessor Interface

While a significant degree of parallelism is observed in the tests that involve more than one processor (tests 2 and 3), no parallelism at all exists in the single processor tests. This is because the Motorola coprocessor interface does not provide a low-cost asynchronous invocation mechanism of the form:

1. Make request.
2. Do work (save context).
3. Await result (new context to switch to).

Using such a coprocessor interface, context saving could proceed in parallel with scheduling computations in the Taskmaster, hiding the cost of scheduling behind the cost of state saving in operations that lead to context switches. Table 5 illustrates the cost of Taskmaster scheduling computation (Taskmaster Compute Time), showing that the cost of scheduling in the Taskmaster is 13% of the total message cycle time. Reducing this cost to 0 by performing it in parallel would produce an improvement of 15%.

5.4 Dedicated Taskmaster

Much of the time in conversation with the Taskmaster is spent passing parameters back and forth between the Taskmaster and the processor. This might suggest that the Taskmaster is too far away, and should be closer to the processors; dedicated to a single processor. Dedicating a Taskmaster to a single processor has two positive effects and two negative effects. The positive effects are: (1) contention for the Taskmaster and the shared bus is eliminated, and (2) the cost of Taskmaster I/O is reduced. The negative effects are: (1) the single-node, multiple CPU case, which Sylvan makes very fast, disappears, and (2) Taskmaster utilization is reduced.

Table 6 illustrates the cost of parameter passing between the Taskmaster and the processor in single processor tests. For these tests, the cost of Taskmaster I/O averages 13%, and eliminating this cost would produce an improvement of 15%.

5.5 Multithreaded Processors

The purpose of the context handlers is to assist the Taskmaster in context switching the processors as quickly as possible. A better solution would be to provide hardware support for fast context switching in the processor itself, such as that prototyped in the Denelcor HEP[29]. Some current RISC processors, such as the AMD 29000[16] and the Sun SPARC[12], provide support for multiple contexts in the form of multiple register banks, and TID-tagging of cache and TLB entries. The APRIL project at MIT[1] is investigating the use of multiple register banks in just this way using a modified SPARC processor.

Tests: Singe CPU Only		TM Compute Time	TM Time	Message Cycle Time
One Pair of Tasks	Time in μsec.	18.1	38.3	136
	Compute Cost %	100	47	13
Two Pairs of Tasks	Time in μsec.	44.4	113.8	343.3
	Compute Cost %	100	39	13

Table 5: Compute Time Candidates for Concurrent Context Saving

Tests: Single CPU Only		TM-CPU I/O	TM Time	Message Cycle Time
One Pair of Tasks	Time in μsec.	16.8	38.3	136
	I/O Cost %	100	44	12
Two Pairs of Tasks	Time in μsec.	48.4	113.8	343.3
	I/O Cost %	100	43	14

Table 6: I/O Cost Fraction of Taskmaster Time

An innovative possibility for multi-threaded processors attached to a Taskmaster is for the Taskmaster to provide a *next context* TID to the processor ahead of time. When a blocking primitive is executed, the processor can switch to the next task immediately, without waiting for the Taskmaster to decide who should go next, effectively pipelining the entire scheduling computation. This restricts the system to scheduling algorithms that are unaffected by the current primitive, i.e. a single level of priority, with a circular queue of ready tasks. For cases where a higher level of priority is essential, the Taskmaster could resort to the interrupt/**Reschedule** technique used when a higher priority task has become ready on a processor that has not issued a primitive.

6 Relative Performance

In this section, we briefly compare the performance of Sylvan to similar systems. Sylvan is most similar to the V system[10, 8], since both were derived from Cheriton's Thoth. Like Sylvan, V provides a tasking model in which tasks are given a separate address space, and communicate by synchronous message passing. In [8], Cheriton reports a local message passing cycle time of 480 μseconds for a 32-byte message, in contrast to Sylvan's 136 μsecond message cycle time for the same message size.

The Amoeba system[21] is a minimal tasking kernel similar to V System, but also has capability and object oriented software layered in and on top of the tasking facility. The local remote procedure call time (semantically identical to a message passing cycle) is 800 μseconds, again in contrast to Sylvan's time of 136 μseconds.

In [24, 25], Ramachandran et al. describe a proposal and simulations for a system very similar to that originally proposed for Sylvan in 1985[7, 6]. Development is presumed to be concurrent, since no reference is made to the earlier Sylvan proposal. Ramachandran presents two significant systems (labeled II and III) running on Motorola VERSAmodule Monoboard Microcomputers[20] (an 8 MHz MC68000 version of the VM04). System II is essentially similar to Sylvan in its hardware makeup, and system III describes a (simulated) system with a custom bus enhanced to support queue manipulations. System II is reported to have an SRR latency of 5598 μseconds, and system III a latency of 3912 μseconds.

While the 68000 processors used are approximately three times slower than Sylvan's 68030 processors, the performance figures are 30 times slower than the Sylvan times. Furthermore, the performance of these systems is five times poorer than that of the V system reported in[10] on comparable 68000 processors, without hardware assistance. Rather than enhancing tasking performance, Ramachandran's hardware support seems to have *slowed down* tasking operations. On examining the internal structure of Ramachandran's system, we discover that some of this cost is due to his decision to place *all* data for task descriptors in a memory that is shared between the processor and his tasking coprocessor. Thus the critically important scheduling and context switching information is far away from both the processor and the scheduler, slowing down all tasking operations by 17%.

Recently, two new efforts have been undertaken to further reduce the cost of inter-task communications through software-only systems. In [19], Massalin and Pu describe the Synthesis Kernel, which dynamically generates small fragments of code to perform specific tasking operations, similar to Sylvan's context handlers. The performance achieved by Synthesis is excellent (under 20 μseconds for a context switch), but has not incorporated multiple processors or virtual memory, which add to complexity and degrade latency. Massalin and Pu also do not discuss the semantics of inter-task communications facilities, so a direct comparison to the message passing cycle cannot be made.

In [3], Bershad et al describe a mechanism called "Lightweight Remote Procedure Call." LRPC is said to be a combination of the "heavy-weight" semantics of remote procedure call (message passing) between separate threads, and the lighter-weight semantics of capability-protected simple procedure call. The essential aspect of normal RPC that has been removed for performance reasons is the existance of a separate thread for servers. This simplification has lead to performance gains: the null LRPC call costs 127-157 μseconds (depending on optimizations). However, the LRPC semantics sacrifice the existance of the server's thread, essentially rendering a LRPC server as a monitor.

7 Conclusion

The Sylvan system provides tasks that are viable as a unit of abstraction by enhancing the latency of tasking operations above what is normally possible. Using such tasks as units of abstraction allows modular systems to be constructed that are both responsive to external input (interrupts), and are scalable. Responsiveness is preserved by integrating the interrupt mechanism into the low latency tasking operations. Scalability is preserved by providing semantics in tasking primitives that are independent of proximity.

8 Acknowledgments

The authors would like to express their sincere appreciation to Dennis Vadura for the time which he spent helping us with the details of this paper. We would also like to thank Gordon Cormack, John Rogers and Dermot Harriss for their comments and suggestions.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. *Computer Architecture News*, 18(2):104–114, June 1990. Proc. 17th International Symposium on Computer Architecture.

[2] Richard J. Beach, John C. Beatty, Kellogg S. Booth, Darlene A. Plebon, and Eugene L. Fiume. The Message is the Medium: Multiprocess Structuring of an Interactive Paint Program. *Computer Graphics*, 16(3):277–287, July 1982.

[3] Brian N Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. In *Proceedings of the Symposium on Operating Systems Principles*, 1989.

[4] P. A. Buhr, Glen Ditchfield, and C. R. Zarnke. Adding concurrency to a statically type-safe object-oriented programming language. *SIGPLAN Notices*, 24(4):18–21, April 1989. *Proc. ACM SIGPLAN Workshop on Object-Based Concurrent Programming.*

[5] F. J. Burkowski, G. V. Cormack, and G. D. P. Dueck. Architectural Support for Synchronous Task Communication. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–53, Boston, MA, April 1989.

[6] F. J. Burkowski, G. V. Cormack, J. D. Dyment, and J. K. Pachl. A Message-Based Architecture for High Concurrency. In *Proc. 1st Conference on Hypercube Multiprocessors*, pages 27–37, Knoxville, Tenessee, August 1985.

[7] F. J. Burkowski and D. Dyment. Sylvan: A Message-Based Multiprocessor Architecture. In *Proc. 18th Hawaii International Conference On Systems Sciences*, pages 31–42, Honolulu, January 1985.

[8] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[9] David R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager. Thoth, A Portable Real-Time Operating System. *Communications of the ACM*, 22(2):105–115, February 1979.

[10] David R. Cheriton and W. Zwaenepoel. The Distributed V Kernel and its Performance for Diskless Workstations. In *Proc. 9th ACM Symp. Operating Systems Principles*, pages 128–140, October 1983.

[11] S. Crispin Cowan. Realization and Analysis of the Sylvan Kernel. Master's thesis, University of Waterloo, 1990.

[12] R. Garner, A. Agarwal, F. Briggs, E. Brown, D. Hough, B. Joy, S. Kleiman, S. Munchnik, M. Namjoo, D. Patterson, J. Pendleton, and R. Tuck. Scaleable Processor Architecture (SPARC). In *COMPCON, IEEE*, San Francisco, California, 1988.

[13] N. H. Gehani and W. D. Roome. Concurrent C. *Software — Practice and Experience*, 16(9):821–844, September 1986.

[14] W. Morven Gentleman. Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept. *Software — Practice and Experience*, 11:435–466, 1981.

[15] W. Morven Gentleman. Using the Harmony Operating System. Technical Report NRCC No. 23030, National Research Council of Canada, December 1983.

[16] Mike Johnson. *Am29000 Streamlined Instruction Processor*. Advanced Micro Devices, 1988.

[17] Frank Kolnick. *The QNX Operating System*. Basis Computer Systems, Inc., Willowdale, Ont., 1989.

[18] B. Liskov, M. Herlihy, and L. Gilbert. Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing. In *13th Annual ACM Symp. on Principles of Programming Languages*, pages 150–159, June 1986.

[19] Henry Massalin and Calton Pu. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the Symposium on Operating Systems Principles*, 1989.

[20] Motorola, Inc. *VERSAmodule Monoboard Microcomputer User's Guide*, 1982.

[21] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba — A distributed Operating System for the 1990's. *IEEE Computer*, 23(5), May 1990.

[22] David A. Patterson and John L. Hennessy. *Computer Arcitecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, first edition, 1990.

[23] Umakishore Ramachandran, Marvin Solomon, and Mary Vernon. Hardware Support for Interprocess Communication. *Computer Architecture News*, 15(2):178–188, June 1987. Proc. 14th International Symposium on Computer Architecture.

[24] Umakishore Ramachandran, Marvin Solomon, and Mary Vernon. Hardware Support for Interprocess Communication - II. Technical Report GIT-ICS-88/13, Georgia Institute of Technology, 1988.

[25] Umakishore Ramachandran, Marvin Solomon, and Mary Vernon. Hardware Support for Interprocess Communication. *IEEE Transactions on Parallel and Distributed Systems*, 1(3), July 1990.

[26] John W. Rogers. Distributing Sylvan: Distributed Kernel Primitives for a Fast Message-Passing Multiprocessor. Master's thesis, University of Waterloo, 1990.

[27] J. Roos. A Real-Time Support Processor for Ada Tasking. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–171, Boston, MA, April 1989.

[28] M. L. Scott and A. L. Cox. An Emperical Study of Message-Passing Overhead. In *7th International Conference on Distributed Computing Systems*, pages 536–543, Berlin, W. Germany, September 1987.

[29] Burton J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the International Conference on Parallel Processing*, 1978.

[30] F. Tuÿnman and L. O. Hertzberger. A Distributed Real-Time Operating System. *Software — Practice and Experience*, 16(5), May 1986.

[31] United States Department of Defence. *Reference Manual for the Ada Programming Language ANSI/MIL-STD-1815A-1983*. United States Department of Defence, February 1983.

[32] Dennis Vadura. A Proposed Design for the Sylvan Kernel. Master's thesis, University of Waterloo, 1985.

[33] Waterloo Microsystems, Inc., Waterloo, Ont. *Waterloo Port User's Guide*, 1984.