AN IMPLEMENTATION OF
MULTIPROCESSOR PATH PASCAL

BY

BRIAN JOHN HAFNER

B.S., University of Illinois, 1986

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois

CHAPTER 1.

INTRODUCTION

Path Pascal [Kols84] is a non-preemptive concurrent computer language. It is a superset of Berkeley Pascal [Joy84] with additional constructs for specifying processes, data–encapsulating objects, and path expressions [Camp76] which synchronize processes with respect to objects. Objects are similar in concept to monitors [Deit84], except that the number of processes allowed in an object is not limited to one, but rather is controlled by the path expressions.

This thesis describes an implementation of a multiprocessor Path Pascal compiler. Multiprocessor Path Pascal differs from the previous single processor implementation [Grun85] because it allows Path Pascal processes to truly execute in parallel. The single processor implementation simulates parallelism by context switching among the Path Pascal processes; however, it never executes more than one Path Pascal process at a time. Throughout the thesis, the terms "multiprocessor" and "single processor" refer to the executables produced by the compilers; both compilers use a single processor during compilation.

There are three primary goals for the multiprocessor implementation. First, it should demonstrate scalable performance as the number of processors is increased. Second, it should preserve the semantics used by the single processor version. Finally, the multiprocessor implementation should share as much code as possible with the single processor implementation.

Chapter two begins by describing the overall organization of the Path Pascal compiler for both the single processor and multiprocessor implementations. Since the major task in constructing the multiprocessor implementation is rewriting the runtime library, chapter two also outlines the runtime library for the single processor implementation. This outline may be used as a basis for comparing the single processor and multiprocessor runtime libraries.

The core of the thesis comprises chapters three and four. Chapter three presents the original design for the multiprocessor runtime library while chapter four details the actual implementation. Design plans documented in chapter three were modified when experience gained during implementation suggested better methods.

The fifth chapter evaluates the multiprocessor implementation based on data collected from testing several Path Pascal programs. The single processor implementation serves as a point of reference for this evaluation.

The summary reexamines the project goals in light of the completed implementation in order to measure the success of the project. A list of future research topics relevant to multiprocessor Path Pascal is also provided. Finally, appendix one presents an outline of the steps necessary to port either the single processor or the multiprocessor Path Pascal compilers to new architectures.

CHAPTER 2.

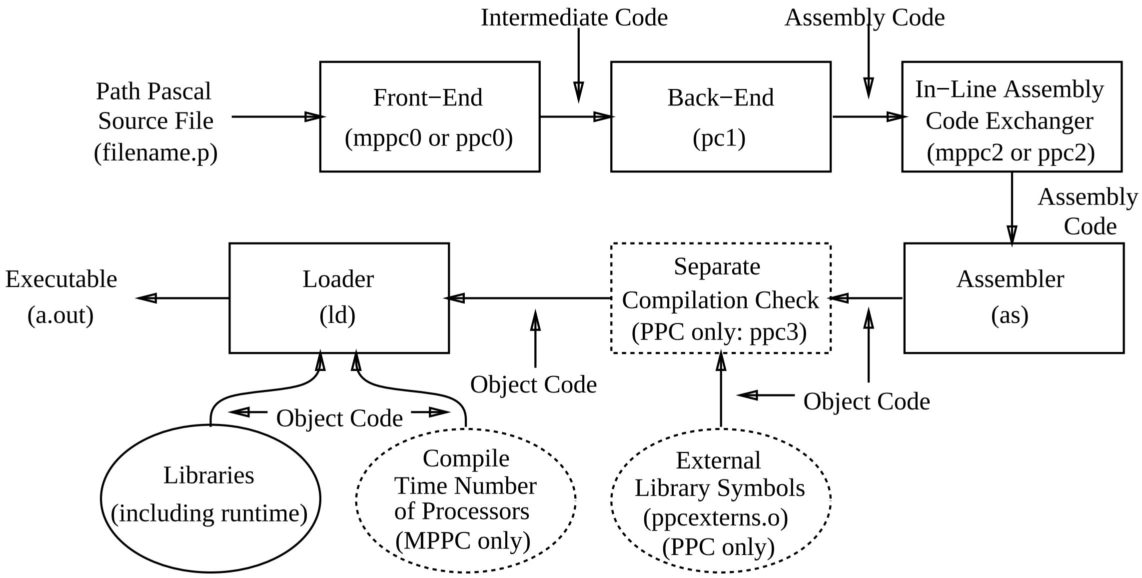
Overview of the Path Pascal Compiler

This chapter provides an overview of the entire Path Pascal compiler. The first section describes the compiler components for both the single processor and multiprocessor implementations. The second section details the single processor runtime system. Since the remaining chapters deal exclusively with the runtime library for the multiprocessor implementation, the second section provides a basis of comparison for those chapters.

2.1. Path Pascal Compiler Overview

The single processor and the multiprocessor compilers for Path Pascal share a similar overall structure shown in figure 1. The solid boxes and solid ovals denote compiler components common to both implementations while the dashed symbols indicate components which are particular to one of the implementations. Boxes denote compiler components and ovals represent external data. The compiler driver programs, ppc for the single processor version, and mppc for the multiprocessor version, execute the compiler components and handle the input and output along the entire chain.

The front-end translates Path Pascal programs into an intermediate form [Kess83, John84]. There are two significant differences between the multiprocessor front-end, mppc0, and the single processor front-end, ppc0. First, the multiprocessor front-end emits shared memory assembler directives to insure that global variables can be accessed by any processor. Second, the multiprocessor front-end has an additional field in its sema-



Path Pascal Compiler Structure

Figure 1

phore structure to store a lock that protects the semaphore from simultaneous access by multiple processors. To adapt the single processor front-end into the multiprocessor front-end requires modifying approximately 34 lines of code spread over 7 files.

The back-end, pc1, takes intermediate code generated by the front-end and produces assembly code for the target architecture. The back-end is identical for both Path Pascal implementations and for the Berkeley Pascal compiler.

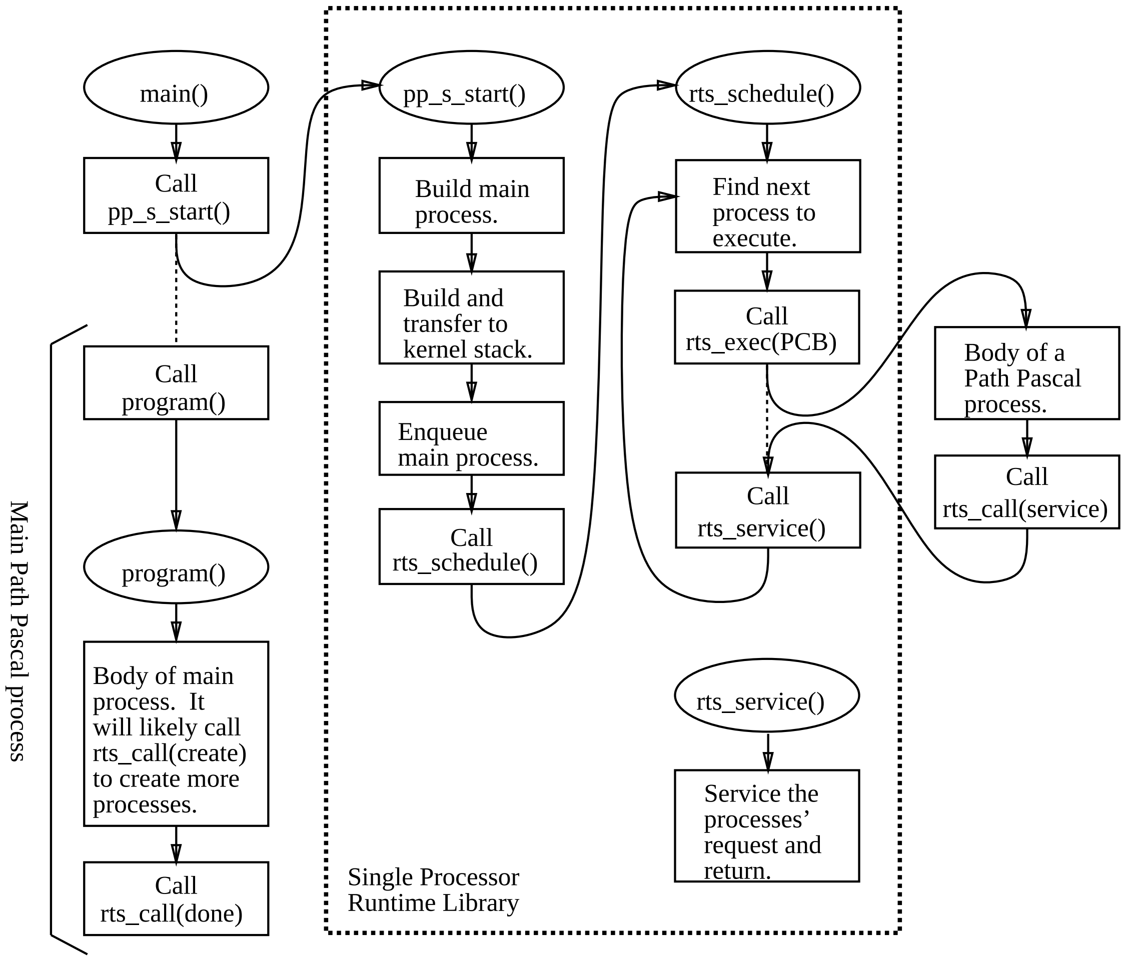
The in-line assembly code exchanger receives assembly code produced by the back-end and substitutes assembly code for certain subprogram calls. For Path Pascal, substitutions are made for calls that request services from the runtime system. For Berkeley Pascal, substitutions are made for simple library calls that can be executed more efficiently in-line. The Path Pascal implementations can also benefit from having simple, standard library functions executed in-line. This was done for the Vax and MC68000 single processor versions of Path Pascal but, because of time constraints, not for the Sequent Symmetry and Sequent Balance single processor implementations nor for the the current multiprocessor implementation. The in-line assembly code exchanger for the single processor implementation, ppc2, and the multiprocessor implementation, mppc2, differ by approximately 85 lines of code.

Following assembly into object code, the single processor compiler uses ppc3 to verify that files compiled separately are consistent with one another and the libraries. Since the multiprocessor implementation is experimental, this consistency check is not currently performed. However, it is anticipated that adding the separate compilation audit to the multiprocessor implementation would be straightforward since the existing single processor version contains no machine dependent code.

The last step in the compilation process is loading the object files and libraries together into an executable. Here, the multiprocessor implementation loads an object file unneeded by the single processor implementation. This additional object module defines the default number of processors to use if such a value has been specified on the compiler command line.

2.2. Single Processor Runtime Library Overview

The structure of the single processor runtime implementation and the modules it interacts with are shown in figure 2. In the figure, the elements of the runtime library are surrounded by a heavy dashed line; lightly dashed lines connect adjacent code segments where



Single Processor Runtime System

Figure 2

the lower segment is invoked via a context switch between the runtime system and a Path Pascal process. Ovals represent procedure entry points and boxes describe code segments. Arrows show flow-of-control.

The two routines: *rts_exec()* and *rts_call()* are not detailed in the diagram. They perform context switches between the runtime system and Path Pascal processes. *Rts_exec()* is called by the runtime system to execute a Path Pascal process and *rts_call()* is invoked by a Path Pascal process to request a runtime service. The services supported by the single processor implementation are: create process, delay process, enqueue process on a semaphore, dequeue process off a semaphore, suspend process, and destroy process.

As an example, consider the sequence of events at program start-up time. The main program calls *pp_s_start()* to initialize the runtime system. After *pp_s_start()* has built the process control block (PCB) for the main process and enqueued it on the ready queue, it calls *rts_schedule()*. The scheduler looks for a process to execute and finds only one, the main process. The scheduler executes the main process by invoking *rts_exec()*, sending the PCB of the main process as the only parameter. The main process then executes until it needs some kernel service and calls *rts_call()* with the number of the requested service and any other parameters required to complete the service call. The call to *rts_call()* returns control to the scheduler which then calls *rts_service()* to handle the process request. After the request has been handled and *rts_service()* has returned, the scheduler again looks for a process to execute and the cycle repeats. Essentially, the Path Pascal runtime system and the currently executing Path Pascal process behave as coroutines [Andr83].

CHAPTER 3.

Design of the Multiprocessor Runtime System

This chapter documents the design phase of the Path Pascal multiprocessor runtime system. An effort was made to keep the original design straightforward to accelerate the process of building a working prototype. An organized working prototype allows complex improvements to be made quickly and efficiently.

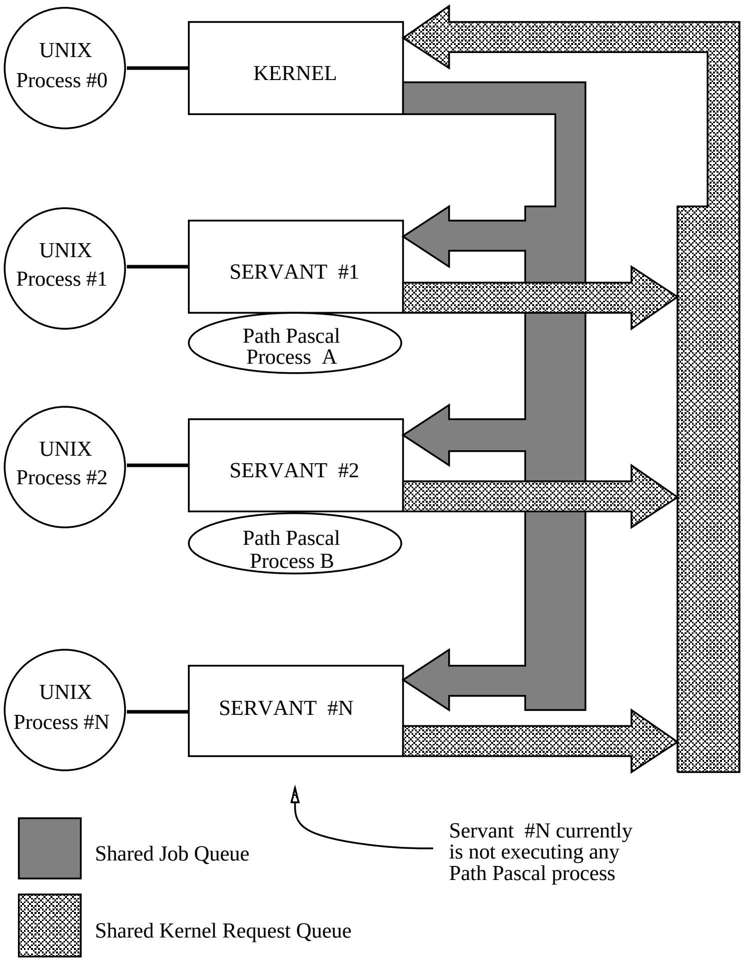
The chapter is divided into the following five sections: section one overviews the multiprocessor runtime system; section two explains the kernel/servant interactions; section three surveys the parallel support provided by the host multiprocessor operating system; section four enumerates the services provided by the multiprocessor runtime system; and the last section explains the principle used to allocate shared memory.

3.1. Overview of the Multiprocessor Runtime System

A high-level representation of the multiprocessor runtime system is shown in figure 3. It consists of the kernel process, a set of one or more servant processes, and the two shared memory queues that interconnect them.

The kernel, the heart of the runtime system, and the servant processes are heavy UNIX®[1] processes intended to execute in parallel on a shared memory multiprocessor computer. The actual level of parallelism exhibited by these UNIX processes depends on several

[1]UNIX® is a registered trademark of AT&T



Multiprocessor Runtime System Overview

Figure 3

factors including the operating system of the host computer and its current process load. For instance, if the operating system does not allow a subset of available processors to be devoted for exclusive use by the multiprocessor runtime system, other processes running on the multiprocessor system may preempt the kernel or servant processes and prevent the highest potential level of parallelism from being achieved.

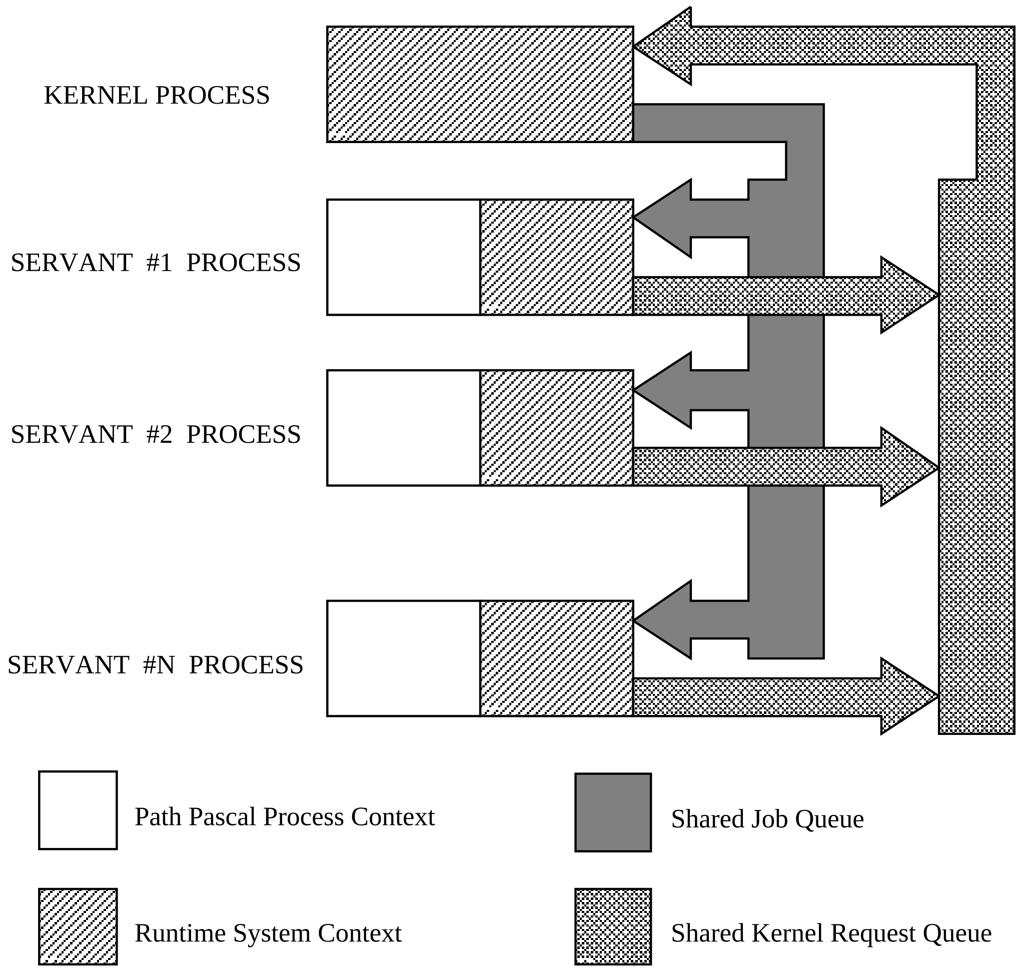
The kernel and servant processes are created at program initialization time and remain alive for the entire duration of the Path Pascal program. Path Pascal processes, which are not fixed in number, are executed as lightweight processes by the UNIX servant processes. In Figure 3, a running Path Pascal process is shown as an oval below the square denoting the servant process which is executing it. Note that not all servant processes are always busy executing Path Pascal processes.

Logically, the multiprocessor runtime system consists of the kernel process, the runtime system portion of each servant process, and the shared memory queues which connect the runtime system side of the servants to the kernel. Refer to figure 4. The servant processes context switch between runtime system code and Path Pascal processes like a pair of coroutines.

A servant process is similar to the entire single processor runtime system described in chapter two. In fact, the key difference between the multiprocessor and single processor runtime systems is that the multiprocessor implementation has multiple, concurrently executing servant processes, each operating like a single processor runtime system.

3.2. Kernel/Servant Interactions

The kernel process and the servant processes form a master-slave relationship. The kernel sends Path Pascal processes to the servants to be executed while the servant processes return service requests made by the Path Pascal processes to the kernel.



Logical Extent of the Multiprocessor Runtime System
with respect to Path Pascal Processes

Figure 4

Two shared memory queues are used for the kernel-servant communication. The job queue is used by the kernel to distribute Path Pascal processes to the servants. Only the kernel adds tasks to the job queue while all of the servants share the job queue to obtain new processes to execute. The kernel request queue is shared by the servants to send service requests to the kernel from their client Path Pascal processes. While all of the servants add requests to the kernel request queue, only the kernel removes and acts on those requests.

The contents of messages traveling between the kernel and servants are more general than described above. Messages sent to servant processes are not limited to execution requests, but may also include kernel directives such as *initialize* and *shutdown*. Similarly, messages sent from the servants to the kernel may provide status information rather than requesting a service.

Since there is only one kernel process, it can be trusted not to corrupt the data structures that must be maintained in order to satisfy service requests and generate new tasks for servant processes. The kernel-servant model was chosen because of this inherent simplicity over more complicated symmetric schemes where kernel requests might be handled by any processor.

3.3. Host Operating System Support

Early in the design process, the operating system support provided by the Sequent Symmetry system for parallel applications [Oste87] was evaluated. This evaluation lead to the following design decisions.

It was decided to let UNIX processes spin in the absence of available work. The alternative of having the runtime system put temporarily unused processes to sleep was rejected because it was felt that the overhead required to manage the sleeping processes would be more costly than enduring short busy waits. Further, since the number of servant processes can be chosen by the user at runtime, it is expected that an appropriate number of servant processes will be selected for a given situation.

The Sequent Symmetry parallel programming library contains a variant on the standard UNIX *fork()* system call. *M_fork()* [Oste87] starts a programmer specified number of processes working on the same function. When all processes return from the function, the *m_fork()* call returns. At that point, the UNIX processes are still alive, spinning. They can be killed, suspended, or reused by another call to *m_fork()*. These semantics could be used to change the number of servant processes during program execution; however, to keep the initial design from becoming overly complex, it was decided that the number of servant processes would be fixed during execution.

Since the kernel and the servants form a master-slave relationship and do not share code, the design called for using standard *fork()* system call to create the kernel process followed by a call to *m_fork()* to create the identical servant processes.

The Sequent Symmetry system supports spinlocks which can be operated [Oste87] by either function calls or in-line macros. Design tests indicated that using in-line macros in place of function calls could reduce execution times as much as fifty percent for programs with extreme levels of lock contention. Since lock contention within the runtime system was not anticipated to be nearly as high, a more modest decrease in execution time was expected. The mild disadvantage of using in-line macros is that they increase code size. However, since the number of lock operations in the runtime system was estimated to be less than twenty, the increase in code size was considered negligible. For these reasons, in-line macros are preferred, but because of testing and debugging needs, support for using either in-line macros or function calls was planned for the implementation.

3.4. Runtime System Services

Analogous to how the multiprocessor runtime system requires services from the host operating system, Path Pascal processes require services from the runtime system. As the front-end passes over a Path Pascal program, it inserts calls to the runtime system for various service requests. This section describes the services in the order they were designed to be implemented in the multiprocessor runtime library.

At a minimum, the runtime system must provide the *done* service call, requested by a terminating process. This service is necessary for all Path Pascal programs, even those which use only standard Pascal constructs, because the main program in a Pascal program is a process in Path Pascal.

To create Path Pascal processes, two more service calls are needed: *create* and *suspend*. *Suspend* is used to suspend the main process[2] and prevent it from exiting until its last child awakens it. During the service call *done*, a check is made to determine if the parent of the calling process is suspended. If it is and if the calling process is the last child of the parent to exit, then the parent is awakened.

To implement path expressions, which synchronize Path Pascal process with respect to objects, the runtime system must support the service calls *pop* and *vop* which perform P and V operations [Dijk68], respectively on counting semaphores within the object. The front-end translates path expressions into a sequence of P and V operations and inserts the necessary runtime requests into the intermediate code.

[2] In general, any process which shares an activation record with a child will need to suspend itself prior to exiting [Grun85].

The single processor runtime system allows Path Pascal processes to handle P and V operations themselves, requesting service from the runtime system only if a process needs to enqueue itself on a semaphore or signal that some other process should be dequeued. This is a safe strategy for the single processor implementation since there is no danger of simultaneous access to a semaphore.

The above approach was rejected in the original multiprocessor runtime design because more than one servant process might attempt to perform P or V operations simultaneously on the same semaphore. It was felt that only the kernel process should be trusted to handle P or V operations.

The single processor runtime system maintains a simulated time clock and provides two functions to access it. The first of these, *delay(x)*, is called by a process requesting to be delayed by *x* simulated time units. The second function, *wallclock*, returns the current simulated time. Simulated time starts at zero upon program invocation and jumps by intervals determined by the process which is currently delaying for the least amount.

While real-time delays are difficult to implement in the single processor runtime system, they are quite reasonable to implement in the multiprocessor version since the kernel is a dedicated process. The multiprocessor version was therefore designed to provide the same functions, *delay(x)* and *wallclock*, with the same semantics as above, except that the runtime system clock would measure real-time instead of simulated time.

3.5. Shared Memory Design Plan

Data required by multiple runtime system processes must be stored in shared memory. For example, both the kernel request and job queues must be in shared memory to allow the kernel and servant processes to send messages to one another. Other structures associated with Path Pascal processes must also be in shared memory. These include Path Pascal process stacks and process control blocks.

Regardless, no attempt was made at design time to make a comprehensive list of shared memory data items. Rather, it was decided that items should be placed into shared memory *on demand* when it was clear they belonged there. This practice prevents placing more items in shared memory than necessary, thus conserving a limited resource.

CHAPTER 4.

Implementation of the Runtime System

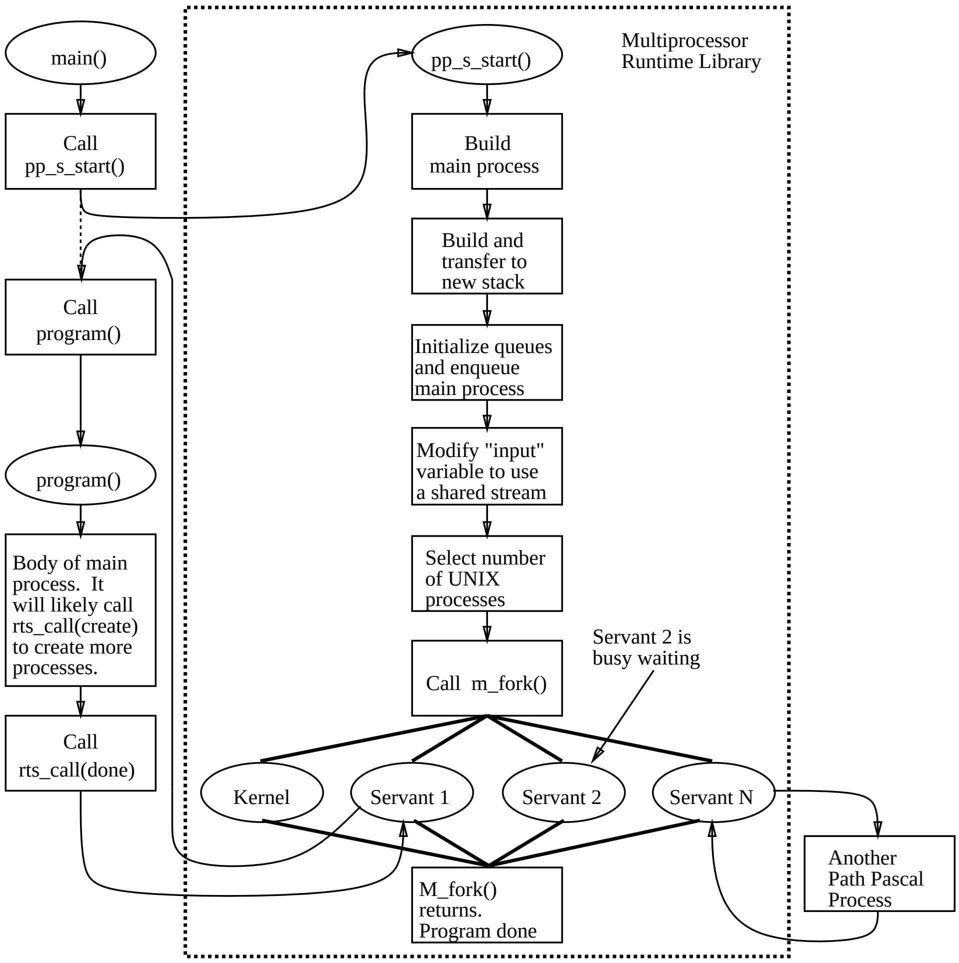
This chapter documents the implementation of the Path Pascal multiprocessor runtime system. During implementation, several design plans, as presented in chapter three, were changed after unforeseen details arose during this phase. Those modifications are discussed in the text when they arise.

The chapter is divided into the following sections: The first section describes the multiprocessor runtime system initialization procedure; section two addresses the implementation of queues; section three discusses multiprocessor specific details of the kernel and servant process implementations; and section four comments on several shared memory data structures and presents a complete list of the shared memory data structures.

4.1. Runtime System Initialization

As with the single processor system, the multiprocessor runtime system is initialized by calling *pp_s_start()*. This routine initializes the multiprocessor runtime system by performing the following: bootstrapping the initial Path Pascal process, modifying the Pascal text variable *input* to work in the multiprocessor environment, selecting the number of UNIX processes, and forking the kernel and servant processes. This section describes these initialization tasks.

Figure 5 shows the multiprocessor runtime system surrounded by the routines it interacts with. Figure 5 is analogous to figure 2 which displays the single processor runtime



Multiprocessor Runtime System

Figure 5

system.

4.1.1. Bootstrapping the runtime system

Pp_s_start() begins by building the main Path Pascal process. A process control block (PCB) for the main process is allocated and initialized with default process parameters and the frame pointer used within *pp_s_start()*.

Later, a servant process will perform a context switch to the main process by calling *rts_exec()* which will load the above frame pointer and return control to the instruction just beyond the call to *pp_s_start()*. See the lightly dashed line in figure 5. The scheme used here is similar to the one employed by the single processor implementation; however, here, the *rts_exec()* calls, which execute Path Pascal processes, are made by several UNIX servant processes rather than by a single process.

After the PCB for the main process is built, a new stack is created for *pp_s_start()*. The original stack, inherited by the program from the Sequent Symmetry operating system, will be used by the main Path Pascal process. It is important that *pp_s_start()* have its own stack so its environment is preserved during the *m_fork()* call discussed below.

Next, *pp_s_start()* initializes several data structures including the kernel request queue and the job queue. Once these queues are ready, the main process PCB is packaged into a message, marked ready to execute, and enqueued on the kernel request queue.

A message is the unit of data shuttled between the kernel and servant processes via the kernel request and job queues. All messages have a type and may contain optional data such as a pointer to a PCB.

At this point, the runtime system is primed and will begin operating as soon as the kernel and servant processes are created.

4.1.2. Modifying input to use a shared stream

During testing, it was discovered that if a Path Pascal process which reads from the predefined Pascal text variable *input* undergoes a context switch to another servant, it fails to continue reading correctly. Specifically, large portions of input are skipped causing the program to reach end-of-file prematurely.

The problem was traced to the UNIX standard input stream upon which the Pascal *input* variable depends. The standard input stream is block buffered using a non-shared memory buffer. This prevents proper operation of the stream for Path Pascal processes which are switched among servants.

An initial solution was simply to set the standard input steam to be unbuffered. However, this produced a highly noticeable degradation in reading performance.

A better solution was found that involved allocating a new stream file and its associated buffer from shared memory and copying the standard input file structure into the newly created stream file. The appropriate field in the Pascal *input* structure was then modified to point to the new shared memory stream file.

While this solution does not solve the problem of reading external files, it does provide a high performance solution for the most frequent case: reading from standard input. Further, the adaptations required to read correctly and efficiently from arbitrary files should be similar to the above standard input modifications.

4.1.3. Selecting the number of UNIX processes

During early development, the number of UNIX processes created was simply hard coded. When testing began, more flexible methods of selection were needed.

The following algorithm was implemented to determine the number of UNIX processes to create:

(1) If the environment variable "PARALLEL" is set, then its value sets the number of processes to create. Note that this allows the number of processes to be set at runtime without recompiling.

(2) If (1) above does not apply, then the number of processes created is determined by the compile time option, -Pn, where n is the chosen number, assuming the option was specified on the compiler command line.

(3) If neither (1) nor (2) applies, then the number of processes forked is four or the number of online processors minus one, whichever is less.

In all cases, the Sequent parallel programming library requires the number of processes selected to be less than or equal to the the number of online processors minus one. Of the UNIX processes created, one will execute the kernel, and the remainder will be servants.

4.1.4. Forking the kernel and servant processes

The original design called for using the standard UNIX *fork* system call to create the kernel process, followed by invoking *m_fork()* to create the servants. This asymmetric approach was planned because the servant processes and the kernel process are dissimilar. The servants all execute the same code while the kernel has its own private code. Further, since the *m_fork()* call starts several processes all running the same function, it seemed perfect for creating the identical servants. However, this plan was not usable because of an undocumented incompatibility between *fork* and *m_fork()*.

Instead, the kernel process and the servant processes are created with a single *m_fork()* call. As each child of the *m_fork()* call is created, it is assigned an identification number which is one for the first child, two for the second child, and so on. After all child processes have started, the parent, which is assigned the identification number zero, also executes the given subprogram [Oste87]. Since the processes can determine their identification number, the parent process branches to the kernel code while the children branch to the servant code.

4.2. Queue Organization

This section discusses the implementation of queues used in the multiprocessor runtime system.

The design chapter defines two shared memory queues, the kernel request queue and the job queue, which transport messages between the kernel and servant processes. Collectively, these two queues are referred to as *external queues* since they carry messages to external processes.

The kernel also uses internal queues which are manipulated only by the kernel and, unlike external queues, do not require any process synchronization during access. Currently there are two internal kernel queues: the suspend queue, where a parent process waits for its children to complete and the delay queue where a process waits to be resumed after a period of real-time has passed.

The original implementation of queues used *queue nodes* as the basic element manipulated by the queues. For external queues, which transport messages, these queue nodes pointed to messages. This extra level of indirection was tolerated to allow one set of queueing routines to operate on both queue types, external and internal.

Later, it became evident that using just one set of queue routines was not practical. The process control block of a process waiting on a semaphore is held within the private queue for that semaphore. Since the front-end defines process control blocks as the element the semaphore queues manipulate, large modifications to the front-end would be required in order to accommodate the the indirect queue node implementation. One of the project goals is to modify as little code as possible outside the runtime library. For this reason and because maintaining free lists of queue nodes can cause deadlock if interactions with other queues are not carefully considered, the queue nodes were removed from the implementation.

The queue configuration in the current implementation uses messages as the basic element for external queues and process control blocks as the basic element for both semaphore queues and internal kernel queues. While this solution requires two sets of queue operation routines, it does not suffer from the problems described above and it allows different operations to be defined on the two queue types. The external queues support *initialize*, *enqueue*, *dequeue*, and *empty check*; while process control block queues support *enqueue*, *dequeue*, and *remove queue*. The *remove queue* operation allows any specified element to be removed from the queue. This operation is necessary to remove a parent process from the suspend queue after its last child process terminates.

4.3. Kernel and Servant Processes

This section explains the implementation of the kernel and servant processes. These UNIX processes interact to execute Path Pascal processes and handle their service requests. Emphasis will be placed on implementation issues which significantly differ from the single processor runtime system. Therefore, no further mention will be made of the following runtime system service calls: *done, suspend,* and *create* since these are implemented in a similar manner by both runtime systems.

The section begins with an overview of the flow-of-control within the kernel and servant processes. Next, specific kernel/servant implementation issues are explained. These issues include: idle loops, deadlock detection, P and V operation handling, a new runtime system service call called *preempt*, and the implementation details of the real-time functions: *wallclock()* and *delay()*.

4.3.1. Overall Kernel and Servant Flow-of-Control

The kernel process and the servant processes share a similar overall flow-of-control. Compare the flow-of-control of a servant process shown in figure 6 with the flow-of-control of the kernel process shown in figure 7. The only differences are the deadlock detection step and the monitoring of the delay queue which are both performed only by the kernel process. Messages received by the kernel process usually request a kernel service call while messages received by servant processes usually request that a Path Pascal process be executed.

```
While not shutting down {
        While no message is pending
                Spin;
        Dequeue a message;
        Handle the message;
}
```

Servant Process Flow-of-Control

Figure 6

```
While not shutting down {
        Deadlock detection;
        Repeat
                If (delay queue is not empty) then
                        Resume all ready Path Pascal processes;
        Until a message is pending;
        Dequeue a message;
        Handle the message;
}
```

Kernel Process Flow-of-Control

Figure 7

4.3.2. Kernel and Servant Idle Loops

The single processor runtime system does not have an idle loop; there is only one processor and it is always busy running either a Path Pascal process or servicing a runtime system request. However, the processors in the multiprocessor runtime system will not always be doing useful work; they may be idling. This section describes the idle loops.

To determine if a message is available for dequeuing, both the kernel process and the servant processes use the function *is_empty_eq()* to return the state of the given queue. To reduce lock hold time, *is_empty_eq()* does not lock out other processes while ascertaining whether the given queue contains messages.

Since the kernel process is the only reader of the kernel request queue, there must be a waiting message after *is_empty_eq(kernel_request_queue)* returns false. The servant processes, on the other hand, may find that there is no pending message even after *is_empty_eq(job_queue)* returns false. This is possible because between the time a servant process determines there is a waiting message and attempts to dequeue it, another servant process may have successfully dequeued the message first.

Undoubtedly, an in-line macro would be more efficient than calling the function *is_empty_eq()* to determine if an external queue is empty. However, this approach was not used for two reasons. First, there is little to be gained by optimizing the idle loops of the kernel and servant processes. Second, using an in-line macro would prevent performance tools from collecting valuable metrics localized within the function. For example, the Interactive Performance System (IPS) [Holl90], generates reports and histograms of internal program metrics including CPU time, elapsed time, execution time, procedure call counts, number of spinlock operations, and spinlock holding time. These metrics are maintained for each procedure within each process.

By using IPS to measure the CPU time spent in *is_empty_eq()* over the course of the program, one can approximate how much CPU time per process is spent idling. This information may be useful when analyzing whether a process has enough unutilized CPU power to handle additional tasks. See table 1 for an example idle time analysis employing IPS metrics.

4.3.3. Deadlock Detection

The runtime kernels for both the single processor and multiprocessor systems detect deadlock among Path Pascal processes. The single processor kernel declares deadlock has occurred when there exist active processes but none are on the ready or delay queues. The multiprocessor kernel, on the other hand, must make a more subtle test since the above criteria do not imply deadlock if Path Pascal processes are executing on servant processors.

Figure 8 shows the single processor deadlock detection algorithm. This detection algorithm works for the single processor runtime system because a process ready to execute must be on either the ready queue or the delay queue. If no process is available for execution, yet there are active processes waiting on either the semaphore queues or the suspend queue, then deadlock has occurred.

Test: T22G 06/28/91 A				
Process	Kernel	Servant 1	Servant 2	Servant 3
Elapsed Time	41.29	41.29	41.29	41.29
CPU Time (process)	27.63	26.66	27.16	27.27
CPU Time (*is_empty_eq()*)	6.88	7.65	8.92	8.73
Percentage of CPU time spent in *is_empty_eq()*	24.9	28.7	32.8	32.0

Process Idle Time Analysis

Table 1

```
If (ready queue is empty) then
    If (delay queue is empty) then
        If (active process count is not 0) then
            deadlock is detected;
```

Single Processor Deadlock Detection Algorithm

Figure 8

The multiprocessor implementation may not be deadlocked if the conditions shown in figure 8 hold. This is possible because servant processes may be executing Path Pascal processes allowing both the job queue and the delay queue to be empty while the active process count is positive. The multiprocessor runtime system resolves this by maintaining a count of waiting processes in addition to active processes. If these two quantities become equal after the main process has been injected, then deadlock has occurred. Note that the multiprocessor deadlock detection algorithm is actually simpler than the single processor version.

4.3.4. P and V Operation Handling

This section describes three approaches to handling P and V operations that were implemented and tested. They are explained in order from easiest to implement with poorest performance to most difficult to implement with best performance.

4.3.4.1. Kernel Handling of P and V Operations

The original design called for the kernel process to handle all P and V operations. This conservative approach required, for each P and V operation, a context switch from the Path Pascal process to the servant process which then sent a message to the kernel. Large inefficiencies resulted from sending P operations to the kernel that did not necessitate enqueuing the process. In those cases, the kernel merely put the process on the job queue for further execution. This design was easy to implement since it did not require locks to protect semaphore queues. It was also extremely slow.

4.3.4.2. Servant Handling of P and V Operations

To improve performance, a second approach for handling P and V operations was implemented. Instead of sending all P and V operations to the kernel, the servant code was altered to determine if the P or V operation actually required modifying a semaphore queue. In the case of a P operation, if the process did not need to be enqueued on a semaphore, the servant immediately performed a context switch back to the Path Pascal process. Otherwise, the process was sent to the kernel.

The key challenge when implementing servant handling of P and V operations is preventing servant processes from simultaneously accessing semaphores. The ideal solution is to have a lock for each semaphore. At first, this idea was rejected because it required modifying the semaphore structure defined in the front-end. Dirk Grunwald warns in his thesis that, "Extensive modifications to the [semaphore] data structure would require modifications to the semaphore initialization and allocation routines of the path expression compilation section of the compiler. Fortunately, it is not envisioned that the data structure will be changed often." [Grun85]

Two alternative methods to prevent simultaneous access to semaphores were considered:

(1) Using a single lock for all semaphores.

(2) Creating a semaphore lock manager within the runtime library which would maintain the semaphore locks within its private data space. Processes would call the lock manager to lock and unlock semaphores before and after they accessed the semaphore, respectively. If a semaphore was being accessed for the first time, the lock manager would first allocate a lock and associate it with the given semaphore. This scheme would not require any modifications to the front-end semaphore structure.

After consideration, the alternative solutions were rejected: (1) was likely to cause high levels of contention and (2) was as difficult to implement as the ideal solution. Consequently, the front-end was modified: a lock field was installed into the semaphore structure and the semaphore initialization routines were altered to emit intermediate code to initialize the locks.

Since each semaphore could now be locked individually, the servant processes could perform P and V operations themselves and request kernel assistance only if a process required enqueuing on a semaphore or dequeuing off a semaphore.

After the servant handling design was installed, an error with an interesting result was found in the servant code. When a servant performed a P operation which required enqueuing a process, the servant did not hold the lock on the semaphore long enough. The lock was held only while the semaphore count was decremented and tested. If the semaphore count tested negative, the lock should of been held until the process requesting the P operation was safely on the kernel request queue.

Since the lock was not held long enough, a race condition developed between the enqueue request generated by the P operation and a dequeue request generated by another process performing a V operation on the same semaphore. If the enqueue request reached the kernel first, everything worked as planned; however, if the dequeue request reached the kernel first, a panic would result if the queue for the given semaphore were empty.

Two solutions to remove the race condition were contemplated. The simplest was to hold the lock on the semaphore for a longer period during an enqueuing P operation. The second solution allowed the race condition to develop but avoided the panic by temporarily holding the premature dequeue request in the kernel until its corresponding enqueue request arrived. These two solutions were experimented with briefly. Preliminary results showed the "panic avoidance" method may reduce the number of P operations requiring enqueuing by as much as four percent. With the "panic avoidance" method, P operations hold the semaphore lock for a shorter period allowing V operations to proceed sooner, which may explain the decrease in the number of enqueuing P operations. More research is needed before a definitive explanation to the above result can be given.

4.3.4.3. In-line Process Handling of P and V Operations

While servant handling, rather than kernel handling, of P and V operations improved performance remarkably, the single processor compiler continued to produce faster executables than the multiprocessor compiler regardless of the number of processors used. A comparison with the single processor compiler traced this poor performance to the number of context switches incurred by the multiprocessor runtime system: at least two for each P and V operation handled by a servant.

The multiprocessor performance was improved to its current level by by allowing Path Pascal processes to perform P and V operations in-line similar to how P and V operations are handled within the single processor runtime system. After the modifications, a context switch to a servant occurs only when an enqueue or dequeue message must be sent to the kernel.

In order for P and V operations to be performed within multiprocessor Path Pascal processes, in-line lock operations must be installed to insure mutual exclusion during semaphore access. Therefore, the simple in-line assembly code exchanger sequences for "POP" and "VOP" were replaced with more complex sequences that included lock and unlock calls to the Sequent parallel programming library.

4.3.5. The Preempt Service Call

During development, the *preempt* service call was implemented to help approximate the behavior of the yet unimplemented *delay* service call. While it was mostly unsuccessful at this task, *preempt* remains in the implementation since it can conceivably be of use in a multiprocessor Path Pascal program.

The *preempt* service call causes the invoking Path Pascal process to be preempted in favor of a waiting process, if one exists. If the servant successfully dequeues a waiting process from the job queue, then the preempted process will be enqueued on the job queue and the new process will be executed by the servant. If the job queue is empty when the servant enquires, the original process resumes executing. Note that no kernel interaction is required to handle the *preempt* service call.

The *preempt* service call can be used to prevent Path Pascal processes from monopolizing servants. Suppose three servant processes are used to execute a Path Pascal program requiring no process synchronization. Further, suppose the main Path Pascal process is to create five additional Path Pascal processes. The first three Path Pascal processes may usurp the three servants and prevent the main process from creating the remaining processes until the first three have relinquished control of their servants due to termination or requesting preemption.

4.3.6. The Wallclock and Delay Service Calls

Unlike the single processor runtime system which implements the service calls *wallclock* and *delay* in terms of simulated time, the multiprocessor version realizes these calls with a real-time clock. A call to *wallclock()* will return the number of milliseconds since the Path Pascal runtime system initialized itself. A process calling *delay()* will wait on a kernel queue for the specified number of milliseconds before being returned to the job queue.

The heart of the multiprocessor runtime system clock is an unsigned 32-bit microsecond clock furnished by the Sequent Symmetry operating system. It begins counting at zero when the machine is powered up and thereafter rolls past zero approximately every seventy-one minutes (2^{32} microseconds). The runtime system routine *Pp_time()* uses the microsecond clock to return the current Path Pascal runtime system time. Calibrated in milliseconds, *Pp_time()* is initialized to zero upon program initialization and runs continually during the life of the program. Care has been taken to avoid a discontinuity in the millisecond clock when the microsecond clock rolls past zero; however, an incorrect *pp_time()* result is possible if *pp_time()* is not called at least once every seventy-one minutes during the execution of a Path Pascal program. While this disclaimer is not likely to affect the average Path Pascal user, if it were a problem, the kernel could be modified to insure that *pp_time()* was called often enough automatically.

Both *wallclock()* and *delay()* use *pp_time()* in their implementations. *Wallclock()* simply calls *pp_time()* and arranges for the result to be sent back to the calling Path Pascal process. When *delay()* is invoked by a process, its servant uses *pp_time()* and the duration of the delay to calculate when the delay should end. This end-of-delay time is recorded in the process control block by the servant who then sends the process to the kernel to be inserted into the delay queue. The delay queue is an internal kernel priority queue that maintains the process control blocks of delaying processes in order based on their end-of-delay times. As the kernel idles, and after each message is handled, the delay queue is inspected. Each process which has waited the duration of its delay or longer is moved to the job queue.

In cases where enough servants are available to run all ready Path Pascal processes, tests show that actual delay times match requested delay times closely, often to within one millisecond. Larger discrepancies may occur if a resumed process must wait on the job queue before being dequeued by a servant.

Finally, hindsight suggests that simulated time could also of been implemented into the multiprocessor runtime system furnishing two modes of timing for *wallclock()* and *delay()*. The simulated time mode would be useful for making comparisons between the single processor and multiprocessor runtime libraries when executing Path Pascal programs that utilize *wallclock()* and *delay()*.

4.4. Shared Memory Utilization

During implementation, data structures were placed in shared memory when it became clear that the structure must be shared among the kernel and servant UNIX processes. This final section of the implementation chapter enumerates the data structures assigned to shared memory and comments on several shared memory issues. some of the more prominent structures belong in shared memory.

Since Path Pascal processes may be executed by any servant process, all state information associated with Path Pascal processes, directly or indirectly, must be stored in shared memory. Consider Path Pascal objects. Access to objects by processes are governed by path expressions which are translated into sequences of P and V operations on semaphores. Therefore, semaphores must be accessible by all Path Pascal processes being executed by any servant process. Further, once a Path Pascal process satisfies the path expressions, that process may access data encapsulated within the object. For these reasons, object data must be allocated from shared memory.

The single processor implementation allocates data space for objects by calling the pascal library procedure, *new()*. In order to remain consistent with the single processor version and to insure that objects and dynamic Path Pascal variables are allotted from shared memory, calls to *new()* are replaced by calls to *shmalloc()*, a function defined in the Sequent parallel programming library. The in-line code exchanger, mppc2, performs the substitutions and handles the parameter differences between *new()* and *shmalloc()*.

In order for any servant to execute any Path Pascal process, the stack of each Path Pascal process must be in shared memory. When a process creates a child process, a shared memory stack is allocated for the new child. However, recall that during multiprocessor runtime system initialization, the main Path Pascal process inherits its stack from the operating system. The stack is not created by the same mechanism used to create stacks for child processes. Fortunately, the Sequent parallel library arranges for the system stack to be in shared memory when a call to *m_fork()* is present in the code [Oste87]. Had this feature not existed, the original system stack and the shared memory stack created for *pp_s_start()* could have been exchanged since the stack for *pp_s_start()* need not be in shared memory.

Other data structures which must be stored in shared memory include: process control blocks, external queue messages, file variables, strings allocated during compilation, and the process display, used to access data within scopes of other processes. Note that the runtime display is not shared. Each servant maintains private storage for a runtime display which is filled with the runtime display of the currently executing Path Pascal process. When a process is not executing, its runtime display is kept in its process control block.

CHAPTER 5.

Results and Interpretations

This chapter provides results and interpretations that assess the success of the multiprocessor Path Pascal compiler. Section one uses timing results collected from test runs of Path Pascal programs to demonstrate how multiprocessor execution times scale as larger numbers of servant processors are applied. The second section compares the two compilers with respect to their compilation speed, runtime library complexity, and related metrics. Finally, the last section discusses how Path Pascal processes are load balanced among the servant processors and how programming choices can affect performance.

Data collected for this chapter was obtained from a Sequent Symmetry computer with twenty Intel 80386 processors. In order to gather accurate data, test runs were made during periods of low load or under control of locally implemented batch system that runs programs with special privileges during off-hours. In all cases, individual program configurations were run at least three times and their results averaged.

5.1. Compiled Code Performance

Ideally, Path Pascal programs compiled by the multiprocessor compiler will exhibit execution times that decrease as the number of UNIX servant processes is increased. The measure of this phenomenon is referred to as processor scalability. This section evaluates the multiprocessor implementation with respect to processor scalability by considering the results of two Path Pascal programs.

5.1.1. Program One: Process Contention

Contention between Path Pascal processes for a limited resource affects processor scalability. One example of such a resource is a shared variable protected from simultaneous access by an object. Contention emerges as multiprocessor Path Pascal processes collide with one another while attempting to increment the protected shared variable. Test program one uses the above technique to cause variable contention. The program parameters will be configured three ways to induce three levels of contention: none, medium, and high.

Test program one, shown in figure 9, has 48 Path Pascal processes each incrementing one or more shared variables. Each shared variable is stored within its own object which protects it from being incremented by more than one process at a time. The level of contention between the Path Pascal processes is controlled by the program constants *ProcsPerObjgroup* and *ObjsPerObjgroup*. *ProcsPerObjgroup* determines how many processes will share a single group of objects. As this constant increases, so does the level of contention. *ObjsPerObjgroup* sets the number of objects in each object group. The level of contention decreases as this constant increases. Roughly speaking, the level of contention can be interpreted as the fraction *ProcsPerObjgroup* over *ObjsPerObjgroup*.

Consider program one configured for no contention where constants *ProcsPerObjgroup* and *ObjsPerObjgroup* both have the value 1. The graph in figure 10 shows reciprocal elapsed time versus the number of servant processes. Curves are drawn for: ideal speedup, based on the performance of the single processor executable; multiprocessor Path Pascal (Mppc); and single processor Path Pascal (Ppc). The graph indicates that the single processor executable is faster than the multiprocessor executable using one servant process. This result can be explained by the extra overhead endured by the multiprocessor executable that

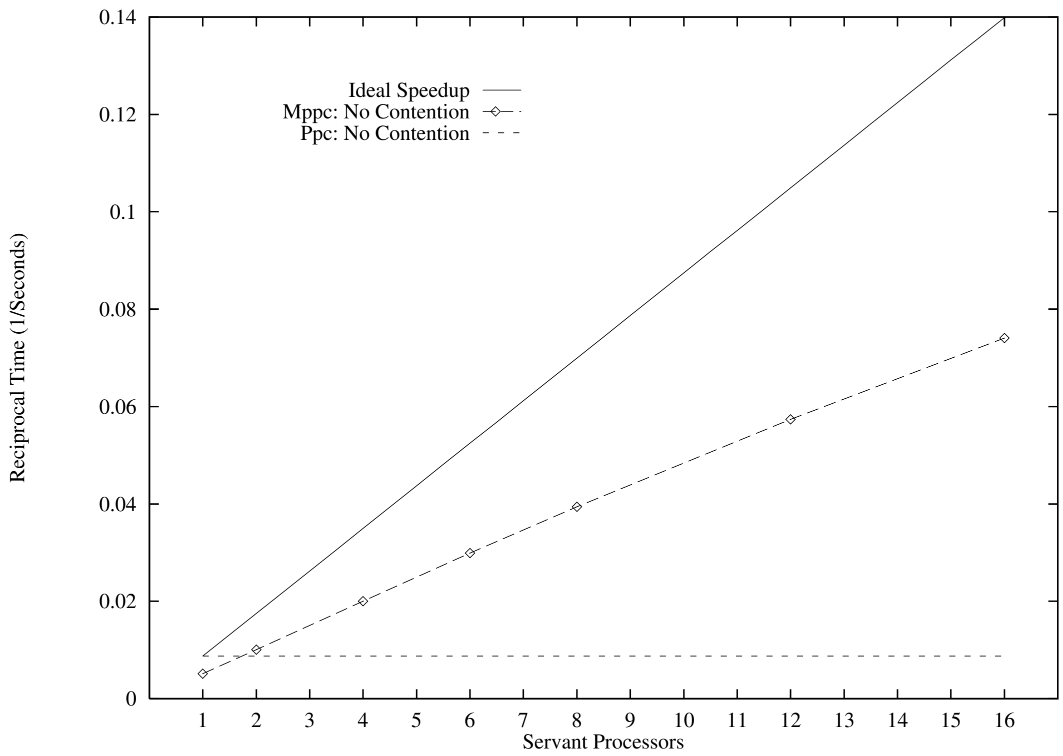
```
program contention(output);
    const
        MaxProc               = 47;         { 48 processes: numbered 0 .. 47       }
        Iterations            = 10000;
        ProcsPerObjgroup      = 2;          { Contention Level is 2 / 1            }
        ObjsPerObjgroup       = 1;
    type
        critical_obj = object
            path 1:(safe_incr) end;         { Path assures mutual exclusion        }
            var
                global_sum: integer;
            entry procedure safe_incr;
                begin
                    global_sum := global_sum + 1;
                end;
            initially;
                begin
                    global_sum := 0;
                end;
            finally;
                begin
                    writeln('The total sum is ', global_sum:1);
                end;
        end;
    var
        adders: array [ 0 .. MaxProc ] of critical_obj;
        i, objgroup, low_obj: integer;
    process increment(my_id, low_obj: integer);
        var
            i: integer;
        begin
            for  i := 0 to (Iterations - 1) do
                adders[low_obj + i mod ObjsPerObjgroup].safe_incr;
            writeln('Process ', my_id:1, ' done.');
        end;
    begin
        for  i := 0 to MaxProc do begin
            objgroup := i div ProcsPerObjgroup;
            low_obj := objgroup * ObjsPerObjgroup;
            increment(i, low_obj);
        end
    end.
```

Program One: Adjustable Process Contention

Figure 9

enables it to use multiple processors. With two servant processes, the performance of multiprocessor Path Pascal is approximately equal to single processor Path Pascal. Above two servant processors, the performance of multiprocessor Path Pascal is near optimal while the performance of single processor Path Pascal remains constant.

Figure 11 shows total user time for all processors versus the number of servant processors for the case with no contention. Surprisingly, as servants are added, the total user time decreases. This outcome may seem impossible: even if the servant processors displayed perfect linear speedup, their combined user time would remain constant. The key here is the

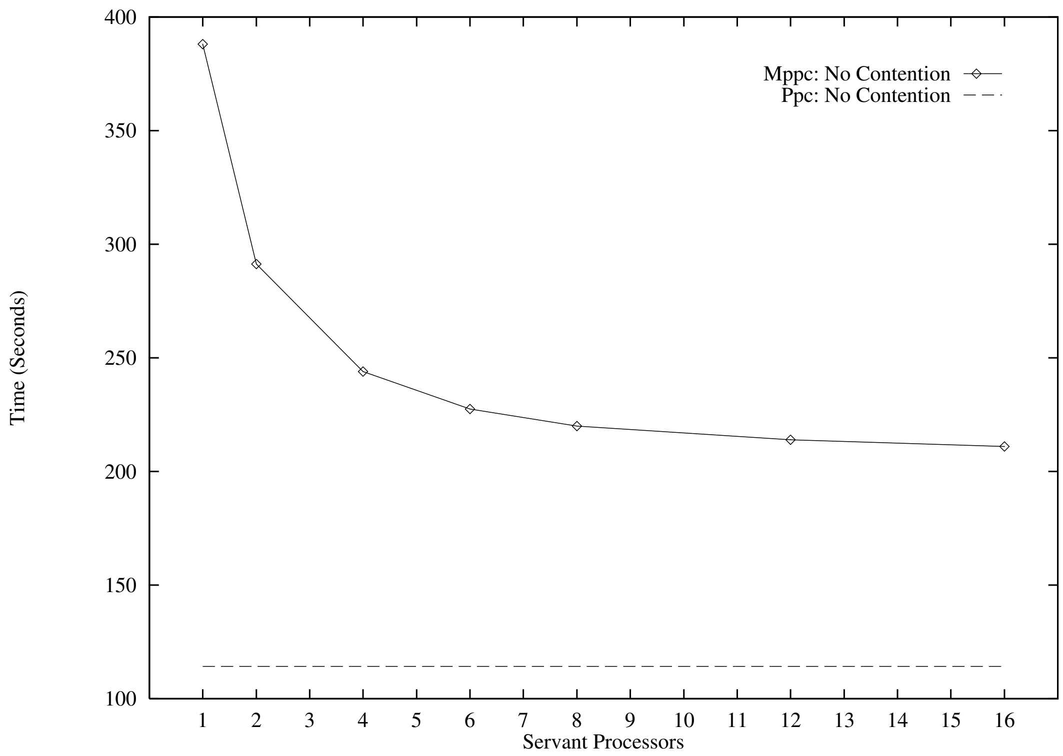


Elapsed Time Versus Number of Servant Processors
No Contention Case

Figure 10

kernel process which does not directly contribute to the completion of Path Pascal processes, but does accumulate user time. As servant processes are added, the elapsed time to complete the program decreases, and the kernel accumulates less user time.

While the case with no contention reveals the best possible multiprocessor performance, contention between Path Pascal processes induces terrible performance. Two levels of contention are defined: *medium*, where the constants *ProcsPerObjgroup* and



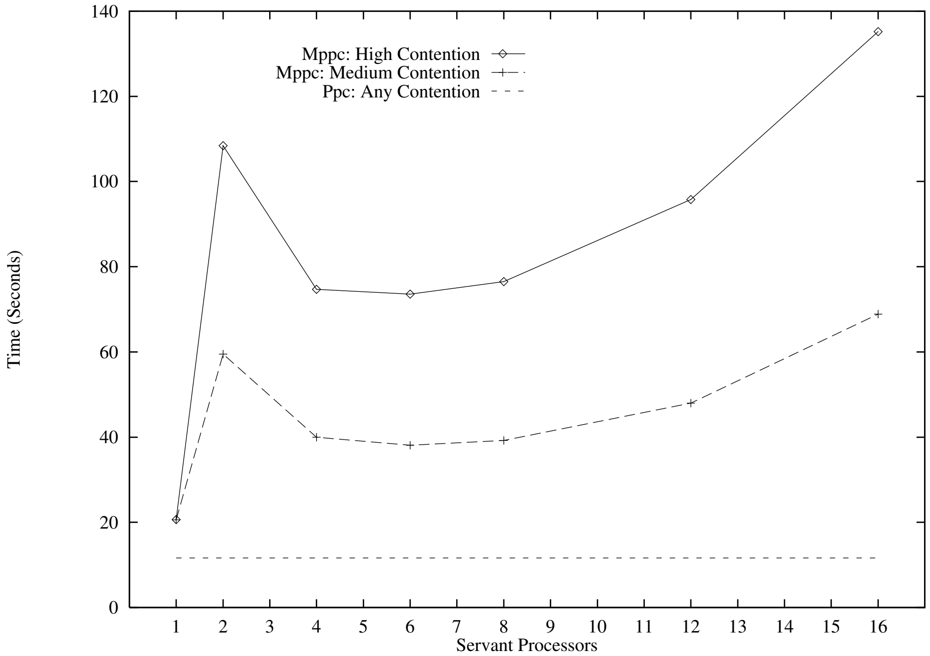
Total User Time Versus Number of Servant Processors
No Contention Case

Figure 11

ObjsPerObjgroup have values 3 and 2 respectively; and *high*, where the above constants have the values 2 and 1, respectively. Since the contention examples require considerable elapsed time to execute, the number of iterations used in program one was reduced to one tenth the number used in the above case with no contention.

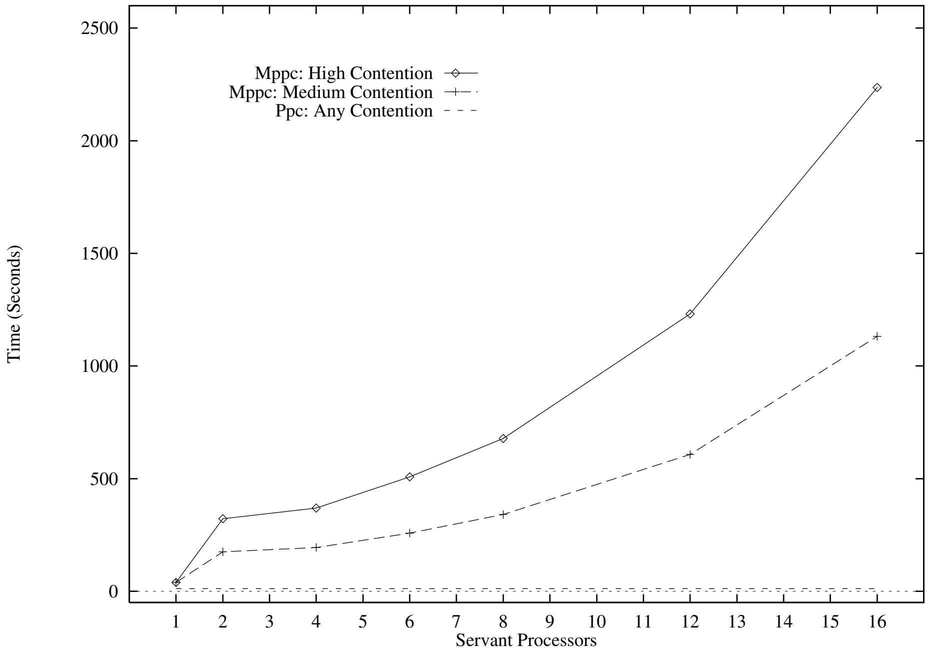
The graph in figure 12 shows multiprocessor and single processor elapsed times for both the medium level and high level contention situations. Note that the single processor performance is not affected by contention. Similarly, both of the multiprocessor contention examples share the same minimum elapsed time when only one servant processor is used. Although program one expresses process contention, that contention is not realized unless the processes are truly executed in parallel. Figure 12 also shows that above 2 processors, the multiprocessor elapsed time drops until it reaches a local minimum at 6 processors beyond which it monotonically increases. This can be explained by considering the two following counteracting forces and how their effects combine as the number of servant processes increase. First, as more servants become available to execute Path Pascal processes, the elapsed time declines as speedup is realized. Second, additional servants increase the contention between simultaneously executing Path Pascal processes, resulting in higher elapsed times.

The total user time as a function of the number of servant processes for the high and medium contention cases is shown in figure 13. When contention exists between Path Pascal processes executing in parallel, they will block one another precipitating large numbers of context switches. Since the speedup is far from ideal, the kernel effect discussed in the no contention case is vastly insufficient to cause the total user time to decline or even remain constant.



Elapsed Time Versus Number of Servant Processors
Medium and High Contention Cases

Figure 12



Total User Time Versus Number of Servant Processors
Medium and High Contention Cases

Figure 13

5.1.2. Program Two: Shared Buffer Example

Unlike test program one, many Path Pascal programs have only a small number of processes. For example, a Path Pascal program to simulate an operating system might use only four processes: one for the CPU, one for the scheduler, one for the pager, and one for the disk controller. Test program two, shown in figure 14, has just two processes: an *injecter* which adds characters to a shared buffer, and a *reader* which removes characters from the buffer.

```
program buffer(output);
    const
        Bufsize        = 5;
        Maxbuf         = 4;
        MaxIO          = 400000;
    type
        buffer = object
            path Bufsize:( 1:(fill); 1:(empty) ) end;
            var
                inptr, outptr: 0..Maxbuf
                buf: array[ 0..Maxbuf ] of char;
            entry procedure fill(ch: char);
                begin
                    buf[inptr] := ch;
                    inptr := (inptr + 1) mod Bufsize;
                end;
            entry procedure empty(var ch: char);
                begin
                    ch := buf[outptr];
                    outptr := (outptr + 1) mod Bufsize;
                end;
            initially;
                begin  inptr := 0; outptr := 0;  end;
        end; {of object}
    var
        bufx: buffer;
    process injecter;
        var
            index: integer;
            injectee:  char;
        begin
            for  index := 0 to (MaxIO - 1) do begin
                injectee := chr( ord('A') + index mod 26 );
                bufx.fill(injectee);
            end;
        end;
    process reader;
        var
            temp: char;
            index: integer;
        begin
            for  index := 0 to (MaxIO - 1) do
                    bufx.empty(temp);
            writeln ('Reader Done');
        end;
    begin
        injecter;
        writer;
    end.
```

Program Two: Shared Buffer Example

Figure 14

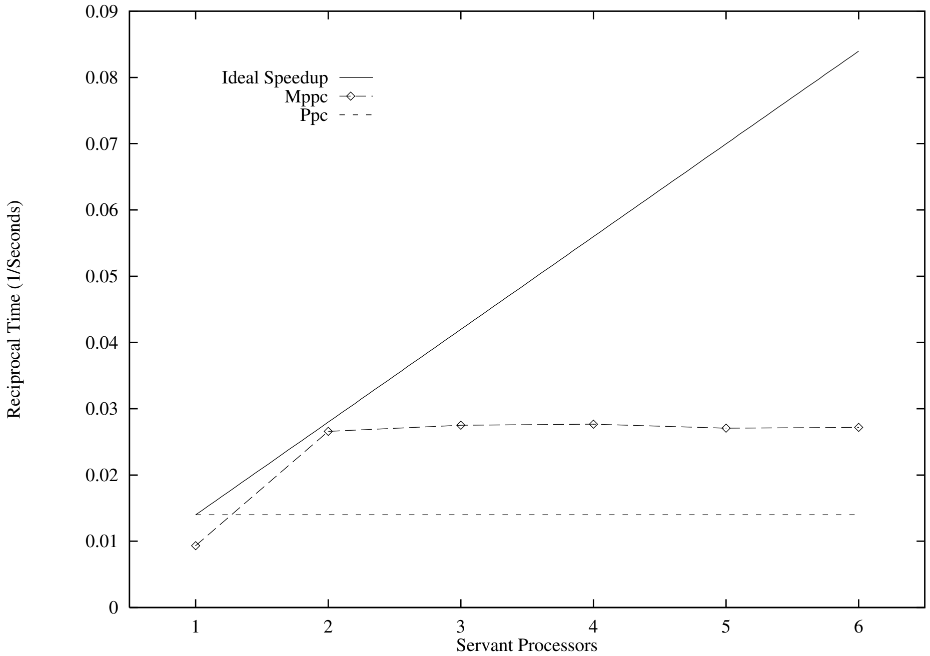
The graph in figure 15 displays reciprocal elapsed time versus the number of servant processors for the shared buffer program. Note how the performance of multiprocessor Path Pascal approaches the ideal speedup line as the number of servants grows from one to two. This tremendous performance improvement is a consequence of the reduction in the number of context switches as the second servant processor is added. See the graph in figure 16. When two servant processors are used, the *injecter* process can fill the buffer as the *reader* process is emptying it. A single processor executable, on the other hand, must cycle between completely filling and completely emptying the buffer - incurring context switches at each extreme.

If there are only two Path Pascal processes, additional servants in excess of two will not solve the task any faster. In general, a multiprocessor Path Pascal user must exercise care in choosing the number of servant processes. If possible, the number of servant processors selected should divide the number of Path Pascal processes evenly to avoid "remainder" processes which may not execute until the majority of processes have completed. Reconsider test program one. The 48 Path Pascal processes can be evenly distributed among all the servant configurations tested: 1, 2, 4, 6, 8, 12, and 16 servant processors.

5.2. Comparison of the Path Pascal Compilers

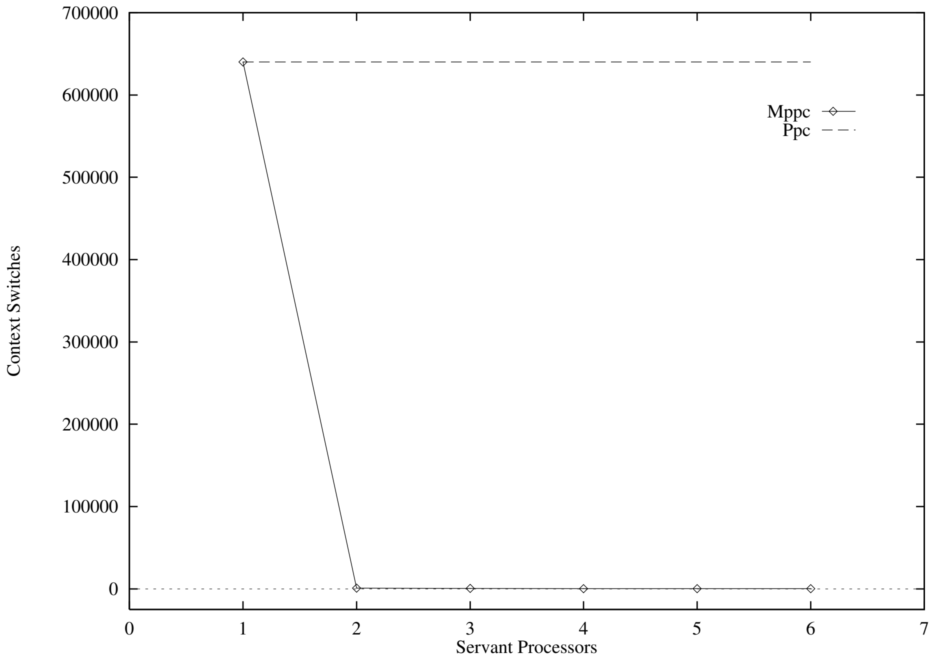
This section compares the single processor and multiprocessor compilers with respect to their performance, the size and complexity of their runtime libraries, and the size of the executables they generate.

All Path Pascal programs can be compiled by either the single processor or multiprocessor compilers. This desirable trait results from the fact that the two compilers use front-ends



Elapsed Time Versus Number of Servant Processors
Shared Buffer Example

Figure 15



Context Switches Versus Number of Servant Processors
Shared Buffer Example

Figure 16

that are nearly identical. Further, the two implementations share the exact same back-end. Only the runtime libraries are unique to each compiler. The time required by the multiprocessor implementation to compile the average Path Pascal test suite program is approximately 2.5 seconds longer than the time required by the single processor compiler. Subdividing the compilation process into its component parts reveals that 75% of the increase is due to linking the multiprocessor libraries. The remaining 25% is a consequence of compiling the multiprocessor module that defines the default number of UNIX processes to use provided a default value was specified on the compiler command line.

The single processor compiler uses only one nonstandard library, its runtime library, which is 10.8 kilobytes in length. The multiprocessor compiler requires the following nonstandard libraries: its runtime library, (27.3 kilobytes), the parallel programming library (26.5 kilobytes), and the microsecond clock library (53.4 kilobytes), which cumulatively total 107.2 kilobytes. The longer compilation times and larger executables exhibited by the multiprocessor compiler can be attributed to the number and size of these libraries.

Assuming that the quantity of source code is a reasonable indicator of complexity, the multiprocessor runtime library is over twice as complex as the single processor runtime library. Without comments or blank lines, the multiprocessor library contains 1581 lines of code compared to 741 lines within the single processor runtime library. Both runtime libraries devote a portion of their code for debugging purposes. The multiprocessor version has a set of 23 debugging "topics" which can be toggled on and off independently of one another. The single processor version has three global debugging levels which determine the quantity of debug output produced.

5.3. Load Balancing

Since multiprocessor Path Pascal uses a shared job queue to distribute Path Pascal processes to servants, good load balancing is expected. To measure load balancing, the number of messages arriving at each servant is counted. Since messages sent to servants nearly always contain new processes to execute, the number of messages handled by a servant can be used as a measure of the servant’s load.

The 16 processor, high contention configuration of program one was analyzed to determine the load balance of Path Pascal processes among servants. The coefficient of variation of the number of messages received by the servants was found to be 4.2%.

CHAPTER 6.

Conclusion

As single processor Path Pascal was an improvement over the Path Pascal interpreter which preceded it [Grun85], multiprocessor Path Pascal is an improvement over its predecessor, the single processor implementation. Both improved versions show faster execution times than their forerunners and both require slightly longer compile times.

The goals of the multiprocessor implementation have been met for the most part. First, scalable performance has been demonstrated by Path Pascal programs which allow it. Some Path Pascal programs do not lend themselves well to multiprocessing. While these programs may take longer to execute using the multiprocessor compiler, the multiprocessor results are still valuable because they express the true parallel characteristics of the Path Pascal program, which may not be evident from simulating parallelism with a single processor. Second, the semantics of single processor Path Pascal have been preserved except for the *delay()* call which now accepts real-time parameters specified in milliseconds rather than simulated time units. Third, both implementations share nearly all the code in the compiler except for their respective runtime libraries. This makes future porting and modification easier.

While working on the multiprocessor project, topics for future research presented themselves. The following is a partial list of those topics.

- Currently, the number of servant processors is fixed once the program begins running. The Sequent parallel program library *m_fork()* call offers functionality that could be used to allow the number of servants to vary during program execution.

- Instead of the master-slave (kernel-servant) model used now, better performance could be attained by spreading the kernel functions among the servants to form a symmetric runtime system.

- An X Window tool to observe the Path Pascal processes in their various states would be useful when debugging Path Pascal programs. The tool could be integrated with a standard debugger to allow stop points to be set for viewing the current state of the runtime system. It would also be useful if the tool could slow the the execution of the program in order to see processes make individual state transitions.

- The current implementation uses shared memory queues to communicate between UNIX processes. Many operating systems provide general message passing facilities. It would be interesting to determine if improved performance could be achieved by using these operating system supported message passing systems.

As a final note, a secondary goal of this thesis has been to provide an overview of the entire Path Pascal compiler and to explain concepts which a future implementor would find useful. This goal was motivated by the author's difficulty in finding documentation explaining fundamental details of the Path Pascal compiler.

APPENDIX A.

Porting Guide for Path Pascal

This appendix is intended to ease porting either the multiprocessor or single processor implementations of Path Pascal by providing a basic outline of steps to follow.

The first step is to collect necessary software. Besides the Path Pascal distribution itself, it is extremely important to secure the Berkeley Pascal sources for the target architecture. Otherwise, the task of porting Path Pascal must be preceded by a much larger chore: porting Berkeley Pascal. The Berkeley Pascal sources are useful because machine dependent portions of an existing set of Path Pascal source files can be replaced by the corresponding machine dependent code lifted from the Berkeley sources. Since Path Pascal uses the Berkeley back-end unmodified, it is also important to locate either the back-end executable on the target machine or the sources to create it. Often the back-end will already be installed in a standard library. If the Berkeley Pascal sources for the target architecture are not available, it may be more feasible to write a stripped down Path Pascal parser/back-end using the compiler generating tools Lex [Lesk84] and Yacc [John84a].

Step two is to port the front-end, ppc0 or mppc0. Even though the front-end emits intermediate code for the most part, there are assembly code fragments which the front-end passes unchanged to the back-end. The bulk of the routines which emit this assembly code are in the file "fend_<architecture>.c" and/or "fend.c". The <architecture> suffixes include: "i386" for Intel 80386, "ns32000" for National Semiconductor 32k, "vax", and "mc68000" for Motorola 68k. Besides these two files, the following list of front-end source files are likely to require modification: var.c, tmps.c, tmps.h, objfmt.h, align.h, p2put.c, pc.h, pccaseop.c, sconv.c, lab.c, and ppc0/whoami.h. Once the front-end is complete, it should be possible to pass a Path Pascal program through the front-end and the back-end to produce assembly code that the target assembler accepts.

Step three is to port the runtime library. The runtime libraries for both the single processor and multiprocessor compilers are written in C with the exception of the context switching routines *rts_exec()* and *rts_call()* which should be translated first. The debugging symbol table string entries (stabs) for the context switching routines can easily be generated by compiling the following lines of C code into assembly with the command *cc -g -S test.c*.

```
void __rts_exec()
{ }
void __rts_call()
{ }
```

A strategy for porting the runtime library is to implement the runtime services in an order that allows intermediate testing of the library. Start with the service calls *done* and *suspend*. When these runtime services are available, it should be possible to compile a standard Pascal program. Next, install the system call *create* to enable process creation followed by the service calls *enqueue* and *dequeue* required to use path expressions. Finally, add the *delay* service call.

As the runtime library is being ported, the in-line assembly code exchanger (ppc2 or mppc2) will also require modification. When the main program wants to suspend itself, it will make a call to the undefined function *WFS* (wait-for-sons), which is translated by the assembly code exchanger into a short segment of assembly requesting the *suspend* service from the runtime library. The wait-for-sons pattern must be defined before standard Pascal programs can be compiled. Path expressions depend on the P and V operation patterns, *POP* and *VOP*

The last step is to build the compiler driver (ppc or mppc) which invokes all of the compiler passes. Ppc3 (single processor version only) is nonessential but completely machine independent and should compile easily.

During development, a makefile that compiles test programs compiler component by compiler component is useful as it allows intermediate results to be viewed. Finally, a good source level debugger with assembly level debugging capabilities is essential.

REFERENCES

[Andr83]

Andrews, Gregory R. and Schneider, Fred B., ‘‘Concepts and Notations for Concurrent Programming,’’ *Computing Surveys* **15**(1) p. 7 ACM, (March 1983).

[Camp76]

Campbell, Roy H., ‘‘Path Expressions: A Technique for Specifying Process Syncronization,’’ Ph.D. dissertation, Computing Laboratory, University of Newcastle upon Tyne, England (August 1976).

[Deit84]

Deitel, Harvey M., *An Introduction to Operating Systems,* Addison-Wesley, Reading, MA (July 1984).

[Dijk68]

Dijkstra, E. W., ‘‘The structure of ’THE’ multiprogramming system,’’ *Communications of the ACM* **11**(5) pp. 341-346 (May 1968).

[Grun85]

Grunwald, Dirk C., ‘‘An Implementation of Path Pascal,’’ Masters Thesis and Tech Report UIUCDCS-R-85-1213, Dept of Computer Science, University of Illinois, Urbana, IL (June 1985).

[Holl90]

Hollingsworth, Jeff, Irvin, Bruce, and Miller, Barton P., ‘‘IPS User’s Guide [version 3.0],’’ Draft of unpublished manual, Dept. of Computer Science, University of Wisconsin, Madison, WI (November 1990).

[John84a]

Johnson, S. C., ‘‘Yacc: Yet Another Compiler-Compiler,’’ in *Unix Programmer’s Manual - Supplementary Documents*, Computer Science Div., Dept. of EECS, University of California, Berkeley, CA 94720 (March 1984).

[John84]

Johnson, S. C., ‘‘A Tour Through the Portable C Compiler,’’ in *Unix Programmer’s Manual*, *Supplementary Documents*, Computer Science Div., Dept. of EECS, University of California, Berkeley, CA 94720 (March 1984).

[Joy84]

Joy, William N., Graham, Susan L., Haley, Charles B., McKusick, Marshell K., and Kessler, Peter B., ''Berkeley Pascal User's Manual Version 3.0,'' in *Unix Programmer's Manual - Supplementary Documents*, Computer Science Div., Dept. of EECS, University of California, Berkeley, CA 94720 (March 1984).

[Kess83]

Kessler, P., ''The Intermediate Representation of the Portable C Compiler as used by the Berkeley Compiler,'' Technical Report, Dept. of EECS, University of California, Berkeley, CA 94720 (1983).

[Kols84]

Kolstad, Robert B., Campbell, Roy H., and Grunwald, Dirk C., ''Path Pascal User Manual,'' Documentation supplied with Path Pascal source code, Dept of Computer Science, University of Illinois, Urbana, IL (November 1984).

[Lesk84]

Lesk, M. E. and Schmidt, E., ''Lex - A Lexical Analyzer Generator,'' in *Unix Programmer's Manual - Supplementary Documents*, Computer Science Div., Dept. of EECS, University of California, Berkeley, CA 94720 (March 1984).

[Oste87]

Osterhaug, Anita, *Guide to Parallel Programming on Sequent Computer Systems*, Sequent Computer Systems, Beaverton, OR (1987). With Bibliography

TABLE OF CONTENTS