Porting Chorus to the PA-RISC: Building, Debugging, Testing and Validation

Ravindranath Konuru,
Marion Hakanson,
Jon Inouye,
Jonathan Walpole*
Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology

January 27, 1992

Abstract

This document is part of a series of reports describing the design decisions made in porting the Chorus Operating System to the Hewlett-Packard 9000 Series 800 workstation. This document describes the environment for building the Chorus kernel, the various kernel tests, and the debugging environment used for porting the Chorus operating system to the HP PA-RISC.

The information contained in this paper will be of interest to people who wish to:

- Use the PA-Chorus kernel for development and/or modification,
- Know about the build environment for Chorus kernel on PA-RISC,
- Know about the PA-Chorus approach to debugging,
- Know about the Chorus kernel tests, their build environment, and the modifications made to the kernel tests to port them to PA-RISC.

The aim of the document is to give a detailed overview of the various items mentioned above. The document does not delve into the specific tails of configuration like the values for various global macros and so on. For an in-depth understanding, one would have to study the sources. For the purposes of this document the word **kernel** and **nucleus** mean the same and will be used interchangeably.

*This research is supported by the Hewlett-Packard Company (HP), Chorus Systèmes, and Oregon Advanced Computing Institute (OACIS).

Contents

1 Introduction

This document is part of a series of reports describing the design decisions made in porting the Chorus Operating System to the Hewlett-Packard 9000 Series 800 workstation. This document describes the Chorus build environment for the kernel and kernel tests on PA-RISC, and our approach to debugging the Chorus kernel. The document is organized as follows. The PA-Chorus build environment and configuration management is detailed in section 2. The PA-Chorus development environment is detailed in section 3. Chorus facilities for debugging and the PA-Chorus approach to debugging are detailed in section 4. The details of the build environment for the kernel tests, a short description of the various kernel tests, and our modifications to the kernel tests to port them to PA-RISC are presented in section 5.

2 Build Environment for the Chorus kernel

Chorus software is organized as showin in fig 1. The directory **chorus_x.x** contains the sources for the Chorus *nucleus* version x.x. The directory **Nm**_x.x contains the sources for the Network manager version x.x. The directory **SVRx.x** contains the sources for a set of actors (*sub-system* implementing the UNIX System V version x.x interface. The directory **build** is used for configuration management and to build the main targets of the operating system i.e., the Chorus nucleus, Network manager, the Unix sub-system objects, the boot actor and the boot archive[IHKW92]. The standard **make** utility is used to *make* the various targets. By convention, there exists a **makeTarget** for each directory in the directory hierarchy. For a given directory d, the various targets that should be built in that directory, the compilation options for the targets, and the procedures to perform them are given in the file d/**makeTarget**. The **makeTarget**s form a hierarchy corresponding to the directory hierarchy. The *make* of all the targets of a sub-tree is performed by typing in

```
make -f makeTarget
```

in the root directory of that sub-tree. The **make** utility is recursively invoked in the all the sub-directories of that sub-tree thus building all the specified targets in that sub-tree. For each *make* in the sub-directories, the **makeTarget** in that directory is used as the input to the **make** utility.

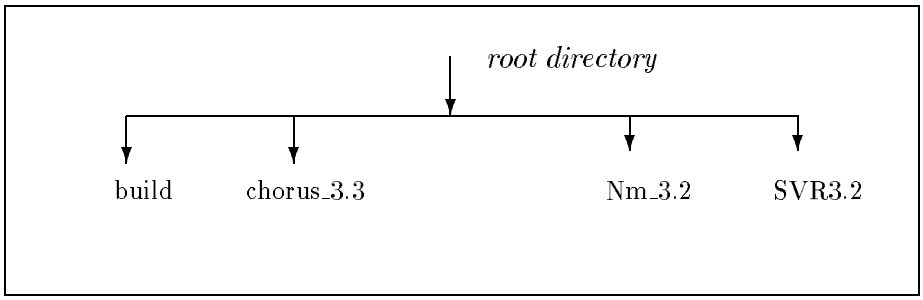

Figure 1: Chorus high-level directory organization

We will concentrate on the directories **builder** and **chorus_x.x** (**chorus_3.3** in the case of PA-Chorus) since these are the directories that are needed to build and test a Chorus kernel. The contents of the **builder** directory are detailed in section 2.1 and those of the **chorus_3.3** are detailed in section 2.2.

2.1 builder directory

The **builder** offers a single point from which all the targets that are required can be *made*. In addition, the creation of target specific configuration file[s] and rules for compilation for the target environment is performed through the files in this directory. This creation is part of the software tree initialization. The files in this directory are:

mkopt.gen : This file is the generic configuration file in which the the configuration parameters are specified under #ifdef flags for each target machine. The configuration parameters are global macros that are used in building the kernel and other targets. These macros should be approriately assigned based on the target machine environment. For PA-Chorus, the configuration parameters are specfied under flags HP800, HPUXLD, hpux, HPUX_CC, and HPUXCPLUSPLUS. Some examples of global macros defined in this file are MAKE, AR, CC, LD, BUILDDIR, PROC_DIR, and MACH_DIR.

mkrules.gen : This generic file specifies the compilation rules for various target environments. These rules are used by **make** in creating various targets. For example, the compilation rule to be applied for creating a `.o` file from a `.c` file is specfied in **mkrules.gen**. There is usually a set of such compilation rules for each target environment. For PA-Chorus, the compilation rules have been defined under the flag HP800.

syst.HP800 : This file contains the list of actors that should be loaded into the boot archive. This file has been created for PA-Chorus.

mk.HP800 : This file gives the procedures for creating the chorus symbol actor and the boot archive. The symbol actor is basically a composition of the symbol tables of the various boot actors enveloped as an executable image. Since each one of the boot actors is a seperate executable image, the additional work to load the symbol tables of the boot actors into main memory is performed through this file. We currently are not using the symbol actor.

makeTarget : This file is given as input to **make** . This file specifies the dependencies and the procedure to build various targets. To build a particular target or to perform a set of actions associated with a target name, the command has the following syntax:

make -f makeTarget [CONFIG=*conf-file*] [TARGET=*target*] [DEF=*suffix*] *targetName*

The macro CONFIG specifies the name of target specific configuration file. For PA-Chorus, this macro is set to "**confHP800**". This information is used only when *make*ing INITCONFIG. The macro DEF specifies the suffix for the file defining the boot actors which have to be loaded at boot time. DEF is the same as TARGET by default. For PA-Chorus, the DEF and TARGET are both set to "**HP800**". So specifiying the target name is sufficient for building the various target.

build/makeTarget forms the high level input to **make** utility. The file is organized such that a *make* of a specified target is achieved by recursively calling **make** with the target specific **makeTarget**. This process continues until the actual procedure to perform the target operations are reached. Then the specified actions are performed.

The following targets can be *made* from the **build/makeTarget**:

1. INITCONFIG : to initialize the builder tree. The files **mkopt.gen** and **mkrules.gen** are preprocessed to generate **mkopt** and **mkrules** that are specific to the target environment.

2. MKBOOT : to produce the boot archive.
3. CHORUS_INIT: to initialize the kernel source tree.
4. NM_INIT : to initialize the Network manager source tree.
5. SSU_INIT : to initialize the Unix subsystem source tree.
6. CHORUS : to compile the kernel sources and librairies.
7. NM : to compile the Network manager sources.
8. SSU : to compile the Unix subsystem sources.
9. SYSTEM_INIT: to perform SSU_INIT, NM_INIT and CHORUS_INIT targets.
10. SYSTEM : to perform SSU, NM and CHORUS targets.
11. GEN : to perform SYSTEM_INIT, SYSTEM and MKBOOT targets.

Initialization in the **builder**'s context refers to the creating the target specific files for configuration and compilation rules.

UTILS: This directory mainly contains the sources for the **make depend** utility.

2.2 chorus_3.3 directory

The sources for the chorus nucleus, boot image[1], and network device manager(ndm) are present in the **chorus_3.3** directory. The sources have the organization shown in fig 2. The **boot** directory contains the **boot** program and utilities for generating the boot archive. **include** contains the include files exported by the kernel for use by sub-systems and users. **kern** contain the nucleus sources. **lib** contains the sources needed for building the various libraries for kernel clients and for the kernel itself. **ktests** contains the sources for the kernel validation suite. The tests are discussed in section 5. **ndm** contains the sources for the network device manager. The **ndm** directory will not be discussed any more in this document.

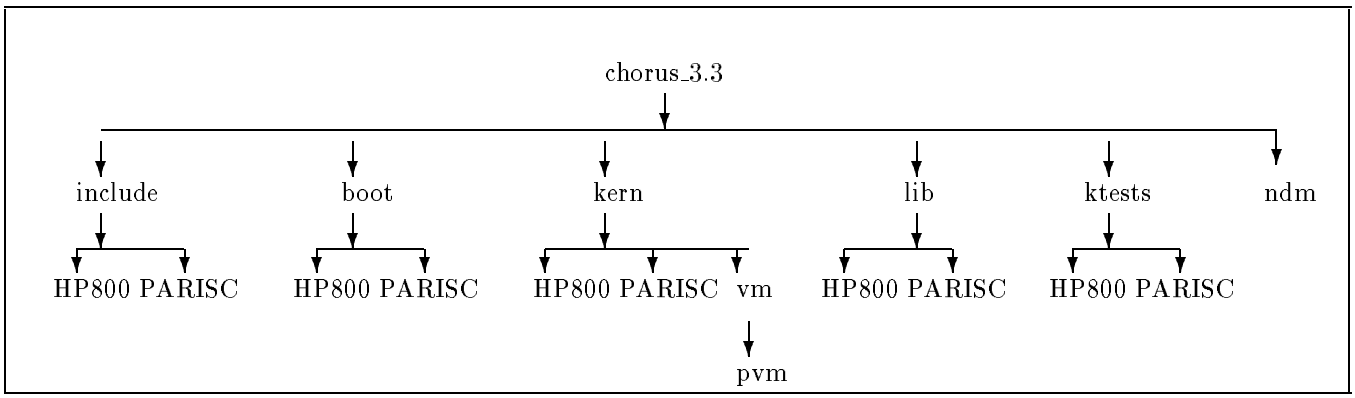

Figure 2: directory hierarchy of chorus_3.3

Each of the directories, i.e., **include**, **boot**, **kern**, and **lib** contain target independent files and two target specific directories: a machine dependent directory **HP800** and a processor dependent directory **PARISC**. These target dependent directories contain the target specific header files and

[1]boot image contains the entry point of boot archive[IHKW92].

sources needed to implement the Chorus machine dependent layer. The sources above the target specific directories in the directory hierarchy include the target dependent header files and use the symbolic constants exported by the lower layers in the target dependent directories.

The **chorus_3.3** directory, in addition to the afore mentioned directories, consists of files used for building the various targets for this directory hierarchy. They are:

mkrules.gen : contains compilation specific to the Chorus nucleus. Compilation rules to *make* various types of objects have been specified under the flag 'HP800'.

mkopt.gen : contains various configuration options specific to Chorus nucleus. Values for various configuration macros have been specified under the flag 'HP800'.

mkMkopt.c : offers a 'C' interface for automating the generation of starting addresses of various boot supervisor actors including the kernel.

makeTarget : The following targets can be *made* through this file:

1. INITCONFIG: performs similar operations as in **builder/makeTarget** for the **chorus_3.3** directory. **mkopt.gen** and **mkrules.gen** are preprocessed and target specific **mkopt** and **mkrules** are created.
2. INITMK: concatenates the output of **mkMkopt** to **mkopt**.
3. CHORUS: A recursive *make* is performed in the directories **include**, **lib**, **kern**, **boot** in that order.
4. SYSTEM: In addition to the CHORUS target, the a recursive *make* is performed in the **ktests** directory.

The names of the actual source files are not detailed in this document. Inspect the source tree for this information.

3 Development Environment

The hardware environment consisted of two HP 9000/834 machines:

- *louvre*: This was our development machine where all the software development was carried out.
- *orsay*: This was our test mule[2].

We did not have a boot monitor that was capable of accepting down loads of the OS, either from a LAN or an RS232 interface. So we used the following approach:

1. Build the Chorus kernel
2. Build the kernel tests
3. Generate the boot archive consisting of the **boot** image, the kernel image, and one or more kernel test executable images.
4. Peform an **rcp** of the boot archive from *louvre* to *orsay*.

[2]During the course of the port, there were suggestions (that weren't taken) that we should rename *orsay* as *yo-yo*

5. Reboot *orsay* with the boot archive.

The software environment consisted of:

Operating System : HP-UX 7.0 operating system.

C++ compiler : We installed the GNU CC version 2.0 on *louvre* and we used the standard **cc** available on HP-UX .

RCS : We found the version of the RCS available with HP-UX 7.0 did not have adequate functionality for our purpose. Therfore, We ported version 5.5 of RCS to the operating system.

During the development, we had a scenario where multiple users were editing the same work tree simultaneously. There was an overlap in the use of some files. Each developer needed his own set of debug compilation flags to debug his module[s]. To preserve consistency, we adopted the the following approach:

- A shadow tree was created for each developer that was an exact replica of the work tree except that every source file had a symbolic link to the corresponding file in the work tree. The various files dealing with configuration, compilation options, and making targets are separate for each developer thus giving him total control over the compilation and configuration options he wishes to use. This property allow parallel *make* s. Note that the **make** should be performed in the shadow tree only.

- A single work tree is maintained with an RCS directory for each sub-directory in the directory hierarchy. Each developer checks out with a lock so that no other user can check out the file until he is finished with it, checks in the potentially modified file and removes the lock. For the purposes of having a consistent source file during parallel *make* , each developer is required to copy the file to be modified into his shadow tree, perform the modifications, test the new file and then integrate it into the work tree. This allowed us to work with the older version of a file rather than work with a potentially inconsistent and undebugged module.

- The access permissions to all RCS files were set such that only the members of the PA-Chorus group could access the files.

The rest of the tools (for example, linker, assembler, C compiler, etc,) used were those that came with the operating system.

4 Debugging

None of the members of the PA-Chorus project had any prior experience in kernel debugging. Therefore, we could not totally figure out ahead of the port exactly what kind of debugging tools and information to be instrumented into the kernel. Most of our debugging instrumentation and tools evolved as their need arose during the port.

The main difference from debugging user-level programs and the kernel is that the kernel must be able to handle the errors it generates and, at the least, fail gracefully indicating the execution state at the time of the error. The user program on the other hand, can leave the responsibilityy to the operating system and use the debugging tools provided by the OS with confidence in them. The OS and the tools are the two entities on which he/she can rely on (usually, although OS and tools have been shown to have a few bugs and occasionally crash).

In kernel development, there is no under-lying layer that provides a debugging platform for the development other than the kernel itself. The kernel should either have the whole of the debugging layer within the kernel itself or have instrumentation for remote debugging. For example, an application programmer, when debugging is his/her application with **printf()**s, assumes that what is bieng printed is the actual state of his process and the **printf()** itself is not corrupting the memory. In kernel development, the debugging tools (for example, print) should themselves be thorougly debugged before they can be used with confidence. Since the kernel operates in the privileged mode, it is possible for the kernel to write to arbitrary locations without generating a fault at that time, but make a user program fault at some later time. It can create improper mapping of virtual addresses to physical addresses which would cause a strange and peculiar errors. In the case of user program, an access outside its protection domain is detected immediately in the form of a memory protection violation. No data is corrupted. In short, a lot of care is required when writing systems software rather than rely on debugging.

The port was performed in two major phases:

- A kernel with interrupts disabled,
- A kernel with interrupts enabled.

We adopted this approach from Chorus, since we were able to isolate the problems in the logic of code dealing with context switches, trap handling, system call interfaces, memory management layers without worrying about the complications of interrupt handling. Interrupt handling by itself tests another section of code, and we found it easier to use this incremental test methodology. By the time, we started debugging the kernel with interrupts enabled, we had a fair degree of confidence in our code with the exception of the code that gets executed during interrupt handling. This proved a crucial factor in narrowing down the location of the bugs.

There were four types of debugging that were used:

Hardware assisted debugging : Early in the boot sequence of the kernel or because of an unrecoverable error in the kernel, the kernel may not be able to save the execution state for analysis of the error. PARISC provides hardware support to store this execution state in non-volatile memory which can analyzed later. The target system HP 9000/834's boot monitor provided a hex-dump facility that also was quite useful. Pieces of **Tut** code were reused to provide the crash analysis facility.

Traces : Traces were the most important debugging tool. Chorus provides portable tracing modules for various kernel data structures like the scheduler, **actor**, **thread**, **port**, etc and various levels of tracing functions within the kernel. For more details see [Cho90]. The first problem was getting something to be written to and reading from the terminal. The second problem was writing reliable **putChar()** and **getChar()** function. The **putChar()** and **getChar()** function should work in both virtual and physical memory modes. Since all the portable code of the kernel performed its printfs and scanfs in terms of these two calls, we could use the traces when these functions were implemented. The trace level can be set dynamically and also through the debugger and proved a valuable tool in isolating module testing.

Kernel Debugger : A kernel debugger was implemented as part of the *Supervisor* interface. The implementation of the PA-Chorus kernel debugger is presented in [KHIW92]. It is a low-level debugger and does not provide symbolic debugging. However, it proved very useful during

the port. The postmortem tools mentioned below have been integrated through the debugger user interface.

PostMortem tools : Almost all of these tools are integrated through the kernel debugger. All of the tools maintain a rolling history of a certain event and the relevant information about it. Information was stored about traps, interrupts, context switches, cause of context switches (premption or voluntary), and the execution states of the threads involved at the time of the context.

We used traces extensively during the first stage of the port. However, with the interrupts turned on, By the time we made a length trace to the terminal, the round robin interval of 10ms was long past. We never could do any useful work. Relying on post mortem tools became a necessity in this case.

A brief description of the implementation of **PutChar()** and **GetChar()** is now presented.

The trace functions used in the kernel used **printf()** that was specifically written for the kernel. This code was written in a portable manner such that the only machine dependent functions that needed to be implemented to have a working kernel **printf()** were **PutChar()** and **GetChar()**. These two functions have been implemented in the following manner. The functions **PutChar()** and **GetChar()** call **SupPutChar()** and **SupGetChar()** respectively. **SupGetChar()** is implemented in terms of **SupPollChar()**. **SupPollChar()** returns a character from the terminal (console) if one is available. Otherwise, it returns 0. It is a non-blocking function. **SupPutChar()** and **SupPollChar()** are both implemented by calling a function **SupIodcCall()** with appropriate operation type. All the **Sup*()** routines are implemented in **kern/PARISC/svBoard.c**. **SupIodcCall()** in turns calls 'write' and 'read' functions exported by the Input Output Dependent Code (IODC) of the console device[IHKW92]. The IODC is called with virtual memory and the interrupts disabled. The execution state is restored before returning from the **SupPollChar()** and **SupGetChar()** routines. For more details on what an IODC is and how the concept works, see [JBO86, Hew90].

It is pertinent to mention here that the **printf()** routine provided in the kernel is not as generic as a **printf()** available to the user program. In fact, the maximum number of parameters to the kernel **printf()** is 8. We found this limit the hard way when a trace had more than 8 parameters. Chorus should specify that a limit exists on the number of parameters in their implementation specfication guide.

5 Testing and Validation

We adopted the same approach as Chorus for validating the PA-Chorus kernel, i.e., pass the kernel test suite. This section presents the kernel test build environment, a brief description of the various kernel tests, and the modifications we had to do to port the kernel tests.

5.1 The Kernel test build Environment

Chorus provides a set of kernel tests that test the kernel functionality and interface. The tests run as **actor**s on the Chorus kernel. The test suite sources are organized under the **chorus_3.3/ktests** directory. Siimlar to the kernel source tree organization, the kernel test sources are divided into 'target-independent' and 'target-dependent' parts. The target dependent files are placed in **HP800** and **PARISC** directories.

Each of the directories, i.e., **ktests**, **ktests/HP800**, **ktests/PARISC**, contain a file **makeTarget** used for building the kernel tests. A *make* in the **ktests** directory compiles all the objects in the directory and recursively makes in the machine dependent sub-directorys to create executable images. These **makeTarget**s include the configuration options and compilation rules (**mkopt** and **mkrules** from the **chorus_3.3** directory. A *make* in the **ktests** directory has the following syntax:

make -f makeTarget *targetName*

The main names that can be used as *targetName* can be any of the following:

- SYST : build kernel test executable images such that they run as supervisor actors. Basically the the object modules are linked with the supervisor actor library and the different starting virtual address that does not overlap with the address ranges of other supervisor actors or the kernel. In addition they are created with the -N option that allocates the code and data spaces contiguously (The data space begins at the next page aligned address after the last code address).

- USER : build kernel test executable images such that they run as user actors. These actors are linked with Chorus user Actor library. The starting address of the text segment and the data segment are the default values provided by the linker, 0x800 and 0x40000000 respectively.

If no *targetName* is supplied, then both types of actors are *made*. There are other *targetName*s that can be supplied to **make** that build sub-targets in the **ktests** tree. However, the purpose of this document is only to give an overview of the overall organization thus giving an initial start up rather than to form a substitute for reading the source tree by dwelling deeply into the values for various configuration options and the different sub targets that can be made from each of **makeTarget** in the whole directory hierarchy.

The **ktests** directory also contains provisions for defining tests' specific **mkopt** and **mkrules**. However, we did not need to use them.

5.2 The Kernel tests

The kernel tests sources in the **ktests** are organized in the following manner:

- All the definitions and declarations for a test are placed on a test specfic **.h** file. Defintions and configurations that are global to all tests are placed in a file (**ktests/kt_conf.hxx**) that is included by all the test sources.

- All the procedures needed by a test are place in a test specfic library. Those that are generic are divided again into virtual memory related functions and others. These functions are placed in seperate files.

- Most other files in the **ktests** directory correspond to specific tests, usually one per test. For example, **kt_act01.cxx** contains the source for the test `kt_act01`.

- The target dependent functions and values of symbolic constants are defined in the target dependent sub-directories **HP800** and **PARISC**. All the functions that are necessary for the kernel test suite have been implemented for PA-Chorus in the sub-directories of **ktests**.

We now present a short functional specification of the various tests in the kernel test suite. For a more detailed specification, see [Ger90].

kt_act01 : This is considered the simplest test in the kernel test suite. It serves the dual purpose of validating the kernel test environment and all the necessary configuration features are more or less in place. The test performs a few actor specific system calls. No threads are created. The test should pass in the both kernel and user spaces, i.e., when linked/loaded as a supervisor actor or as a user actor.

kt_act02 : Test actor related functions at kernel configuration limits. The test should pass in both kernel and user spaces.

kt_excep01 : Test the kernel exception processing functionalities. The test should pass in user space. It is not intended for the kernel space.

kt_grp01 : Test group broadcast related functions. The test must pass in both user and kernel spaces.

kt_ipc01 : Test the functions related to inter process communication. The test requires two actors: a client and a server. The test should pass in both kernel and user spaces.

kt_vm01 : Test the page pool operations. The test should pass in both kernel and user spaces.

kt_vm02 : Test the kernel virtual memory functionality up to the kernel configuration limits and test the IPC using virtual memory functions. This test involves 3 actors: a client, a server and a mapper. This test should pass in both kernel and user spaces subject to constraints on the placement of server and mapper actors with respect to the client actor.

kt_sync01 : Test the *sempahore* and *mutex* related functions. The test should pass in both kernel and user spaces.

kt_thr01 : Test basic thread related functions along with the *semaphore* and *mutex* interface. The test should pass in both kernel and user spaces.

kt_thr02 : A more difficult test performing thread related functions up to the kernel configuration limits. This includes creating the maximum number of threads and ports as specifed by the configuration parameters. The test should pass in both kernel and user spaces.

kt_ui01 : Test the unique identifier (UI) related functions up to the configuration limits. The test should pass in both kernel and user spaces.

kt_svCalls01 : Test a part of the supervisor actor interface related to time-out, exception and trap handling. This test should pass in the kernel space. It is not intended to run in the user space. This test was written for PA-Chorus.

5.3 Modifications to the portable layers of kernel tests

There were few modifications to the 'portable layers' of the kernel test suite. These involved changing the assumptions about user address space range, stack direction, and thread's system stack size. There was one instance of each of the above assumptions. A couple of procedures were not generic enough to handle the total range of their formal parameters. The modified files are **kt_main.cxx**, **kt_libVM.cxx**, **kt_lib.cxx**, and **kt_vm1.cxx** in the **ktests** directory.

6 Remarks

The chorus source organization was a big plus in the process of porting. It was very easy to identify the location of various functions.

As there was not enough documentation on the kernel tests, it took us a long time to figure out how the tests are supposed to run, and the interface that needs to be implemented by the target-specific functions. However, compared to writing our own kernel validation suite, the work that was involved in porting the kernel tests was negligible.

The shadow tree approach to development proved very beneficial. However we would have liked to have a tool that could take snap shots of all the tree and retreive it symbolically.

7 Acknowledgements

We thank Jean-Jacqués and Olivia Giffard of Chorus Systémes for expediting the process of obtaining the sources of kernel tests.

References

[Cho90] CHORUS Kernel v3.2 Implementation Guide. Technical Report CS/TR-90-5, Chorus Systèmes, 1990.

[Ger90] Jean-Jacques Germond. Specifications of the CHORUS/MiX Kernel v3.2 Test Suites. Technical Report CS/TR-90-27, Chorus Systèmes, 1990.

[Hew90] Hewlett-Packard. *Precision I/O Architecture Reference Specification*, 0.93 edition, January 1990.

[IHKW92] Jon Inouye, Marion Hakanson, Ravindranath Konuru, and Jonathan Walpole. Porting Chorus to the PA-RISC: Booting. Technical Report CSE-91-4, Oregon Graduate Institute, 1992.

[JBO86] David V. James, Stephen G. Burger, and Robert D. Odineal. Hewlett-Packard Precision Architecture: The Input/Output System. *Hewlett-Packard Journal*, 37(8):23–30, August 1986.

[KHIW92] Ravindranath Konuru, Marion Hakanson, Jon Inouye, and Jonathan Walpole. Porting the Chorus Supervisor and Related Low-Level Functions to the PA-RISC. Technical Report CSE-91-6, Oregon Graduate Institute, January 1992.