SCOTT: The Southampton COFF Tools for Transputers

Denis A NICOLE and Rajini SIVARAM
Department of Electronics and Computer Science, University of Southampton, United Kingdom

Abstract. We have built a software development toolset for T800 and T9000 series transputers suitable for compilers, including ACE eXpert and `gcc` implementations, using the COFF object format. Our toolset includes a bootstrap, loading and configuration environment, an interactive debugger and implementations of the PVM and MPI message-passing environments. The `tcl` scripting language is extensively used.

1. Introduction

As part of our work within the Esprit GPMIMD (P5404) project, we have developed a set of tools for loading, running and debugging transputer programs compiled to the Common Object File Format (COFF). At the moment, the only development system producing such code is that produced by ACE/PACT in the Netherlands as their eXpert Parallel Development System; this suite includes C and FORTRAN compilers. A C compiler [1] based on gcc has recently been released; it feeds into the ACE assembler or into its own assembler and linker.

The system supports two high level standard parallel programming environments; PVM 3.3.7, including `xpvm`, and MPI. PVM on the T9000 system is fully integrated into the standard workstation network PVM release; PVM jobs may be run on the T9000s alone or on mixed sets of T9000s and workstations.

In this release, our tools support mixed networks of T800 and T9000 transputers but require explicit `tcl` scripts for the processor bootstrap and program loading. The `tcl` scripts have to be manually designed for each distinct network. A prototype parser for the INMOS NDL based on SPOC [2] has also been developed; it may be configured to generate appropriate TCL scripts either for module bootstraps or for user link configuration.

The entire package is based around the `tcl` [3] scripting language and the `tk` suite of X windows commands. We have implemented versions of the `wish` windowing shell for both T800 and T9000 processors; some of our tools also require `wish` to be running on the workstation host.

As released, the tools are configured for X windows workstations; they were developed on a Sun 4 running SunOS. They are readily portable to any other UNIX host running X windows. Versions of `wish` and `gdb` exist for PCs under DOS, Windows and WindowsNT and the basic tools, including the debugger, can be set up to run under these environments.

2. Overview

There are five main components to the SCOTT tool set:

- The extended `iserver`.

- An enhanced WIndowing SHell, `wish` which runs on a T800 series transputer connected to the host; it may also be configured to run on a T9000 after the processor has been bootstrapped. This provides the bootstrap and loading environment for the T9000s and for T800 networks.

- A port of the GNU debugger `gdb` for transputers which runs on a workstation host and communicates with debugging stubs on T800 and T9000 series transputers. A full screen graphical interface for `gdb` is also available.

- A port of the PVM daemon `pvmd` which runs on T800 or T9000 transputers. This can spawn tasks on either the T9000s or the workstation host. Together with the PVM library, this allows PVM jobs to be run on the T9000s either from the PVM console or from `xpvm`.

- A port of the Message Passing Interface (MPI) library supporting SPMD programming.

The `wish` shell is an established standard for tool integration; it has, for example, been selected by the PTOOLS consortium. `tcl` and the `tk` toolkit were developed at the University of California, Berkeley and comprise an elegant command scripting language `tcl` and a Motif-style library `tk`. The `wish` shell incorporates `tcl` and a `tcl` interface to the `tk` library; it provides an easy way to build command scripts with a full Motif-style interface. It is extensively documented [3,4]. We have ported `wish` to T800 series transputers and added a small number of key features in order to support the configuration, booting and debugging of T800 and T9000 transputer networks.

A wide variety of program development tools have been developed as `wish` scripts. These include:

- `mxedit`: An X based editor.
- `tkcvs`: A graphical front end for the `cvs` concurrent version control system.
- `tgdb`: A full graphical front end for the gdb debugger.
- `xpvm`: A graphical front end and trace display for the PVM parallel programming environment.

Overall, wish is a natural framework for the development of a modern suite of program development and bootstrap tools. It is the standard scripting language for rapidly implementing X windows interfaces; many applications have already been developed; integration between applications is explicitly supported and the scripting language has the necessary power to implement a bootstrap system.

3. The Extended Iserver for X

For the moment, it is necessary to use a non-standard host server, included within the release, with our COFF transputer tools. This is principally because of serious bugs in the ACE library support for the INMOS `iserver`, requiring us to use ACE `xserver` file I/O. A second important reason is the need for enhanced socket support. Our system makes

extensive use of sockets for debugging, X windows and PVM traffic with the host; the standard INMOS iserver is unable to support sufficiently many sockets. Our server provides the functionality of :

- A traditional INMOS iserver.
- The file I/O extensions of the ACE/PACT `xserver`.
- Some additional support for NFS directories through `xserver`.
- BSD Sockets partly compatible with the INMOS socket extensions.

Our server uses the INMOS `linkops` connection manager and can thus be compiled to support a collection of different physical transputer (OS or DS) links. We have also developed a technique for including legacy `linkIO` binary libraries for special devices whose sources cannot be modified for linkops. If a `linkIO` module is available only in object form, and defines the functions *`OpenLink, CloseLink`* etc. as required by the older versions of `iserver`, it cannot be linked to the new, since the function names will clash with the functions used by `iserver`. The utility `tostatic` has been implemented to overcome this problem It is invoked as

```
tostatic ObjectFile Symbol1 Symbol2 ...
```

This program searches the symbol table of *`ObjectFile`* for the symbols specified, and converts the symbols to static by altering the `n_type` field of the symbol. The program assumes that *`ObjectFile`* is in the SPARC `a.out` format.

The old `LinkIO` module can thus be linked to the new `iserver` by first linking it with a `linkops` stub library and then hiding the clashing *`OpenLink, CloseLink`* etc. linker names using the `tostatic` program.

The transputer-hosted tools in SCOTT are themselves built with the ACE/PACT compilers; bugs in the `iserver` support in the ACE/PACT libraries prevent them running under a standard `iserver`. For T9000 bootstrap, the `wish` tool can be recompiled without X support as a simple `tclsh` shell; none of the X features are essential to the bootstrap scripts. It will then run under the standard ACE/PACT `xserver`. It will also run under an MSDOS based `xserver`. The multiple server feature and the `gdb` tool do, however, require our socket extensions.

4. `tcl` for the transputer

4.1 Introduction

`tcl` and `tk` have been ported to transputers and a few extra functions have been added to support booting and execution of programs on T800s and T9000s. This section describes the changes that have been made to `tcl`/`tk`, the limitations imposed by the transputer implementation, and the extended functionality when running on T800s. These extensions consist of a minimal set of additional features to permit efficient full control of both OS and DS links.

The `tcl` message primitives are very simple; they offer the ability to send or receive a message on a link or virtual channel. More interesting is the provided concept of *binding* an

additional protocol layer onto a link or virtual channel. Thus a virtual *virtual channel processor* (VCP) may be bound to an OS link; it is then possible to communicate using virtual channels through a C100 link converter attached to the link. Similarly, an *iserver* can be bound, either to a virtual channel of the virtual VCP or to another OS link. The virtual iserver is not a real `iserver`; it simply multiplexes any iserver traffic to the host `iserver`. The multiplexor ensures that the reply `iserver` packet is sent to the originator of the command.

4.2 Virtual VCP

The Virtual VCP currently supports communication between the T800 running TCL and a network of T9000s which is connected to the T800 through a C100 protocol converter. Messages can be sent to or received from a T9 using the additional TCL commands `vcpsend` and `vcprecv`. Virtual channels are dynamically created and deleted for each command. All messages are packetized and sent in the format required by the C100. Messages can be stored either in `tcl` variables or in a file. Both blocking and non-blocking communication are supported, with an option to specify time-outs.

Whenever a Virtual VCP is initialised on a link, three threads are created: a sender thread, a receiver thread and a timer thread.

The sender thread sleeps until a request to send either a message or an acknowledgement is queued. The process that queues the request wakes up the sender thread if it is asleep. Whenever an ACK is to be sent, the sender thread sends the ACK and if the ACK is for the last packet of a message, wakes up the receiver. If any time-out was associated with the request, the time-out is cancelled and the timer thread informed. When a message is to be sent, the first packet of the message is sent immediately, and the rest of the message is queued. The subsequent packets are sent only after an ACK is received.

The receiver thread waits on the link until any message or ACK arrives. If an ACK is received for the last packet of a message, the receiver thread wakes up the sender. If the send request had a time-out associated with it, the time-out is cancelled and the timer thread informed. The sender thread is woken up if necessary, to service pending send requests. For ACKs of other packets, the sender thread is informed, so that the next packet of the message can be sent.

The sender and receiver threads use two channels to inform the timer thread about new time-outs or time-outs that have been cancelled due to the successful completion of a send or receive command. The timer thread performs a `ProcTimerAlt` on the smallest time-out, and these two channels. Whenever a time-out occurs, the corresponding `send` or `receive` command is cancelled and the `tcl` commands associated with the time-out are executed.

When a `vcpsend` command is executed, a message is created, the headers are formed, and the sender thread woken up to send the message. The headers are stored in a VIRTUAL_CHANNEL_INFO structure, which is retained only until the command is complete, thereby allowing dynamic creation of virtual channels. The thread that executes `vcpsend` goes to sleep, and is woken up either after the message is successfully sent, or on a time-out.

When a `vcprecv` command is executed, a request is posted if the message hasn't already arrived. The thread then goes to sleep. The receiver thread receives the message into the user thread's address space and wakes it up. If a packet of the message has already arrived, it is copied into the user's address space, and the thread goes to sleep until the entire message has been received and acknowledged.

All non-blocking send and receive commands are implemented using additional threads to perform blocking sends and receives. All VCP functions are executed at high priority, and it is assumed that the VCP threads will never be pre-empted.

4.3 Servers

The virtual `iserver` built into our `wish` can be used to boot and execute programs on another T800 or T9000. If a program needs to be run on a T800, it is booted by writing the entire bootable file down the link. T9000 bootables are assumed to be in the INMOS bootable file format and are booted using a virtual channel with the single byte header 0. All the server requests except one are passed to the host server without any modification. The SP_COMMAND function is not sent to the host server; it returns the command line parameters of the `tcl iserver` command.

The `coffserver` differs from `iserver` only in the boot procedure. For booting programs on T800s, `coffserver` first writes the primary and secondary bootstrap codes, followed by the COFF parameters, and then the various sections of the COFF file. For booting programs on a T9000, `coffserver` sends a single bootstrap routine, followed by COFF parameters and the sections of the COFF file. `coffserver` assumes that the T9000 has already been configured and the primary bootstrap loaded over the control links. This server also uses a virtual channel with header 0 for booting. The bootstrap routine loads the COFF file, sets up the user's virtual channels and then transfers control to a modified `_CRT_enter` routine.

Additional features of the `iserver` are invoked by the `-window` and `-bind` options. The `-window` option causes standard I/O to be redirected via sockets. The argument of this option is a host command, for example "`xterm -e telnet`", to process the standard I/O. Additional parameters pass the port numbers to this command. Overall, the effect of the `-window` option is to associate standard I/O for this virtual `iserver` with a new window.

Normally, the `iserver` commands block until the associated client program terminates. When the server is invoked with the `-bind` option, the main `tcl` thread of execution continues in parallel with the `iserver`. The argument to `-bind` is executed when the client program terminates.

4.4 Limitations

Since pipes and the creation of processes using `fork` and `exec` are not supported in the transputer C run-time libraries, `tcl` functions that require the use of any pipe or fork are not implemented on the transputer. This includes `open` on pipes, `exec` and process commands such as `pid`. In order to support the execution of shell commands from `tcl`, the `exec` command is implemented in a limited way. The `exec` command is parsed, string redirection performed, and the resulting command is passed to the host UNIX shell. The output of the command is redirected to a file and later read by `tcl`. The modification of environment variables is not supported, nor is execution of commands in the background.

4.5 Manual pages

vcpsend(n) Tk Commands vcpsend(n)

NAME
vcpsend - Send a message over a virtual/physical channel

SYNOPSIS
vcpsend header vchan ?options?

vcpsend link ?options?

OPTIONS
-data data Data is provided in the Tcl variable data in the form of a string. The format for specifying data is element?.element. ... element? . Each element in this string is assumed to be 32-bit if it ends in the character l, 16-bit if it ends in s, and 8-bit otherwise. Hexadecimal numbers should be preceded by 0x and octal numbers by the digit 0.

-file filename Data to be transmitted is in the file specified by filename.

-bind command Perform a non-blocking send and execute the Tcl/Tk command specified in command when the send is complete.

-time-out time command
If the send cannot be completed in the number of microseconds specified by time, abort the send command and execute the Tcl/Tk command specified by command.

DESCRIPTION
This command is used to send a message over a virtual channel or a physical link. When a message is to be sent to a virtual channel, header specifies the virtual channel address of the destination. It is of the form link.subheader?.subheader ...? . Each of the subheaders is a decimal number, and is assumed to be a two-byte header, if its value is greater than 255 or if it starts with a zero. Otherwise, it is assumed to be a single-byte header. link can take values l0, l1, l2 and l3. vchan specifies the virtual channel address where the acknowledgement is expected. It has the format ?link.?subheader . The message is packetized and sent in the format required by C100. The command is complete only when the acknowledgement of the last packet is received. Virtual VCP should be initialized by running vcpinit on link before this call is made.
When a message is to be sent over a physical channel, link specifies the linknumber on which the message should be sent. The command is complete when the message has been successfully sent over the channel.

SEE ALSO
run, vcprecv

NAME

vcprecv - Receive a message from a virtual/physical channel

SYNOPSIS

vcprecv vchan header ?options?

vcprecv link count ?options?

OPTIONS

-data data Store the message received in the Tcl variable data. The message is stored in the format element?.element. ... element? . Each element in this string is a hexadecimal number preceded by 0x and represents one byte of the message.

-file filename Receive the message in the file specified by filename.

-bind command Perform a non-blocking receive and execute the Tcl/Tk command specified in command when the receive is complete.

-time-out time command
If the receive command cannot be completed in the number of microseconds specified by time, abort the command and execute the Tcl/Tk command specified by command.

DESCRIPTION

This command is used to receive a message over a virtual channel or a physical link. When a message is to be received from a virtual channel, vchan specifies the virtual channel address and is of the form link.subheader. The subheader is a decimal number, and is assumed to be a two-byte header, if its value is greater than 255 or if it starts with a zero. Otherwise it is assumed to be a single-byte header. link can take values l0, l1, l2 and l3. header specifies the virtual channel address where the acknowledgement is to be sent. It is of the form ?link.?subheader?.subheader ... ?. The message is expected in the format used by C100. The command is complete when a packet is received with the EndOfMessage flag, and the acknowledgement of that packet is sent. Virtual VCP should be initialized by running vcpinit on link before this call is made.
When a message is to be received over a physical channel, link specifies the linknumber over which the message should be received. count specifies the number of bytes that are to be received. The command is complete when count bytes are received over link.

SEE ALSO

run, vcpsend, inspect

NAME
inspect - Check if any messages have arrived

SYNOPSIS
inspect link variable

DESCRIPTION
This command checks if any messages have arrived on a physical link, on which the virtual VCP has been initialized. If any packets have arrived which have not been claimed, the first bytes of the headers of the packets are returned in a list. The value of the Tcl variable variable is set to this list of headers.

SEE ALSO
run, vcpsend, vcprecv

NAME
run - Execute an in-built function

SYNOPSIS
run function ?parameters?

DESCRIPTION
This command supports the execution of some in-built functions. The return value of this command is the value returned by the function.

FUNCTIONS
vcpinit link
Initialize virtual VCP on the specified physical link. link is the link number preceded by the character l. Raw channel IO cannot be used on the link after VCP has been initialized. vcpsend and vcprecv can be used to send or receive messages on any virtual channel mapped to this physical link. This function returns success/failure.

vcpquit link
Terminate the virtual VCP on link.

localwrite address value
Write the hexadecimal byte specified by value at address.

servinit chan ?-window wincmd? ?-bind command? parameters

coffserver chan ?-window wincmd? ?-bind command? parameters
Run iserver or coffserver on the virtual or physical link specified by chan. Physical links are specified by the link number preceded by l. Virtual links are specified by the four tuple outheader outvchan invchan inheader. Server requests are received on the virtual channel invchan and acknowledgement is sent to inheader. Reply packets are sent to the virtual channel address outheader and acknowledgement is expected at outvchan. outheader and inheader have the format link?.subheader. ...?subheader. Each subheader is a decimal number and is assumed to be 16-bit if its value is greater than 255 or if it starts with 0. Otherwise it is assumed to be 8-bit. outvchan and invchan have

the format link.subheader. If the -window option is specified, the shell command specified by wincmd is executed to create a window, and all standard IO is redirected to the window. A typical window command uses an xterm running telnet. The parameters to telnet can be specified using the variables %host and %port. These will be set to appropriate values before wincmd is executed. If the server is to run in non-blocking mode, -bind option should be specified. The Tcl/Tk command specified by command will be executed when the program has terminated. parameters specifies the parameters to the server as well as those of the application. It has the same format as iserver parameters and all parameters that do not belong to the server are passed to the application program. This function returns serverid, which can be used to terminate the server, if required.

servquit serverid
Terminate the server specified by serverid. serverid should be the value returned by a servinit or coffserver call.

NAME
vcpxfer - Send a list of messages over a virtual channel and receive the reply messages.

SYNOPSIS
vcpxfer sendchan sendackchan recvchan recvackchan ?options?

OPTIONS
-data data Data is provided in the Tcl variable data in the form of a string. The format for specifying data is message?.r.message.r. ... message? . Each message has the format element?.element. ... element? . Each element in this string is assumed to be 32-bit if it ends in the character l, 16-bit if it ends in s, and 8-bit otherwise. Hexadecimal numbers should be preceded by 0x and octal numbers by the digit 0.

-time-out time command
If the messages cannot be transferred in the number of microseconds specified by time, abort the command and execute the Tcl/Tk command specified by command.

DESCRIPTION
This command is used to transfer a list of messages over a virtual channel. The reply for each message is received and ignored. sendchan specifies the virtual channel address of the destination. It is of the form link.subheader?.subheader ...? . Each of the subheaders is a decimal number, and is assumed to be a two-byte header, if its value is greater than 255 or if it starts with a zero. Otherwise, it is assumed to be a single-byte header. link can take values l0, l1, l2 and l3. sendackchan specifies the virtual channel address where the acknowledgement is expected. It has the format ?link.?subheader . recvchan specifies the virtual channel address where the reply is expected, and has the same format as sendackchan. recvackchan is the virtual channel address to which the acknowledgement of the reply message is sent, and has the same format as sendchan.

```
     The message is packetized and sent in the format required by
     C100.  The  command  is  complete when all the messages have
     been sent and their replies received.  Virtual VCP should be
     initialized  by  running vcpinit on link before this call is
     made.

SEE ALSO
     run, vcpsend, vcprecv
```

5 Porting of GDB to T805 and T9000

We have ported `gdb` Version 4.12 to the T805 and the T9000. Apart from the implementation of the machine dependent routines, including the disassembler, the porting also involves the modification of the symbol manipulation routines based on COFF, as the ACE compilers do not conform to the traditional COFF specifications. Our port of `gdb` uses remote debugging over a TCP/IP connection, making use of our extended socket library for the transputer.

Currently, we are set up to run `gdb` on the host workstation; it communicates with the transputer program being debugged via sockets to a stub built into the COFF start-up code. Support for this kind of remote debugging is a standard feature of `gdb`; we have implemented the necessary stubs for the transputer. Some effort has been expended to improve the performance of source-level single stepping.

We have also ported the `tgdb` graphical interface to our version of `gdb`. This is a large `wish` script that uses `gdb` to provide a full windowed graphical debugger. Separate windows can display the source code, disassembly, watch variables, machine registers and memory dump. Breakpoints can be set and cleared by mouse clicks.

6. Implementation of PVM on T9000

This section describes the implementation of PVM on a network of T9000s. In summary, we exploit the standard MPP machine model within PVM. This is typically used for, for example, an Intel hypercube hosted from an Intel workstation. Apart from the need to provide efficient transputer-transputer communication, the main new feature of our implementation is support for a host architecture which is different from the MPP node architecture.

A new PVM architecture, T9000, has been introduced as an MPP machine. A single PVM daemon runs on one of the T9000s; this supports PVM tasks running on all the T9000s as well as on the host machine. It uses the host's IP address. All tasks are spawned on a T9000 by default, unless the `PvmMppFront` flag is set for the task, in which case the task is spawned on the UNIX host front-end. Tasks which have been started as UNIX processes on the front-end are allowed to connect to an already running PVMD by calling `pvm_mytid()`. Node-to-node messages are sent directly between T9000s using virtual channel communication routines. Socket communication is used by PVM tasks running on the front-end to communicate to the local PVMD, and also for communication between PVM daemons.

This standard architecture means that the transputer system and its host can sit as a standard machine in a PVM workstation cluster; they can be controlled by an unmodified `xpvm` or `pvm` console anywhere in the cluster.

By default, the `pvm` console and the `xpvm` graphical interface run on the front-end. The same debugger, `gdb`, and graphical interface, `tgdb`, is supported on both the front-end and

the T9000. PVM applications running on a network of heterogeneous computers can spawn tasks on the T9000s and communicate to the tasks using any of the PVM communication library routines.

7. Implementation of MPI on T9000

The Portable MPI Model Implementation from Argonne National Laboratory and Mississippi State University, has been ported to the T9000. The PVM implementation of Chameleon, a portable message passing system developed by William Gropp and Barry Smith, was first ported to the T9000, and MPI layered on Chameleon was then implemented.

A new architecture, T9000, has been added to both Chameleon and MPI. A few changes had to be made to the sources to support UNIX calls including `symlink`, `mkdir`, `getcwd` and `chdir` which are not supported on the T9000. These have been added to invoke appropriate routines on the front-end. `fork()` has been replaced with `system()` to start-up UNIX processes on the front-end.

8. Communication Benchmarks

The PARKBENCH suite of benchmark programs provides low-level benchmarks to characterise the basic communication properties of a parallel computer by measuring the parameters RINF and NHALF for communication. Within the GPMIMD project, we have measured the communication properties of a variety of high performance parallel computers and compared them with those of our PVM implementation on early T9000 Silicon. The results for the T9000 are considerably worse that would be expected for product-quality compilers targeting fast, product quality Silicon; a large number of patches have been applied to the code generator to cope with the early T9000s.

The COMMS1, or pingpong, benchmark measures the basic communication properties of a message-passing MIMD computer. A message of variable length, N, is sent from a master node to a slave node. The slave node receives the message into a FORTRAN data array, and immediately returns it to the master. Half the time for this message pingpong is recorded as the time, T, to send a message of length, N. The time as a function of message length is fitted by least squares using the parameters RINF and NHALF to the following linear timing model:

$$T = (N+NHALF)/RINF$$

The machines under test were:

- Silicon Graphics Indigo Workstation Cluster (SGI)

  CPU: MIPS R2000A/R3000 Processor Chip Revision: 3.0
  CPU speed: 33 MHz
  Interconnect type: 10MHz Ethernet
  Operating system: IRIX 4.0.5H
  Compiler flags: F77: -O2

- Meiko CS-2

  CPU: SuperSparc
  CPU speed: 40 MHz
  Interconnect type: Meiko Elite/Elan
  Operating system: SOLARIS 2.3
  Compiler Flags: F77: -fast (-O3 -dalign -fnonstd -cg92)

- Cray T3D

  CPU: DEC Alpha 21064
  CPU speed: 150 MHz
  Compiler flag F77: -O 1

- Parsytec GC/PP

  CPU: Motorola Power PC 601
  CPU speed: 80 MHz
  Operating system: PARIX 1.2
  Compiler F77: -O3

- T9000

  CPU: T9000 GAMMA D02
  CPU speed: 20 MHz
  Operating system: SCOTT 1.0
  Compiler flags: F77:

The results were:

	SGI	CS-2	T3D	GC/PP	T9000
RINF (Mflop/s)	0.429	8.43	26.2	2.55	1.83
NHALF (byte)	2668	3190	3878	784	2757
Start-up (μs)	6223.9	378.6	147.8	307.4	1504.3

The `COMMS1` benchmark was recoded to use packing sends and receives as that functionality was added to PVM during the life of this project. They also avoid substantial copy overheads. Several of the systems supported only old PVM versions and could not run the updated code. Those that could, however, gave a realistic evaluation of their underlying communication capabilities.

The following runs are on the machines that have the function implemented. Although this code is not strictly part of the PARKBENCH suite it is included for interest as many PVM implementations are optimised for doing packing sends and receives.

	SGI	T3D	T9000
RINF (Mflop/s)	0.462	42.1	3.96
NHALF (byte)	2345	1677	278
Start-up (μs)	5080.1	39.9	70.2

It can be seen from these results that the Transputer PVM implementation offers excellent start-up and good use of the (single channel) bandwidth of the T9000 communications system.

9. Future Directions

There are two obvious development tracks for our toolset. Firstly, several partners in the GPMIMD project have shown interest in using the T9000 as a communications processor for another high performance compute processor, such as a DEC Alpha or a Power PC. Almost the entire toolset can be used in this environment; SCOTT will boot, control and debug such mixed systems using existing versions of `gdb` and `gcc` or `g77` for the target compute processors.

A second development would replace the ACE compilers with the gcc port for transputers mentioned above. Appropriate assemblers and linkers are now available and, with the use of SCOTT for bootstrap and debugging, we have an almost complete alternative toolset. The only real remaining work is the porting of suitable C and FORTRAN run-time libraries.

Finally, we should look at the choice of implementation language. While `tcl` has attracted a considerable following, it does have some performance limitations. The start-up of `tgdb` is, for example, very slow. An obvious possibility is to rewrite some critical sections in C; a more exciting alternative is to attempt to use the emerging Java language as an alternative to `tcl` for tools integration.

10. References

[1] A prototype can be obtained by anonymous FTP from ftp.botik.yaroslavl.su in directory /pub/local/gcc-t800.

[2] M Debbage, M Hill, D A Nicole and S M Wykes, *Southampton's portable occam compiler (SPOC),* in WOTUG-17, Proceedings of the 17th world occam and transputer user group technical meeting (1994).

[3] John K Ousterhout, *Tcl and the Tk Toolkit*, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley (1993).

[4] T Brent Welch, *Practical programming in Tcl and Tk*, Xerox PARC, Palo Alto, California (1995).

11. Figures

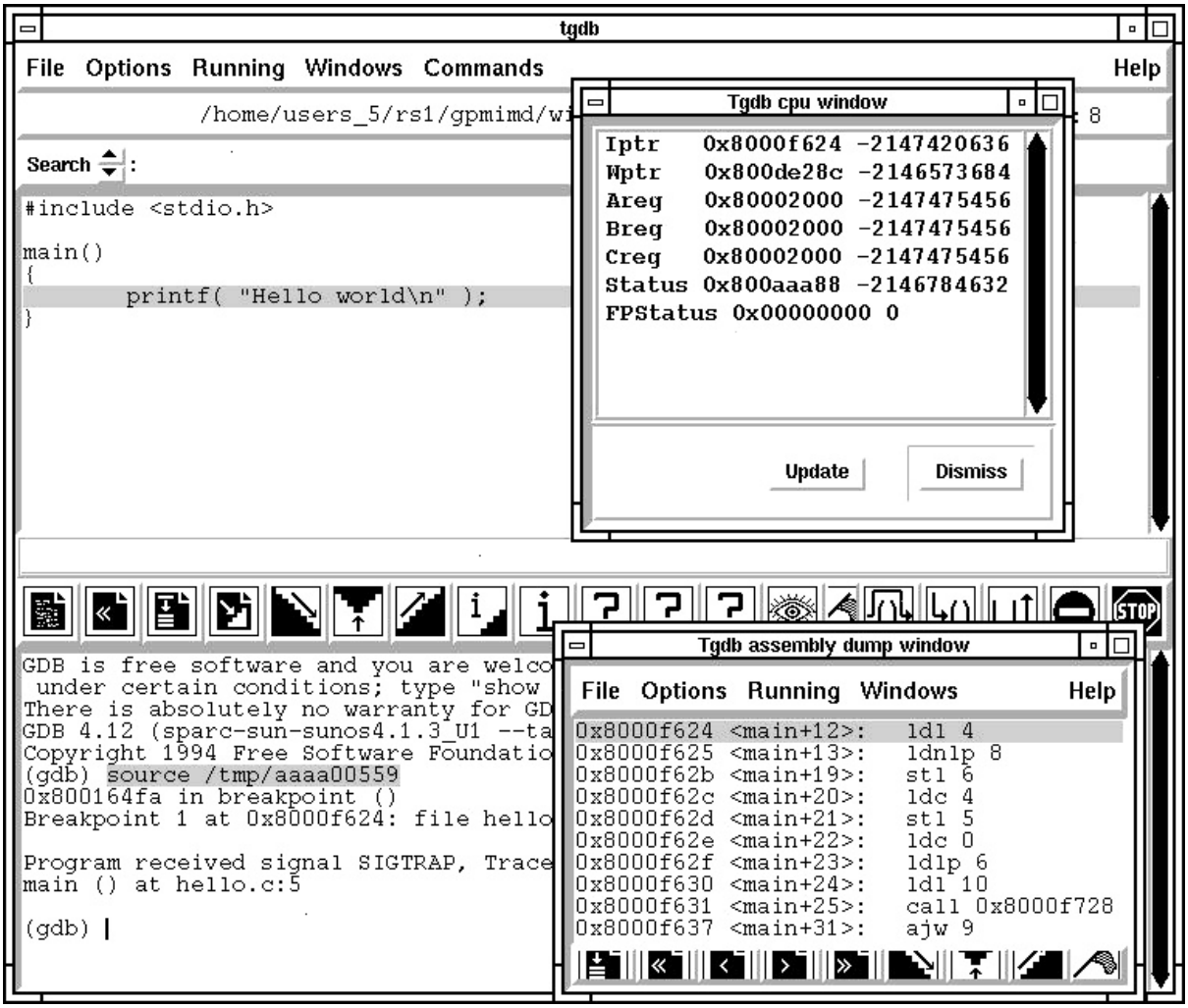

Figure 1. The tgdb graphical front end in action.

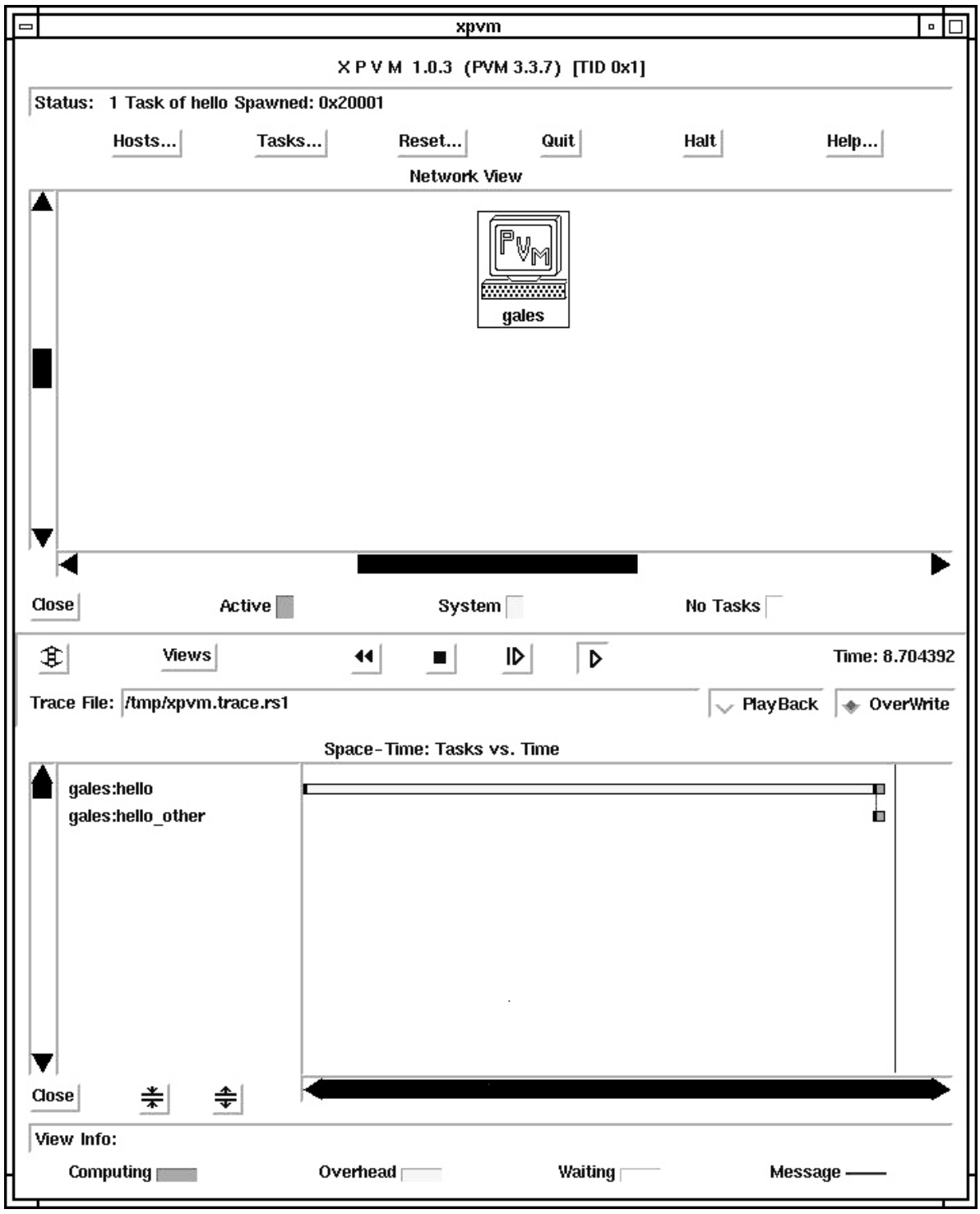

Figure 2. The xpvm console running a job on two T9000s.