The Various Ports

Rob Pike

This document collects comments about the various architectures supported by Plan 9. The system tries to hide most of the differences between machines, so the machines as seen by a Plan 9 user look different from how they are perceived through native software. Also, because we are a small group, we couldn't do everything: exploit every optimization, support every model, drive every device. This document records what we *have* done. The first section discusses the compiler—actually the compiler/assembler/loader suite—for each machine. The second talks about the operating system as implemented on each of the various machines.

The Motorola MC68020 compiler

This is actually the oldest compiler of the bunch. Relative to its competitors—commercial compilers for the same machine—it generates quite good code. It assumes at least a 68020 architecture: some of the addressing modes it generates are not on the 68000 or 68010.

We also use this compiler for the 68040. Except for a few instructions and registers available only from assembly language, the only user-visible difference between these machines is in floating point. Our 68020s all have 68881 or 68882 floating point units attached, so to execute floating point programs we depend on there being appropriate hardware. Unfortunately, the 68040 is not quite so thorough in its implementation of the IEEE 754 standard or in its provision of built-in instructions for the transcendental functions. The latter was easy to get around: we don't use them on the 68020 either, but we do have a library, `-l68881`, that you can use if you need the performance (which can be substantial: `astro` runs twice as fast). We don't use this library by default because we want to run the same binaries on both machines and don't want to emulate `FCOSH` in the operating system.

The problem with IEEE is nastier. We didn't really want to deal with gradual underflow and all that, especially since we had half a dozen machines we'd need to do it on, so on the 68040 we implement non-trapping underflow as truncation to zero and do nothing about denormalized numbers and not-a-numbers. This means the 68020 and the 68040 are not precisely compatible. Our floating-point experts side with us on this one, so don't try to talk us out of it.

The MIPS compiler

This compiler generates code for the R2000 and R3000 machines configured to be big-endians. There is no support for little-endian machines. Considering its speed, the Plan 9 compiler generates good code, but the commercial MIPS compiler with all the stops pulled out consistently beats it by 20% or so, sometimes more. Since ours compiles about 10 times faster and we spend most of our time compiling anyway, we decided the tradeoff was fine.

The compiler is solid: we've used it for several big projects and, of course, all our applications run under it. The behavior of floating-point programs is much like on the 68040: the operating system emulates where necessary to get past non-trapping underflow and overflow, but does not handle gradual underflow or denormalized numbers or not-a-numbers. (MIPS themselves are rumored to have 50,000 lines of assembler to reconstruct the state of the machine and provide this support. Good for them.)

The SPARC compiler

The SPARC compiler is also solid and fast. We have seen it do much better than GCC with all the optimizations, but on average it's probably about the same.

We have no SPARC machines with multiply or divide instructions, so the compiler does not by default produce them. Instead it calls internal subroutines. A loader flag, `-M`, causes the instructions to be emitted. The operating system has trap code to emulate them if necessary, and it works if we read the book right (again, we have no hardware with which to be sure).

The floating point story is exactly the same as on the MIPS.

The Intel i386 compiler

This is really a 386 compiler, not a 286 compiler. It works only if the machine is in 32-bit protected mode. It ignores a lot of the stuff in the instruction set, being intended only for us to get our software running on the particular 386 PC's we have. Given that caveat, it's fairly solid and generates tolerable code, although on both counts its probably weaker than the compilers mentioned above.

Floating point is well-behaved, but we assume you have a 387 to execute the instructions. If you do, it does the full IEEE 754 job, just like the MC68881. By default, we don't use the 387 built-ins for transcendentals; use `-l387` if you want to.

The Intel i960 compiler

This compiler was built as a weekend hack to let us get the Cyclone boards running. It's only been used to run one program—the on-board code in the Cyclone—and is therefore likely to be buggy. There are a number of obvious optimizations to the code that have never been attempted: we got the Cyclones working and moved on.

The AT&T Hobbit compiler

You probably don't have a Hobbit, but we're including the compiler because the chip may grow to have a wider audience, because we wanted to include the version of Plan 9 that runs on this machine, and for boosterism. Of the compilers we've really used (that is, except for the i960), this one generates the poorest code. There are many improvements to be made but we just haven't taken the time. Most glaringly, the Hobbit compiler does not support bitfields.

Like the MIPS, Hobbit has a configuration-time bit to be run either big-endian or little-endian; the compiler produces code for the little-endian case only.

The architecture has floating point defined, and the compiler generates floating point code, but the silicon doesn't implement it. The operating system has emulation code that does IEEE 754 without gradual underflow, etc. etc.

The Gnot operating system

The Gnot is a locally-built machine, similar in power and design to a SUN-3, with a two-bit-per-pixel display and no Ethernet. Instead, it uses an AT&T internal twisted-pair network called Incon. The Gnot has no I/O bus, but an external port on the back lets simple I/O boards talk directly to the CPU's memory bus. We used this for a handful of applications: local cache disk, printers, etc. Most Gnots have no external peripherals.

If you have SUN-3's lying around, you might consider taking the Gnot port (actually it's not a port, it's the original) as the basis for getting Plan 9 running on those machines. You'll need to learn about the SUN-3 MMU, but a combination of the Gnot system and the Sparcstation system will get you quite far. If you want someone else to do the work for you, Sydney University got an older version of Plan 9 running on SUN-3s and have volunteered to be contacted on the subject. Mail

A/Prof. R Kummerfeld
Computer Science
Madsen Bldg F09
Sydney University

Sydney NSW 2006
Australia

for more information.

The NeXTstation operating system

Plan 9 runs on the 68040-based NeXTstation and on no other NeXT machine. This is actually one of our favorite ports: the crisp two-bit-per-pixel display is a delight. The one strange part is that we need a three-button mouse, so we unplug the NeXT mouse and use a 9600-baud serial-port Logitech mouse on serial port B.

To get the system running, you need to install NeXT's `boot` program as in the TFTP directory (called `/lib/tftpd` on Plan 9, if you're booting from there) and `/68020/9nextstation` where BOOTP can find it (we of course put it in `/68020/9nextstation`). Boot that file from the network by typing (say) `ben /68020/9nextstation` to the boot ROM. This will load the `boot` program and use that to load the real operating system. You can set this up to happen automatically; see the NeXT manuals for details.

If you have the old 68030-based cubes on remainder, this system could probably be made to run on them without too much work. Like most manufacturers, NeXT kept the architecture pretty similar between models. The 68030 has a very different MMU, though, so the job won't be trivial.

The MIPS Magnum operating system

This is another favorite: although the tube is nothing like as nice as on the NeXTstation, the zippy MIPS Magnum is all-round a very comfortable place to run Plan 9.

The file you need to access with BOOTP is `/mips/9magnum`. Set the `console` ROM environment variable to `m` and `bootmode` to `e`. Then type `bootp()/mips/9magnum` (or whatever) to the boot ROM to run the system. If you want to run the machine as a CPU server rather than a terminal, set `console` to `1` instead and boot `/mips/9magnumcpu`. We also have a disk-resident boot program for this machine; see `booting`(8) for information.

We have Magnum 3000's and the system will run on only that model. We use 1-bit frame buffers and 8-bit color frame buffers; both run fine from the same binary.

The SGI Power Series operating system

We use SGI Power Series machines as our CPU servers. We don't use them as workstations, so you'll find no code in our system to support SGI frame buffers or the Geometry Engines. The machines we have are 4D series with IO3 boards. If you have older machines with IO2 boards, the code should work on them, too.

You can get started by booting `/mips/9power`, but you'll want to install the program `/mips/9powerboot` as `b` on your boot disk to run the system for real. See the manual page *booting*(8) for more information.

The Sparcstation operating system

We have SUN SLC's and a straight Sparcstation-2; the same binary works on both. It does some auto-configuration at boot time. It does, however, make a couple of assumptions: that there is no external memory and that there is only one frame buffer. The frame buffer it uses is the one in the lowest-numbered SBUS slot; this is also how the ROM selects the main frame buffer. Neither of these assumptions is deep-rooted, but we have very scanty documentation for the machines (SUN was not especially helpful) so we didn't want to try anything risky.

We believe the binary should work on the following machines: Sparcstation 1 (4/60), IPC (4/40), 1+ (4/65), SLC (4/20), Sparcstation 2 (4/75), ELC (4/25), and IPX (4/50), but, as noted, we've only tried two of these. The only frame buffers we know work are the `bwtwo` and `cgsix`, but we think the `cgthree` will work since it is supposed to be the same as the `cgsix` without acceleration hardware, which we don't support anyway (again, no documentation). Another peculiarity is that, to use the frame buffer, you must

set up the ROM's environment variable `output-device` to be `screen` so the ROM will enable the video controller; we don't know how.

SUN's ROMS don't use BOOTP to download. Instead they use RARP to translate their Ethernet address to their IP address. However, the Plan 9 kernel, once loaded, does use BOOTP to discover its IP address. If you're booting from disk, this isn't as much a problem as if you're booting from the network. If the latter, you need to install `/sparc/9ss` as the boot file for the system where TFTP can find it. If you're booting from a Plan 9 system, just make that the value of the `bootf` attribute in the network database (see *ndb*(8)). To run the machine as a CPU server, boot `/sparc/9sscpu`.

As you might guess, this is not our favorite port. But we do believe it's as solid as the rest and we do use it for production work. If you have Sparcstations and Magnums, race them.

The IBM PC operating system

The PC version of Plan 9 boots from DOS so DOS can set up things we don't know about, such as power management. The system should work on any 386 or 486 PC and can handle anything that is within the PC model, but you may want to tweak the system to handle peculiarities of your hardware. We run it on AT&T Safari, AT&T 6386, and Gateway 486 machines.

We use the VGA to support a 648×480 or 800×600 black and white display. Color displays work but only generate monochrome.

To boot the system, copy the files `/386/b.com` and `/386/9pc` to a DOS floppy (see *dossrv*(4)). Then run `b.com` on the PC to bootstrap the system. It will ask what device to boot from. Type `fd!0!9pc` or `fd!1!9pc` depending on the floppy unit number. When the system boots, it will ask where to get its files. At this point, you'll either need an Ethernet and a Plan 9 system (type `il`) or you'll need to get useful files loaded on the PC's disk.

If you want to run a PC stand-alone, follow the instructions in the document *Configuring a PC.*

The Hobbit operating system

This system runs only on a home-built board, so it's probably not of much interest.

The file server

The file server runs on only a handful of distinct machines. It's a stand-alone program, distantly related to the CPU server code, that runs no user code: all it does is serve files on network connections. It supports SCSI disks only, which can be interleaved for faster throughput. See *fsconfig*(8) for an explanation of how to configure a file server.

To boot a file server, follow the directions for booting a CPU server using the file name `9`*machtype* where *machtype* is `power`, `magnum`, `ss`, etc. as appropriate.

The SGI file server

The system runs on 4D multiprocessors with IO2 or IO3 boards, just like the CPU server system. It does not support the internal SCSI bus contollers (neither does the CPU server code); instead it needs an an Interphase V/SCSI 4210 Jaguar dual SCSI disk controller on the VME bus. The system does support the internal LANCE Ethernet controller on the IO2 or IO3, or you can use an Interphase V/Ethernet 4207 Eagle controller on the VME.

The MIPS 6280 file server

This version of the system is pretty much a port of the SGI system, so has the same properties. We use an Eagle Ethernet controller and a Jaguar disk controller, as on the SGI. There is no support for the internal SMD drive.

The MIPS Magnum file server

The Magnum file server doesn't use the built-in frame buffer: it requires a 9600 baud terminal connected to `tty1`. Set the `console` environment variable to `1` and boot using BOOTP.

The Sparcstation file server

The system assumes the same things about the hardware as the Sparcstation CPU server code. However, the file server doesn't use any existing frame buffer: it requires a 9600 baud terminal connected to `ttya`. It is known to work only on a Sparcstation 2 (4/75).

Backup

Our main file server is unlikely to be much like yours. It's an SGI Power Series 4D multiprocessor with 128 megabytes of cache memory, 7 gigabytes of SCSI magnetic disk, and a SONY WDA-610 Writable Disk Auto Changer (WORM), which stores almost 350 gigabytes of data. The WORM is actually the prime storage; the SCSI disk is just a cache to improve performance. Early each morning the system constructs on WORM an image of the entire system as it appears that day. Our backup system is therefore just a file server that lets you look at yesterday's (or last year's) file system.

The SONY WORM is the only one we've tried and therefore the only one for which we have a driver. It shouldn't be too hard to adapt the code to a different brand of jukebox, but there are certainly assumptions about the peculiarities of the device (the things are all weird). You might also consider attaching a CD-R jukebox or even just using a single WORM drive and managing the dumps a little less automatically. This is just a long way of saying that the system as distributed has no explicit method of backup other than through the WORM jukebox.

Not everyone can invest in such expensive hardware, however. Although it wouldn't be as luxurious, it would be possible to use `mkfs`(8) to build regular file system archives and use `scuzz`(8) to stream them to a SCSI 8mm tape drive. `Mkext` could then extract them.

It's also possible to treat a regular disk, or even a part of a disk, as a fake WORM, which can then be streamed to tape when it fills. This is a bad idea for a production system but a good way to learn about the WORM software. Again, see *fsconfig*(8) for details.

Other systems

You may have little-endian MIPSes, or IBM RS6000s, or HP PAs, or DEC Alphas, or all kinds of other machines not on the above lists. Please don't ask us to do the port for you. We've had our fill of ports and, worthy though all this porting activity has been, it's dull and can easily take up all one's time. Although the applications all port trivially (they really do), the work of building and maintaining all the compilers and machine-dependent parts of the operating system and driving all those different brands of I/O devices is substantial. Adding another machine to the list is not free, it's a linear amount of work. The constant is small but *N* is getting large.