Implementation of a Distributed Full-System Simulation Framework as a Filesystem Server

Phillip Stanley-Marbell
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

Full-system simulation of systems comprising hundreds of microcontrollers, at the level of instruction execution, along with simulation of their peripherals, inter-device communication, power consumption and the like, can be tasking even on high-end workstations. To enable the partitioning of these simulations, which have a high degree of coarse-grained parallelism, over a network of workstations, a multi-platform simulation environment was implemented using the Inferno system. The implementation enables the simulation engine, written in ANSI C, and compiled as a library, to be linked against the Inferno emulator with a custom device driver interface. Using a glue application written in Limbo, and harnessing ideas from parallel discrete-event simulation, the framework enables simulations of networks of embedded systems to be partitioned across workstations of heterogeneous architectures. This paper presents the distributed simulation architecture, the design of the emulator device driver (the interface to the simulation engine), the graphical interface and glue application, and the packaging of the system as single-binary modules for multiple platforms. Also presented is a step-by-step guide for developers unfamiliar with Inferno for creating similar systems.

1. Introduction

Simulators enable the study of systems before their actual construction. In some situations, e.g., when prototype hardware exists, they are a tool of *convenience*. When developing new application platforms, in which prototype hardware is either non-existent or not fully understood, simulation becomes a *necessity*.

The level of abstraction within simulators vary. For computing systems, simulation may range from behavioral modeling (e.g., using tools like Matlab/Simulink), and architectural modeling (e.g., SPIM, the simulators bundled with gdb, or the `vi/5i/ki/qi` simulators from the Plan 9 compiler suite), to microarchitectural simulation, modeling motion of instructions through a processor's pipeline, possible out-of-order instruction dispatch and execution, all the way down to modeling the system at the level of transistors.

The simulation framework described in this paper was created as part of the development of a *macro-sensor-electro-mechanical systems (MSEMS)* hardware platform (Figure 1). MSEMS are analogous to *micro-electro-mechanical systems (MEMS)*, which combine integrated circuits and mechanical structures on the same silicon die, and to *sensor networks*, which involve multi-modal sensing on multiple computing nodes. Unlike MEMS, in which electric circuits and mechanical systems are integrated at the silicon die-level, MSEMS incorporate embedded processors, sensors and mechanical actuators over a large surface area. In contrast to sensor networks, MSEMS are *integrated* at high densities into materials, rather than being placed discretely in their environments.

To enable the study of these systems, the Sunflower *full-system simulation environment* [10] was developed. The framework models multiple instances of complete embedded systems, performing architectural and microarchitectural simulation, power estimation, battery and voltage regulator modeling. The framework enables the modeling of networks of user-defined topologies between systems, as well as time- and location-varying analog signals in their environments. The details of the simulation frame-

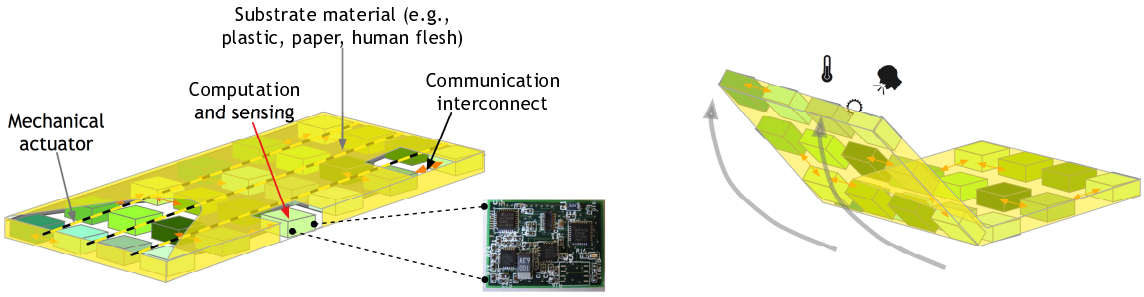


Figure 1: Illustration of one target hardware platform *computationally-animate* MSEMS, the simulation environment is used in the investigation of.

work are presented in [10]. This paper presents the implementation of facilities within the simulation framework that enable the distribution of simulations across multiple simulation hosts.

The implementation of the simulation framework was structured to enable it to be compiled as a library for the Inferno operating system [3]. A device driver was implemented within the Inferno emulator (`emu`), to present the facilities of each simulation engine as a filesystem interface. A control interface and simulation glue logic were implemented in the Limbo programming language [8], to enable connection of the simulation state hosted on different workstations flung across a network, into a single simulation system. The simulation engine's state and the device driver interface to it are described in Section 2., along with a description of the glue logic and graphical user interface which interacts with the device driver. The packaging of the simulation framework as a single binary, which combines the simulation engine, Inferno emulator and the associated filesystem, fonts, etc., is described in Section 3.. The implementation, which uses the Inferno distribution, is detailed in Section 4.. It is followed by a brief evaluation of the performance speedup enabled by the distributed simulation framework, when employing a cluster of five workstations, in Section 5.. We conclude in Section 6. with a summary and directions for future investigation.

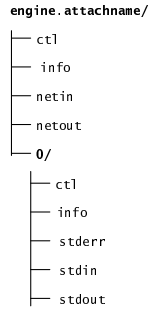
1.1. Terminology

Within the Inferno and Plan 9 operating systems, the hierarchy of entries in the perceived "filesystem", visible to programs, either through the operating system facilities, or by direct communication on the 9P [6]/Styx [7] protocols, is referred to as the *name space* (sometimes as the *namespace*). Entries in the name space, which do not represent actual blocks of data on permanent backing store (say, on disk), are usually referred to as *synthetic files*. Such synthetic entries in the name space may be synthesized by user-level programs and are often dynamic in nature. The entries in the name space can be thought of as having *type structure* in the sense of programming language type theory. The entries have a restricted set of types which distinguish between entries without internal structure (*files*) and those with internal structure (*directories*); together, these files and directory entries in the name space may collectively be referred to as *names*.

The term *device driver* in Inferno is used to refer to modules, compiled into the native operating system or emulator, providing an in-kernel (or in-emulator) interface to some resource, as a hierarchy of names. The term arises since such modules are traditionally used to provide access to hardware devices, but they are often also used to provide access to non-hardware resources; for example, in this paper, a device driver is used to provide access to a library compiled into the emulator (or into a native kernel).

1.2. Related Research

The use of the Inferno operating system emulator for constructing multi-platform applications has previously been discussed in the literature [9], however there has since been very little activity in this direction. While the use of Inferno to provide stand-alone applications for host platforms has recently seen some interest, current implementations [2], do not provide a single binary for the host platform, but rather, a customized installation archive.




```
        ID = 00000000          Active = 0
        PC = 08000000            Tcyc = 1.67E-08
    ntrans = 0.00E+00            Ecpu = 0.00E+00
      Tcpu = 0.00E+00          ninstrs = 0.00E+00
   cputype = SuperH               Vdd = 3.30E+00
  nicnifcs = 0               nicqintr = 0
   P(fail) = 0.00E+00        Max fdur = 0
  tripRate = 0               Sim Rate = 0
  Throttle = 0               R_active = 0.000000
       loc = 0.0,0.0,0.0      sensor0 = 0.00E+00
```

```
; bind -a '#*engine' /n/remote
; cd /n/remote
; du -a
0    engine.focus.local/0/ctl
0    engine.focus.local/0/info
0    engine.focus.local/0/stderr
0    engine.focus.local/0/stdin
0    engine.focus.local/0/stdout
0    engine.focus.local/0
0    engine.focus.local/ctl
0    engine.focus.local/info
0    engine.focus.local/netin
0    engine.focus.local/netout
0    engine.focus.local
0    .
;
; cd engine.focus.local
; echo newnode msp430 0 0 0 0 0 > ctl
; lc
0/   1/   ctl   info   netout   netin
;
```

(a) (b) (c)

Figure 2: (a) hierarchy of entries in the *synthetic filesystem interface* to the simulation engine device driver; (b) simulated node's informational summary returned by a read of its node-specific `ctl` interface; (c) interacting with the simulation engine device interface from the Inferno command line.

The motivation for using Inferno as the medium for implementation of multi-platform applications is usually greatest when these applications execute over a network of computing hosts, i.e., one wishes to construct what would usually be referred to as *distributed applications*. The benefit of using Inferno as an implementation medium is usually due to the ease of interconnecting multiple instances of the native or emulated Inferno system, and accessing these collections through a single name space. For example, in the case of the work described in this paper, Inferno is used to facilitate the interconnection of multiple instances of a microarchitectural simulation engine, into a single distributed simulation engine.

2. Simulation Engine Interface

In the original (console-based) implementation of the Sunflower simulation framework, users interact with the simulator through a command language. Commands issued through this interface are used to perform tasks such as instantiating embedded systems with processors from one of the two ISAs modeled, instantiating network interfaces on those systems, configuring the properties of the interfaces, and the like. Underlying the interactions with the simulation engine, is the concept of a "current" system, which receives *modal commands*. For example, a `newnode` command will always instantiate a new processor in the simulation while a `dumppipe` command will show the contents of the instruction pipeline for the *current* node.

The device driver interface to the simulation engine makes the command interface available as a small structured synthetic file hierarchy, illustrated in Figure 2.(a). At the top-level of the hierarchy are four files, `ctl`, `info`, `netin` and `netout`, and at least one numbered directory. The `ctl` file accepts writes of any commands in the simulation engine's command language; when read, it yields a single string indicating the number of instantiated processors (which could also be determined by a directory read and counting the numbered directories), as well as an indicator as to which processor is the current node. The file `info` can be read for a buffer of recent simulation informational messages (e.g., the output of a previously issued `help` command). Messages specific to individual processors are read from relevant files in the processor's numbered directory.

The entries `netin` and `netout` are interfaces to the simulation engine's MAC layer network modeling. A read of `netout` blocks until a frame is transmitted on *any* of the possible plurality of modeled communication links. Likewise, writing an appropriately formatted (as plain text) frame to `netin` injects a MAC-layer frame into the simulation engine. These interfaces are used to connect simulated networks across multiple simulation hosts, as will be detailed in Section 2.1..

The numbered "line directories" contain a similar set of files for interacting with a specific instantiated processor. Each line directory contains five files: `ctl`, `info`, `stderr`, `stdin`, and `stdout`. While modal commands issued into the top-level `ctl` interface are always with respect to the engine's concept of the "current" node, modal commands issued into the `ctl` interface of a particular node affect that node as though it were the current node. Reading the `ctl` interface yields an abbreviated summary of the node's state, shown in Figure 2.(b).

Reading the `info` interface in a line directory blocks until *node-specific simulation messages* have been printed or buffered. Node-specific information includes the response of all modal commands, as well as node-specific out-of-band messages (e.g., the register file dump printed when a node faults).

To enable the simulation of industry and academic benchmark suites in the absence of an operating system running *over* the simulator, the simulation engine intercepts software exceptions corresponding to system calls in the *Newlib* C library, enabling most POSIX-compliant applications to be compiled against Newlib and run directly over the simulator. For such situations, the eponymous interfaces `stdout` and `stderr` can be read for the respective output streams. The interface `stdin` is currently defunct, but in the future will enable interactive input to Newlib-linked applications executing over the simulator.

Figure 2.(c) illustrates interacting with the simulation engine device interface from the Inferno command line; in practice however, the graphical user interface provides an additional abstraction layer over this low-level interface.

2.1. Engine Glue Logic and User Interface

The use model of the Sunflower simulation infrastructure is for simulating large networks consisting of tens to hundreds of complete embedded systems. This is a tall order even for high-end workstations. The goal of the interface described in the previous section, was to enable a single simulation, composed of, say, one hundred simulated nodes on a single simulated network, to be split across multiple workstations (simulation hosts).

The simulation task is inherently parallel, since groups of instantiated processors in the simulation can be hosted on different simulation hosts. Unfortunately, the task is not quite so trivial: the nodes within a simulation interact via frames transmitted on one or more of the simulated communication media, and the destination of a transmitted frame within a simulation may not be on the same simulation host. The utility of implementing the interface to the simulation engine as a filesystem thus becomes evident, since filesystems across multiple Inferno hosts are easily combined into a single name space.

For simulation of a single network of systems over a cluster of simulation hosts, a simulation configuration is split into n pieces, each of which is loaded onto a simulation engine on one of n simulation hosts. The list of simulation hosts is provided to the glue logic, which serves two purposes. First, it interconnects the simulated networks of all the simulation hosts, using a set of threads to create a fully connected graph of all the simulation host `netin` and `netout` interfaces. Second, it ensures that the passage of time on the different simulation hosts is coherent. The latter is important since the simulation hosts might have different simulation rates — it is possible for the nodes being simulated on one simulation host to be "ahead" (or to lag) those on other simulation hosts, in terms of simulated (virtual) time. This is troublesome, since it breaks the notion of a single simulation time used to schedule events in the simulated system. These issues are well known in the area of parallel discrete event simulation [5]. The simulation glue employs two methods that can be activated separately or in concert by a user.

The first synchronization technique involves monitoring the simulation rates on all the simulation hosts connected to the glue interface, and issuing commands to their `ctl` interfaces to *throttle* the simulation speeds of hosts that are too far ahead in time. The glue interface determines the simulation rates by reading the per-node `ctl` interfaces described in the previous section and in Figure 2.(b), calculating an appropriate throttle factor, and issuing a `throttle` command to the top-level interface on the appropriate simulation hosts. The frequency with which this rate-synchronization occurs is specified through the GUI.

The second synchronization technique accounts for skew in the simulated times, separate from any

difference in the simulation rates. The implementation is similar, periodically calculating an appropriate *pause* to be issued to leading hosts, and issuing the command through the same interface. In practice, the skew synchronization is run at a very low period, e.g., once per five wall-clock minutes of simulation, whereas the rate synchronization re-calculates the throttle factors every minute.

It is necessary to continually monitor simulation rates and skews since, with variation of the simulated applications behavior, the simulation rate may fluctuate over time. For example, in phases of a simulated application binary where it engages in more MAC-layer communication, the simulation engine does more work to perform the modeling of these MAC-layer communications, and simulation proceeds more slowly.

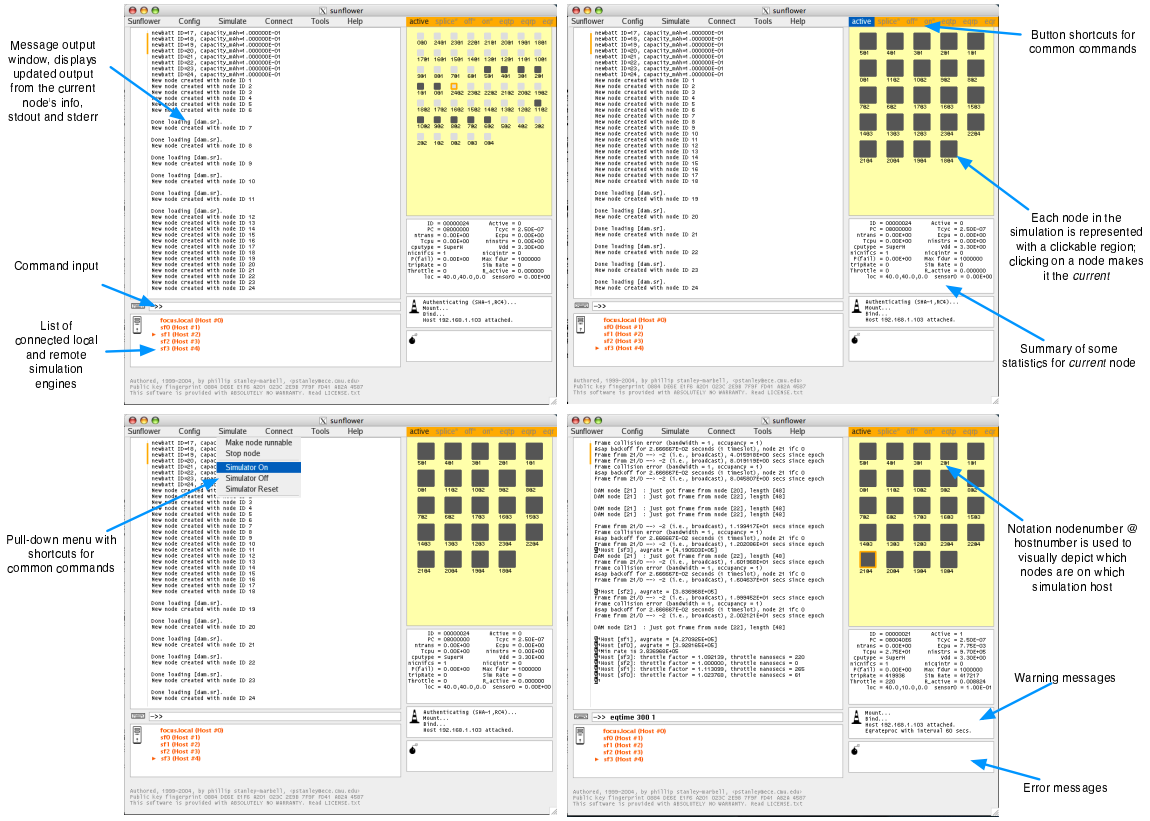


Figure 3: Illustration of the GUI.

2.2. The Graphical User Interface (GUI)

The graphical user interface was implemented to tie together facilities for connecting to remote simulation hosts, interfacing with the simulation glue logic, and interacting with a local simulation engine. The GUI was implemented in Limbo, using a combination of the Limbo/Tk toolkit and a graphics library developed in-house. A screen capture of the GUI, interfaced to a simulation with twenty-four simulated systems, split over five simulation hosts, is shown in Figure 3.

Four classes of commands can be entered at the GUI's command line. The GUI implements a small set of commands, mostly for configuring the glue logic facilities. Any command not in this set is assumed by the GUI to be a command from the simulation engine's command language, and is passed to the appropriate `ctl` interface in the simulation engine filesystem. The third class of commands are any commands preceded with a '`!`'; these are interpreted as names of Limbo programs, and are executed using the Inferno shell. Using the Inferno `os` utility, a fourth set of commands, i.e. any command from the host operating system, can also be entered at the GUI interface. Thus, for example, "`ps`" is a modal command in the simulator command language that lists the power consumption statistics of the currently selected node, "`!ps`" can be used to list the Dis VM threads running within the GUI, and

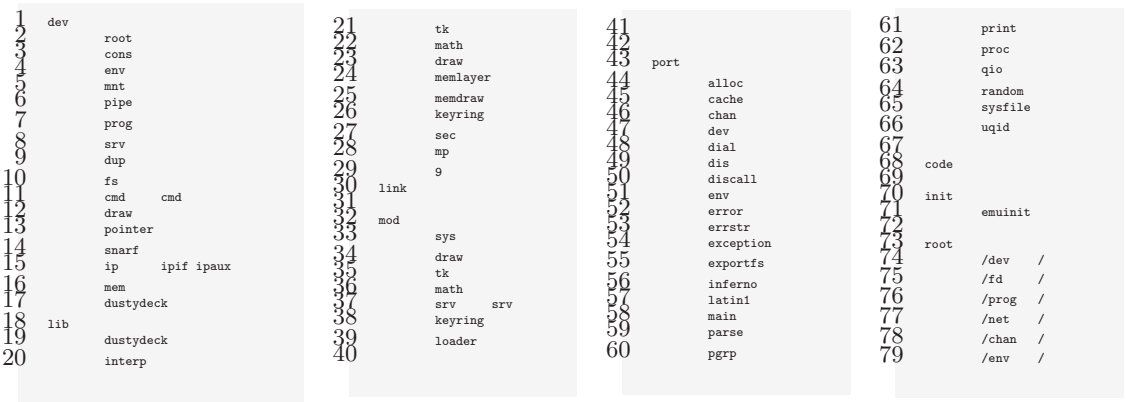


Figure 4: A skeletal emulator configuration file, with entries added to cause the incorporation of the `dustydeck` library and device driver.

"`!os ps`" can be used to list host operating system processes. The outputs of all of these commands are displayed in the GUI's message window.

3. Multi-Platform Packaging

In making the simulation framework available across multiple platforms, the goal was to distribute a single executable that users would execute just like a standard application binary on the target platform, without having to install the Inferno distribution (whether manually or automatically initiated). Implementing the interface to the simulation engine using Inferno had the significant benefit of shielding the implementation efforts from the details of multiple host platforms and their tools. For example, without any prior background in the details of the Windows API, it is possible to distribute a stand-alone windows application (a single "`.exe`" file that can be executed to launch the simulator), that behaves identically to releases on other platforms, e.g., MacOS, Linux and Solaris. These binaries are essentially customized versions of the Inferno emulator, `emu`, with the simulation engine compiled in as a device driver, and with the Limbo application executing at startup being the simulation glue logic and GUI. The GUI was implemented such that it managed its own windows, and could thus run without the Inferno window manager, taking up the whole screen real-estate (as can be see in Figure 3).

4. Implementation

The following provides an overview of the general implementation technique; the description is intended to be sufficiently detailed to enable a developer unfamiliar with the Inferno environment to implement a framework using a similar approach, with a little background reading.

4.1. Obtaining the Inferno source distribution

The Inferno distribution, comprising the sources and pre-compiled binaries for the Inferno emulator, the native Inferno operating system, the Dis virtual machine, the Limbo compiler, Inferno applications and development tools and host OS tools, can be obtained from `http://www.vitanuova.com`. The documents therein provide sufficient information to unpack and install the distribution.

4.2. Preparation

The general basis of the approach is to take an existing application implemented in C, and to compile it as a library, to be linked against the Inferno emulator. Typically, this implementation is placed in a directory with the `lib` prefix (not just a convention, but required by the structure of the emulator build configuration mechanisms), at the top level of the Inferno distribution. There is no need to tailor this library implementation in any way, other than possible changes to identifiers required to remove

```
<../mkconfig

LIB=libdustydeck.a

OFILES=\
        main.$O\
        dustydeck.$O\

HFILES=\
        dustydeck.h\

<$ROOT/mkfiles/mksyslib-$SHELLTYPE

CFLAGS = $CFLAGS -I. -c
```

Figure 5: A skeletal library `mkfile`.

conflicts with symbols in other object files in the emulator implementation. The linking of the library against the emulator is specified by adding an entry to the *emulator configuration file*, `emu`, a plain-text file located at `/emu/$PLATFORM/emu` (Figure 4), where `$PLATFORM` is a platform string such as `Plan9` or `MacOSX`. In doing so, the entry added is the tail of the name the library directory (excluding the required `lib` prefix described above). A template *mkfile* that can be used as a basis for the mkfile in the new library's directory is illustrated in Figure 5.

The interface between the Inferno system and the body of C code is provided by implementing a very simple device driver (easily under a hundred lines of C code). It is through interaction with the interfaces presented by this simple device driver that functions in the compiled library are accessed. One design challenge is to determine an appropriate interface (single entry in the name space, multiple entries, a hierarchy, or a combination of all the above in a dynamic hierarchy). A skeletal implementation of a device driver is illustrated in Figure 6.

The source for the device driver is by convention placed in the `/emu/$PLATFORM/` directory if it is specific to a given host operating system, or in the `/emu/port/` directory otherwise. The device driver is typically implemented in a single file (auxillary routines may exist in libraries), and this file name must begin with the prefix `dev`, due to the structuring of the emulator build configuration file; an entry must be added to the build config file, in its `dev` dection, for the tail of the name (i.e., excluding the `dev` prefix).

4.3. A step-by-step guide

Assuming an instance of the Inferno distribution is located in `/usr/inferno`, the steps involved in taking an existing body of C code and compiling it as a device driver named `dustydeck` are as follows:

1. Make sure that the Inferno tools for your host platform are accessible in your path. For example, on the MacOS platform, these tools will reside in `/usr/inferno/MacOSX/power/bin`, so that directory should be added to your path.

2. Create the directory `libdustydeck` in the root of the Inferno source tree. In this directory, place the source files for the C application being ported.

3. Edit the mkfile for the host platform emu build, e.g., `/usr/inferno/emu/MacOSX/mkfile` on the Mac OS X platform, if necessary.

4. Create a `mkfile` in `/usr/inferno/libdustydeck`, using, e.g., the example in Figure 5 as a template. If all is well, you should be able to type `mk` in the directory `/usr/inferno/libdustydeck`, resulting in the C code being compiled, and a library archive being placed at `/usr/inferno/MacOSX/power/lib/libdustydeck.a`.

5. Create the skeletal device driver interface, `devdustydeck.c`, (illustrated in Figure 6) in the host platform emu portable source file directory, `/usr/inferno/emu/port`.

```
#include        "dat.h"
#include        "fns.h"
#include        "error.h"
#include        <interp.h>
#include        "image.h"
#include        <memimage.h>
#include        <memlayer.h>
#include        <cursor.h>

enum    {Qdir, Qdustydeck};
static  Dirtab dustydeckdirtab[]={"dustydeck", {Qdustydeck, 0}, 0, 0600};

static Chan*
dustydeckattach(char* spec) {
        return devattach('+', spec);
}

static int
dustydeckwalk(Chan* c, char* name) {
        return devwalk(c, name, dustydeckdirtab, nelem(dustydeckdirtab), devgen);
}

static void dustydeckstat(Chan* c, char* db) {
        devstat(c, db, dustydeckdirtab, nelem(dustydeckdirtab), devgen);
}

static Chan*
dustydeckopen(Chan* c, int omode){
        return devopen(c, omode, dustydeckdirtab, nelem(dustydeckdirtab), devgen);
}

static void
dustydeckclose(Chan* c) {
        return;
}

static long
dustydeckread(Chan* c, void* a, long n, vlong offset) {
        return n;
}

static long
dustydeckwrite(Chan *c, void* a, long n, vlong offset) {
        USED(c); USED(a); USED(n); USED(offset); error(Ebadusefd);
        return 0;
}

Dev dustydeckdevtab = {
        '+', "dustydeck", devinit, dustydeckattach, devclone, dustydeckwalk,
        dustydeckstat, dustydeckopen, devcreate, dustydeckclose, dustydeckread,
        devbread, dustydeckwrite, devbwrite, devremove, devwstat
};
```

Figure 6: A skeletal implementation of the emulator device driver interface.

Figure 7: A five PC cluster over which the distributed simulation infrastructure was deployed.

6. Edit the emulator build configuration file (illustrated in Figure 4) for your host platform, adding two new lines (1) to indicate that the library `libdustydeck.a` should be linked into the emulator build, and (2) that the device driver interface `devdustydeck.c` should be compiled and linked in. Both entries are just the string `"dustydeck"`, in the `dev` and `lib` sections of the emulator configuration file.

7. Run `mk` in the emu build directory for your host platform, e.g., in `/usr/inferno/emu/MacOSX/`.

If all goes well, the binary `o.emu` is generated. This modified form of the emulator has the C library `dustydeck` compiled in, and accessible from above the Dis virtual machine through the device driver's interface.

4.4. Integrating the Sunflower simulation engine

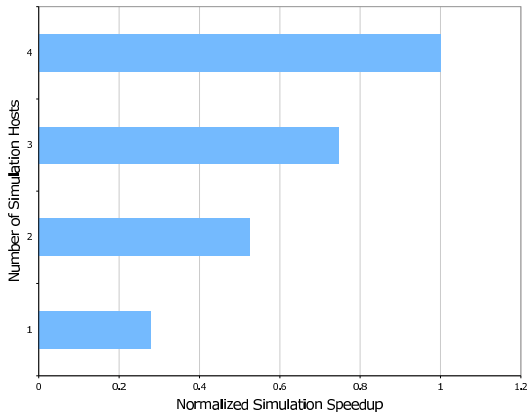
The modifications to the emulator configuration file involve adding new entries to the `dev` and `lib` sections for the simulation engine device driver and library respectively. The `root` in-memory filesystem section was also modified to include all the files needed by the GUI, along with several useful utilities, such as `ps`, `cat`, and `webgrab`; these utilities can be accessed from the simulator GUI command interface, as described previously. New functionality can be added to the GUI by including new Dis executables in the build image, or even by accessing executables over the network at runtime, providing many of the benefits of a "plug-in" architecture without the usual ugliness of plug-ins.

5. Evaluation

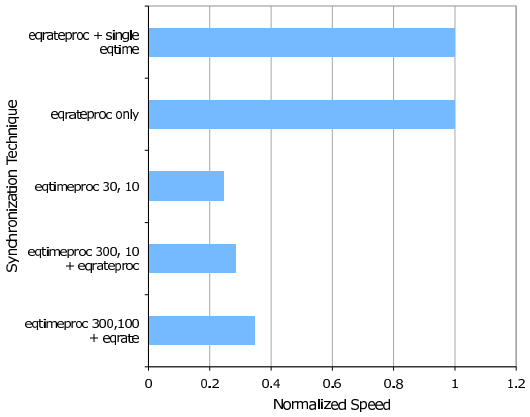
The distributed simulation framework was deployed on a cluster of five Pentium III 933MHz PCs, shown in Figure 7. The PCs have 100Mbit Ethernet interfaces, and are connected through a Netgear GS105 Gigabit Ethernet switch. A sixth PC is employed to run the GUI and glue logic interface.

When simulating networks of processors, the performance of the distributed simulation depends on the amount of communication between simulated nodes, and on the speed of the interconnect between simulation hosts. In the following evaluation, the framework is used to perform microarchitectural simulation with power estimation, battery, network and analog signal modeling, for an object tracking / sensor aggregation application [4]. With the entire simulation of twenty-four simulated nodes running on a single one of the employed host workstations, instruction execution is performed at a rate of 100K simulated clock cycles per second. On a single Pentium 4 3GHz, the simulation rate for the simulated network of twenty-four systems is 200K simulated clock cycles per host second.

Figure 8(a) shows the trend in simulation speedup when a single simulation is split across multiple hosts. In Figure 8, the speedup is shown for a setup in which the central simulation controller is actively monitoring simulation rate on the simulation hosts, but with neither synchronization of time or rate, nor communication between the simulation hosts. This represents the best case behavior of



(a) Speedup with added simulation hosts.



(b) Effect of synchronisation technique.

Figure 8: Performance of distributed simulation facilities in Sunflower. The speedup from partitioned simulation *without synchronization or communication*, but with central simulation controller polling simulation hosts for statistics is shown in (a). The effect of different synchronization techniques on simulation speed is shown in (b).

benchmark applications running over the simulator. In practice however, the communication patterns exhibited by different applications will influence the observed simulation speedup.

The rate equalization technique incurs the smallest overhead, and when combined with a one-time time synchronization, provides the best solution (`eqrateproc`, top two bars in Figure 8(b)). When using repeated time synchronization (*eqtimeproc a b* in Figure 8(b), with *a* being the periodicity in ms and *b* the maximum skew on the simulated machine in ms), performance is degraded by the repeated pausing of simulation. These evaluations, although rudimentary, are promising. An immediate area for improvement is the implementation of the time-synchronisation facilities.

6. Summary and Future Work

Full-system simulation of systems comprising hundreds of microcontrollers, at the level of instruction execution, along with simulation of their peripherals, inter-device communication, power consumption and the like, can be tasking even on high-end workstations. Such simulations however have a high degree of coarse-grained parallelism, and may be partitioned over a network of workstations.

Presented in this paper were facilities for performing such partitioning of a simulation engine, implementing a multi-platform simulation environment using Inferno. The simulation engine, written in ANSI C, is compiled as a library, and linked against the Inferno emulator, with a custom device driver serving as interface to the simulation engine. Using a glue application written in Limbo, and harnessing ideas from parallel discrete-event simulation, the implementation enables simulations of networks of embedded systems to be partitioned across workstations of heterogeneous architectures. Presented was the distributed simulation architecture, the design of the emulator device driver (the interface to the simulation engine), the graphical interface and glue application, and the packaging of the system as single-binary modules for multiple platforms. To enable developers unfamiliar with Inferno to create similar systems, a step-by-step implementation guide was also presented.

Alongside improvements in the implementation of the time-synchronisation facilities, the integration of support for the Amazon EC2 [1] pay-per-compute service is being implemented. This will enable the investigation of larger-scale simulation distributions than the small five-host cluster presented herein.

References

[1] Amazon Inc. *Amazon Elastic Compute Cloud (Amazon EC2)*. `http://aws.amazon.com/ec2`, 2006.

[2] C. Jones. *ACME Stand-Alone Complex (SAC)*. `http://code.google.com/p/acme-sac`, 2006.

[3] S. M. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey, and P. Winterbottom. The Inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, Winter 1997.

[4] Q. Fang, F. Zhao, and L. Guibas. Lightweight sensing and communication protocols for target enumeration and aggregation. In *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing*, pages 165–176. ACM Press, 2003.

[5] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[6] Plan 9 File Protocol, 9P. *Plan 9 4th Edition Programmers Manual, Section 5*. Lucent Technologies, Murray Hill, NJ, Apr. 2002.

[7] D. Ritchie and R. Pike. The Styx architecture for distributed systems. *Bell Labs Technical Journal*, 4(2):146–152, 1999.

[8] D. M. Ritchie. The Limbo programming language. In *Inferno 3rd Edition Programmer's Manual, Volume 2*. Vita Nuova Holdings Ltd., 2000.

[9] R. Sharma. Distributed application development with Inferno. In *Proceedings of the 36th Design Automation Conference (DAC' 99)*, pages 146–150, June 1999.

[10] P. Stanley-Marbell and D. Marculescu. Sunflower: Full-System, Embedded Microarchitecture Evaluation. *2nd European conference on High Performance Embedded Architectures and Computers (HiPEAC 2007) / Lecture Notes on Computer Science*, 4367:168–182, 2007.