A VIRTUAL FILESYSTEM FRAMEWORK TO SUPPORT EMBEDDED SOFTWARE DEVELOPMENT

Bhanu N. Pisupati

Submitted to the faculty of the Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in Computer Science and Cognitive Science

Indiana University

June 2007

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Doctoral Committee

Geoffrey Brown, Ph.D.
(Principal Advisor)

Steven D. Johnson, Ph.D.

Amr Sabry, Ph.D.

Kenneth Chiu, Ph.D.

May 24, 2007

Acknowledgements

First and foremost, I would like to thank my advisor Prof. Geoffrey Brown for his guidance and support right through my Ph.D. His enthusiasm and involvement in my research have been instrumental in helping me stay motivated and excited about my dissertation all along. I am grateful to Prof. Steve Johnson for initiating me into the world of hardware and embedded systems in my early years as a graduate student. Prof. Kenneth Chiu provided invaluable guidance in working and writing on the application of virtual filesystems to sensor networks, which forms an important contribution of this dissertation. I would like to thank Prof. Amr Sabry for readily consenting to sit on my defense committee, and who along with Prof. Kent Dybvig and Prof. Daniel Friedman taught me functional programming, which must be the coolest thing I learnt in grad school.

Much of my work takes immaculately built hardware for granted, for which I have Bryce Himebaugh to thank. Bryce has also been an eternal source of ideas for my research, and I have immensely benefited our conversations with regard to hardware design, realtime systems, football and beyond. I would also like to thank the Computer Science Department Systems Group, particularly Caleb Hess, Bruce Shei and Lynne Crohn for helping me keep hardware and software running smoothly in LH328.

Spending seven years in grad school is perhaps impossible without a network of friends, and I

have been greatly fortunate in that regard. The countless holidays and long summer nights spent with close friends from Computer Science, Chemistry and SPEA provide memories that will last a lifetime. The times were made truly memorable by the cultural vibrancy and innumerable vegetarian eateries that abound the stunningly scenic IU campus in good ol' Bloomington. Last and most importantly, I am greatly indebted to my family, who have always taught me by example. They have given me the freedom and support to shape my life as I saw fit, and I am eternally grateful for that.

Abstract

A VIRTUAL FILESYSTEM FRAMEWORK TO SUPPORT EMBEDDED SOFTWARE DEVELOPMENT

We present an approach to simplify the software development process for embedded systems by supporting key development tasks such as debugging, tracing and configuration. The approach is based on the use of distributed filesystem abstractions; principal building blocks within an embedded system in the form of "systems on chip" (SoC) export filesystem abstractions that are composed together up the system hierarchy, and provide familiar file based interfaces with which to interact with the entire system. The central question addressed in this thesis is as to how the workstation centric idea of a distributed filesystem can be implemented and effectively applied to facilitate various software development tasks in the embedded domain. To this end, a primary contribution of our work is the realization of distributed filesystem implementations that are compatible with resource constrained embedded architectures. We demonstrate use of the filesystems in enabling debugging and tracing in heterogeneous, multiprocessor environments, while addressing issues central to SoC based systems. The virtual filesystem model is also applied to facilitate usage, configuration and deployment in a contrasting embedded application domain in the form of distributed sensor networks, thereby demonstrating the its adaptability.

Contents

List of Figures

1

Introduction

Embedded systems find application in diverse fields including consumer electronics, automobiles, aviation, industrial systems, and communication. At the core of these systems are one or more programmable processors concurrently executing software that support system functionality. This dissertation presents an approach to debug, trace, configure and interact with software executing within multi-processor embedded systems through the use of virtual filesystem abstractions.

The underlying tenet behind the presented work is that a scalable solution for supporting debugging and tracing of software in multi-processor embedded systems can be obtained using distributed filesystem abstractions. In using this approach, individual building blocks within an embedded system export filesystem abstractions that are composed together up the system hierarchy, and provide familiar file based interfaces with which to interact with the entire system. Apart from supporting embedded software debugging and tracing, these filesystem abstractions are well-equipped to meet demands imposed by recent advances in system design, most notably the advent system-on-chip (SoC) based devices.

The central question addressed in this thesis is as to how the workstation centric idea of a distributed filesystem can be implemented and effectively applied to facilitate various software development tasks in the embedded domain. To this end, a primary contribution of our work is the realization of distributed filesystem implementations that are compatible with resource constrained embedded architectures. We demonstrate use of the filesystem architecture in enabling debugging and tracing in heterogeneous, multiprocessor environments, while addressing issues central to SoC based systems such as concurrent debugging, intellectual property (IP) concealment, and development of system level debug solutions. The filesystem model is also applied to facilitate usage, configuration and deployment in a contrasting embedded application domain in the form of distributed sensor networks, thereby demonstrating its adaptability.

1.1 Software in Embedded Systems

The abundance of embedded systems (and correspondingly their software) is reflected by the fact that among all microprocessors produced today, about 98% are used in the embedded domain [17]. This surge in their prevalence has been due to a combination of software based embedded systems replacing hardware based solutions in certain industries (in automobiles), and also in the arrival of new devices/systems (digital cameras) that have been made feasible by embedded systems. The move to implement system logic using software in processors rather than hardware is motivated by several reasons. Reprogramming software within the constituent processors in a system is easier than redesigning and fabricating hardware, which means use of software based approaches enhances system configurability. It is often simpler to implement certain sections of a system's design such as control algorithms and user interfaces in software. Software based designs are highly portable, which means progressively newer (faster) processors can be incorporated in designs while using the same software. On the other hand, hardware based solutions are preferred in some cases over processor based ones due to considerations of performance, power and operation mode (eg: analog).

Software in embedded systems is often executed among multiple heterogeneous processors, each of which is used to handle different a aspect of the system's operation [123]. A common technique is to use a RISC processor for implementing control and device housekeeping tasks, and specialized CISC/VLIW based processors such as DSPs for performing computationally intensive tasks related to multimedia, cryptography, networking etc. Multiple processors of the same kind may also be used as necessitated by the computational complexity of implemented applications. Multiple processors running at lower clock rates can offer comparable performance to a single higher speed processor, while consuming less power in the process [77]. With multiple, dissimilar processors executing within a system, heterogeneous concurrent debugging capability proves

invaluable during software development.

Debugging software within embedded systems in its most basic form involves use of features commonly used in debugging workstation based software, such as: breakpoints, register/memory analysis, single stepping etc. In addition to these 'run control' features, tracing provides an important tool to debug those embedded systems whose execution cannot be stopped using breakpoints either because they operate with realtime operational constraints, or in environments where operation of surrounding entities cannot be finely controlled [23]. A generalization of tracing is system monitoring [55], wherein various aspects of system operation can be observed during system execution. In multi-processor systems the various software development tasks described need to be enabled concurrently among the various processors.

Due to architectural, design and operational characteristics of embedded systems, standard approaches to debugging as used in the workstation domain cannot be directly applied to embedded systems. Workstation debuggers access process state through features provided by the underlying operating system. The popular *gdb* debugger on Linux for instance, uses a combination of the `ptrace` system call and the 'proc' filesystem [73, 48] to achieve run control over processes being debugged. Using a similar approach in running debuggers within embedded systems is typically infeasible, as they lack the memory resources required to run functionally rich operating systems that can facilitate debugging. Systems that are able to run a capable operating system have to additionally allow users to 'login' from remote workstations and execute debuggers locally, which places additional burden on the embedded system. The preferred approach to embedded software debugging is instead to execute debuggers on the workstation side, which perform run control of embedded software using a back-door mechanism, such as JTAG [107] or monitors. As discussed in Chapter 3, existing mechanisms have limitations when used with multi-processor SoC based embedded systems that are addressed as part of our work.

The various stages involved with the software development process for conventional computer systems - code synthesis, compilation, deployment and debug - apply to embedded systems as well. Embedded software is typically developed on workstations and 'cross compiled' for embedded platforms. Resulting software binaries are deployed within the system and debugged remotely from the workstation side. Existing techniques supporting debugging and tracing in the embedded domain are ill-equipped to meet demands imposed by recent advances in system design, most notably the advent of multi-processor, system-on-chip(SoC) based devices. A primary objective of the work presented in this dissertation is to use file abstractions to design a technique that supports various tasks central to embedded software development including debugging and tracing, while also addressing requirements unique to SoC based embedded systems.

1.2 Underlying Methodology

[CHG:model]A fundamental technique used in this dissertation is to extend the conventional notion of a file from merely being a means to access data residing on a storage device (such as a hard disk) to an entity that provides a standard, familiar interface with which to access and control resources of various kinds. The model that we use to build file based abstractions comprises of filesystems implemented by various building blocks within an embedded system, which are exported and accessed externally to interact with the system. The focus of this dissertation is the design of techniques that enable implementation of such filesystems within embedded systems, and the illustration of their use for debugging, tracing, monitoring and configuration.

To illustrate use of file abstractions in the embedded domain, we describe how they can be used to facilitate various software development tasks such as debugging, configuration and monitoring for a network processor such as the Intel IXP2850 [67]. The core responsibility of such a device is to

facilitate packet switching and content processing within networking devices such as routers.

The architecture of the IXP2850 broadly consists of a *control* and a *data* section, as shown in Figure 1.1, implemented respectively using an Intel XScale processor and 16 multi-threaded 'microengines'. Packets entering the network processor through the ingress port, flow through the *data plane* within the processor, where they are progressively processed by various microengines and directed out to the appropriate egress port. The XScale processor enables control operations such as router table updates and microengine control, through an interface exported over a separate *control plane*.

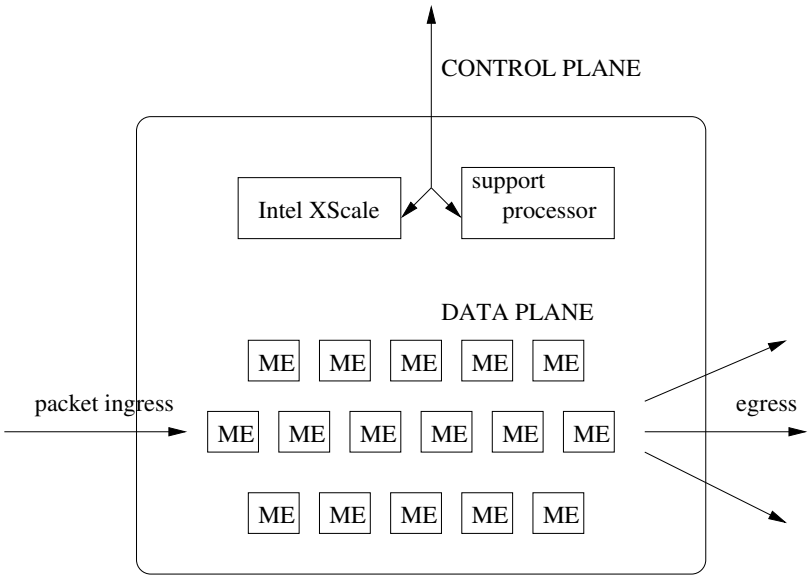

Figure 1.1: IXP2850 Network Processor Architecture

Software debugging support for such a device allows external debugging agents to access various constituent processors (XScale processor and microengines) concurrently during execution. Using filesystem abstractions, software debugging can be supported by exporting from within the device a namespace with files to enable run control of each of the constituent processors. Listing 1.1 shows one such namespace wherein each processor has a separate directory with files to access

processor memory, registers, check status and control execution.

Monitoring allows various aspects of the network processor execution to be observed during its operation. It can be used to detect trends in device operation, observe its general well-being and to keep track of relevant metrics. Examples of operational data that may be exposed through the file interface to facilitate monitoring include counts of packets discarded based on security (virus/spam) concerns, and 'data meters' representing volume of data routed to various domains, measured for purposes of billing. Configuration interfaces exposed through control files enable control of various aspects of the device operation, including routing, load balancing, and control of microengine operation. Listing 1.2 depicts a possible file based monitoring and configuration interface for the network processor.

Listing 1.1: Filesystem Namespace for NP debugging

```
\NPfs
    XScale\
       registers
       memory
       status
       ctl
   microeng1\
       registers
       .......
   microeng2\
   .......
   microeng16\
```

Listing 1.2: Filesystem namespace for monitoring & configuration

```
/NP
  counters/
       droppedPackets
  meters/
      10_1_1_x
      12_1_x
  config/
     routectl
     microengctl
```

The model used in our work implements filesystems within embedded systems using a designated processor (called 'support processor') which are exported using available communication links. Clients access the exported filesystem by mounting them within their local operating system namespaces, after which the filesystem contents are largely indistinguishable from local files. An advantage of this approach is that a single exported file interface can simultaneously support several major software development tasks.

1.3 Motivations for using Filesystem Abstractions

Filesystem abstractions enable use of uniform file interfaces to interact with the heterogeneous processing elements that coexist within embedded systems. The underlying filesystem deals with the heterogeneity within its implementation, thereby saving the end user from having to be cognizant of the differences in debug, configuration and other interfaces among processors. Apart

from offering convenience, this ability provides an effective means by which to safeguard intellectual property of various components within the system, as shown in Chapter 3. File abstractions effectively allow separation of interface from the supporting implementation. Consequently, as shown in the example in the previous section, the same model can be used to simultaneously support diverse set of software development operations. Filesystem representations make the task of naming resources within the embedded system from remote clients (such as debuggers) as trivial as resolving the path for the their respective files. Browsing resources in the embedded hierarchy becomes equivalent to listing the directory contents in the filesystem namespace. File based interfaces offer familiarity to human users and a convenient backend over which to base existing tools for debugging, programming, task automation, data analysis etc.

Embedded systems possess certain unique characteristics that make use of the filesystem abstractions natural. The short time to market and cost factors motivate embedded system designers to use third party components in building their systems [58]. With a large number of providers supplying these components, the platforms and architectures used in the embedded domain are highly fragmented, leading to a proliferation of standards. The industry has attempted to introduce harmony to the embedded domain through universal open standards relating to software debugging [20], on-chip component interaction [86, 13], and communication protocols [65, 30] among other aspects of system design. The presented model is a step in the same direction, as it uses standard, generic file interfaces exported using open communication standards, thereby promoting interoperability between systems and tools interacting with them.

Filesystem abstractions presented in this dissertation is particularly applicable to embedded systems that are hierarchical and/or distributed. As discussed in Chapter 3, hierarchy is evident in SoC based embedded systems wherein IP cores in the form of basic building blocks constitute SoC devices, that are in turn used to build circuit boards and ultimately entire systems. Distributed

hierarchies also exist in sensor networks wherein individual sensor nodes are aggregated to constitute clusters, many of which together make up a network. A key feature of the filesystem model is that by virtue of its compositional nature, software development interfaces for constituent components within a system can be naturally assembled to progressively create higher level interfaces as the system is built up.

The huge volumes in which certain embedded devices (notably consumer electronics) are produced influences designers to keep design costs to a minimum. The primary means of economizing cost is by limiting the resources used in the system. Thus processors, memories, peripherals, communication resources are all at a premium in the embedded domain. With hardware resources on a tight leash, it follows that software written for these systems need to be conservative in terms of resources consumed. A topic that is addressed in detail in this dissertation is how distributed filesystem implementations can be partitioned across different levels of the system hierarchy in order to make them compatible with the available computational and memory resources.

Popular software packages and solutions have been ported to the embedded domain by selectively limiting the offered functionality and thereby restricting the resources required in their operation. This approach has been applied to software in a variety of fields, including operating systems(ucLinux from Linux), software libraries (uclibC from libC) and networking stacks (uIP [42] from IP). A similar approach is taken in realizing filesystem abstractions in our work, whereby the implementation and operation of existing filesystems from the workstation domain is adapted to suit embedded systems.

A limitation of the request-response based filesystem model is its inability to support *push* behavior; in such operations a server initiates communication with the client on its own accord, and not as part of a response to an earlier client request. Push operations can be used in embedded systems to implement event notification in monitoring applications and during software debugging

to implement breakpoints. In this dissertation we discuss a technique to support events within the filesystem infrastructure without using push operations.

1.4 Alternate Approaches

We present an overview of other approaches in distributed systems that may be suitable for supporting software development for multi-processor embedded systems. A popular one among these is the use of object oriented representations of resources to build distributed systems. Indeed, filesystem abstractions are a special case of object oriented representations, wherein a single object type in the form of a file is used. An advantage with the use of objects is that they can be used to represent a wide array of resources. Java Remote Method Invocation (RMI) and Corba [36] are two popular middleware solutions using distributed objects. Both support the notion of remote objects through use of references, which reside locally and enable methods to be invoked on the remote node where the object resides. Java RMI extensively utilizes support for distributed objects in the Java language, as reflected by the fact that they are largely indistinguishable from local objects [124]. The presence of a uniform Java virtual machine enables code to be transferred and executed within machines across the network, unmindful of platform differences. Corba in contrast is a language independent specification for building distributed systems. Users building systems out of Corba can leverage the vast array of services provided by the underlying middleware relating to security, naming, resource discovery etc. and concentrate on building application logic. A feature of object based distributed systems (in contrast to those based on filesystems) is that they provide superior event support, based on both push and pull semantics. The use of both Java RMI and Corba is best suited for service oriented scenarios, which equips them to handle errors and makes them well suited to dynamic environments where resources continually enter and leave the system

- a trait not relevant to the class of systems being targeted in this dissertation. Their sophisticated functionality makes both Java RMI and Corba fairly heavyweight for the embedded domain, as evidenced by the amount of data transferred in completing a single request [56], which was in excess of 1KB for Corba and 7KB for Java RMI.

The World Wide Web uses a distributed document model to enable clients to access files on servers remotely using the HTTP protocol. In applying this model to the presented work, embedded system state can be accessed using various documents being served from within the system. A drawback of the HTTP protocol is that it requires the use of TCP-IP whose resource requirements cannot be met by low end embedded devices. The predominant usage mode of the web involves clients *retrieving* data such as web pages from servers; and not as much as a *writable* medium. This makes the model not directly suited for supporting operations such as debugging wherein data flows both ways. The most important operation supported by HTTP is `get` which is analogous to the `read` operation in filesystems in that it enables retrieval of file contents from the server. The `put` and `post` operations do allow clients to upload data to the server; but these capabilities are fairly basic as indicated by minimal support for synchronizing writes by multiple clients. A final reason why the web based model is unsuitable is that (like the filesystem model) it does not naturally support support push based operations that can be used by servers to notify clients of events (such as breakpoint hits).

Web services provide another approach to building distributed systems that is based on leveraging existing technologies supporting the World Wide Web, most notably the HTTP protocol and text based information exchange through XML. Use of text based data transfer through XML files allow web services to easily deal with issues like processor endianness and achieve platform independence. As with the distributed document model, use of HTTP imposes the requirement of using TCP-IP, which is not always feasible with embedded systems. XML based encodings are bulky,

leading to increased data being communicated between client and server [56]. This is not desirable in power starved embedded devices, where communication is energy consuming. Also parsing XML documents is computationally more challenging that decoding 'linear' packet formats. However, web service interfaces can be developed for embedded devices [88] by using proxies hosted on workstations that act as service providers on behalf of the embedded system; proxies communicate with the embedded system using a compatible mechanism, which could possibly be based on a filesystem interface.

There have been numerous efforts to create middleware for managing networks of distributed devices. Jini [69] is a Java based runtime environment for network of components, devices and services. It is particularly applicable to a autonomous, unfamiliar environments since it is well equipped with discovery and resource registration features. It is also highly adaptable and hence suited to dynamic environments under continuous change caused by entities joining and leaving its space. It is service oriented as opposed to being device oriented and hence shows greater resiliency under errors. However these features come at a price in terms of complexity. The CLDC machine, which is the Java virtual machine configuration with J2ME for small devices requires 128k to 512k memory for operation.

Universal Plug and Play(UPnP) [117] presents a similar functionality as Jini but in contrasting ways. UPnP is primarily used as a means to setup proximity networks for devices in a home or small office environment. UPnP uses a device centric approach since its interface allow direct access to individual devices of the network. UPnP networks use messaging protocols for communication between entities within and outside the network. Jini uses function calls to remote objects using Java Remote Method Invocation (RMI) for enabling interactions between network entities.

1.5 Contributions & Overview of Dissertation

[CHG:summarypub]The main contribution of this dissertation is the implementation of distributed filesystem abstractions compatible with resource constrained embedded architectures, and their use in facilitating various software development tasks. We implemented building blocks necessary to implement a two-level distributed filesystem hierarchy residing at the component and the system levels of an embedded system. Use of these building blocks in assembling filesystems to support concurrent, heterogeneous debugging [99] and tracing in multi-processor environments has been demonstrated. We have also applied our filesystem technology [113, 100] to enable data centric and event based applications in the sensor network domain, in addition to sensor application configuration and binary deployment within sensor nodes. The third application that we present is the use of file abstractions to import console, storage and software resources residing on workstations within resouce impoverished embedded devices; interestingly in this application the roles of clients and servers are reversed between the embedded and workstations sides as compared to the previous two cases.

Chapter 2 describes our work relating to implementation of technological building blocks needed to realize filesystem abstractions. Chapter 3 presents the core application of our work, which uses our implemented filesystem technology to support debugging in multi-processor embedded systems. Use of the approach in a contrasting embedded application domain in the form of sensors networks is described in Chapter 4. Chapter 5 presents a technique for importing workstation based resources into embedded devices using virtual filesystems. Finally we conclude with a summary and ideas for future work.

1.6 Changes

- Steve: justifying static nature of EFS namespaces [CHG:EFS] 2.4
- Steve: implications of using single threaded EFS implementation based on multiple stackless pseudo threads [CHG:singlthrd] 2.4
- Motivation for clone filesystems [CHG:clonefs] 2.4
- Steve: recognizing the role of verification in SW development [CHG:verification] 3.8
- Steve: motivations and implications of using simulators to generate instruction traces [CHG:armul] 3.7
- Steve: elaborating benefits of sensor macro programming [CHG:sensormacro] 4.6
- Geoff: refine description of 9P [CHG:9p] 2.3
- Geoff: emphasise the core pieces in a 9P filesystem [CHG:9pfs] 2.4
- Geoff: describe use of the EFS block to create filesystems [CHG:efs-use] 2.4
- Amr: explain how use of file abstractions for hardware in this dissertation differs from that in Unix [CHG:unix-compson] 2.1
- Amr : state early on what I mean by 'our model' [CHG:model] 1.2
- Amr: consequences of limitations in mount filesystem design [CHG:mfs] 2.4
- Amr: remove information about EFS and MFS design in the Embedded Debugging Chapter repeated from the Technology chapter [CHG:embed-arch] 3.5
- Amr : summary of contributions in the introduction and a description how my publications fit in with the dissertation [CHG:summarypub] 1.5

- clarify the dual occurence of sensor directories within cluster and group directories as being equivalent to hard links [CHG:sensor-links] 4.5

2

Underlying Technology

2.1 Introduction

In this chapter we discuss conceptual and implementation aspects of distributed and synthetic filesystems, that form the core technology used in this disseration. Non-traditional use of these filesystems as a standard means to access and control diverse set of resources is introduced. We describe the technique used for realizing filesystem abstractions that is based on ideas espoused by the Plan 9 operating system [97]. we present core technologies implemented as part of our work to enable adoption of the Plan 9 model within the embedded domain. We also discuss alternative options available for building filesystem abstractions and provide the rationale for our choice of the Plan 9 model.

A filesystem provides a means by which to store and present data through a file based interface. The presented data is consumed by filesystem clients through standard file operations. The entity implementing the filesystem using a combination of software and hardware is referred to as the file-server. In the simplest case, filesystems are implemented and accessed within the same computer node, often through the operating system. In other cases however, the clients and servers are separated (either physically or logically) and therefore cannot interact through the overlying operating system . Distributed filesystems bridge this gap by enabling clients to access remote filesystems as though they were implemented locally. While data associated with a filesystem typically resides on a storage device such as hard disk or flash memory, synthetic filesystems deviate from this norm by generating data to process file requests on the fly. This ability makes synthetic filesystems well suited for implementing file abstractions for non standard data sources such as hardware. They leverage the familiar, well understood notion of file interfaces to facilitate interaction with resources of diverse kinds. Using a combination of the two described filesystem models, our work uses synthetic filesystems residing within embedded systems that are exported over communication links for use by distributed clients. The filesystems expose suitable file based abstractions

for various in-system resources necessary to facilitate client debugging, tracing and monitoring of software executing within the embedded system.

Implementations of synthetic, distributed filesystem models in our work draw from Plan 9, whose two core features included uniform treatment of hardware, software and computational resources as files and easy mechanisms to export (and correspondingly import) contents of the operating system namespace. When combined, the two ideas provide a powerful framework with which to build distributed systems. Filesystems abstracting various resources and exported over communication links form the basic building blocks using which distributed systems are built in the Plan 9 model. These 'building block' filesystems within an embedded system are progressively composed to create higher level filesystems at the system and workstation level. The hierarchy of filesystems provide consistent, familiar file based interfaces with which to debug, trace and configure software executing at various levels within an embedded device. The fundamental question we address is as to how the workstation-centric idea of a distributed filesystem can be implemented within resource constrained embedded devices and used to support various software development operations.

[CHG:unix-compson]File abstractions have been used before to control and access hardware, most notably in Unix [5, 48]. Unlike in Unix, which requires use of non-conventional *ioctl* file operations to achieve device control in certain cases, files abstractions in our work provide a fully sufficient means to interact with hardware. Filesystem models implemented in our work may readily be exported over communication links, which allows clients to access and control hardware remotely though file operations. Support for exporting filesystems to enable remote access is not readily built into Unix; distributed filesystem tools such as NFS that do provide remote access capability for Unix-based filesystems need to accomodate the specialities associated with device files in order to support remote device access.

The organization of this chapter is as follows. we first present the various alternatives for implementing filesystem abstractions and present the rationale for our choice of the Plan 9 model. we then elaborate on the Plan 9 model and describe the necessary extensions to its existing implementations in order to make it suitable to the embedded domain. Finally, the implementations of these extensions are described which serve as building blocks that enable applications described in Chapter 3 and 4.

2.2 Alternatives for File System Implementation

The Plan 9 model is one of several options available for building filesystem abstractions. we discuss the requirements imposed by our work on the adopted filesystem technology, using which we present the rationale for the choice of the Plan 9 approach among various alternatives.

The requirements on the filesystem technology are four-fold. First, the technology needs to support implementation of synthetic filesystems, which use non-standard data sources such as in-system hardware for generating file data on the fly. The data generation process of associated devices often has unique requirements that the filesystem has to account for. For instance, data produced by sensors is typically valid only for a limited duration which makes it unsuitable for being cached within the filesystem. As the second requirement, the filesystems should be capable of being implemented in user space rather than within the kernel. User space code is easier to program and debug, more portable across operating systems, and is less prone to affecting system stability due to faulty implementation than erroneous kernel-resident code. Third, since the filesystems in our work are implemented and accessed in a distributed manner, the underlying technology should enable remote access by clients and other filesystems. Finally, the filesystems should be suitable for being implemented in embedded devices with limited resources. Filesystem memory requirements

should be commensurate with the available memory resources, which in some devices is as little as a few 10s of KB of flash(code memory) and less than 10KB of RAM(runtime memory).

A direct approach to implementing file abstractions for hardware is through dedicated operating system resident filesystems providing the abstraction. These filesystems can be mounted at required locations within the operating system namespace and be used to control and access hardware through conventional file operations. Many popular operating systems alleviate the difficulty in developing filesystems by defining standard, generic interfaces that filesystems can plug themselves into. Examples include VFS [102] in Linux, SunOS VFS [74] and Installable File System [93] in Microsoft Windows. While filesystems have to comply with these interface standards set by the operating system, the implementation logic is largely the filesystems' prerogative. Thus synthetic filesystem behavior could be incorporated by completing various file operations through necessary interactions with the hardware being abstracted. A more restrictive means of implementing kernel-based file abstractions for hardware in Unix-like operating systems is through devfs [5], which enables association of a single file in a global devfs directory (typically /dev) with a hardware device. Handlers for the various file operations are defined in the hardware's device driver and registered with devfs.

As mentioned in the discussion on filesystem technology requirements, kernel based filesystems are not well suited for our work. Since the interfaces between operating system and filesystem interfaces not uniform, the resulting filesystem implementations are not readily portable across operating systems. Kernel software development is a specialized domain where bugs can jeopardize stability of the entire system and are hard to isolate and correct since conventional debugging techniques such as source level debugging and output of diagnostic messages to console cannot readily be applied. Also, popular operating systems such as Windows and the various variants of Unix do not natively support exporting of their namespaces. Therefore filesystems would have to provide

alternate means to support remote namespace access.

Techniques exist to implement filesystems in user space and integrate them within the overlying operating system namespace. The general idea of shifting functionality from within the kernel to user space has been aggressively adopted by the GNU Hurd [64] and Exokernel [46] projects among others to several core aspects of operating system operation including inter-process communication, filesystems, signal handling and networking. Such systems have a minimal microkernel at their core, while much of the operating system functionality is transferred out to user space daemons.

Fuse [52] and Samba-VFS [103] enable building user space filesystems for use with 'regular' operating systems. These tools use the VFS switch in the kernel to redirect relevant system calls to their respective kernel modules, which in turn forward the requests to associated filesystem logic implemented in user space. The filesystem logic provides implementations of callback functions for the various file operations. With Fuse, a table listing these callback functions is registered using the associated kernel module along with a path for the filesystem mount point in the global namespace. Caching within the filesystem can be disabled during operation using the *direct_io* mode, making Fuse well suited for implementing file interfaces for hardware. The Fuse kernel module for a 2.4.21 Linux kernel executing on an x86 platform requires a little less than 25KB of code memory and 1100 bytes of data memory, thereby imposing acceptable memory demands for the embedded domain. Samba is an open source implementation of the SMB-CIFS distributed filesystem paradigm developed by Microsoft. Samba-VFS is a capability built into Samba that allows userspace code handling various file operations to be inserted into the filesystem and invoked each time an incoming request is being processed. This provides a mechanism to implement synthetic filesystems in user space that reside within CIFS based filesystems. These implicitly support the SMB network resource sharing protocol and can thus be accessed remotely using SMB clients. Samba implementations

are fairly heavyweight partly because they are designed to operate in a dynamic environment in conjunction with a resource location service location and require the use of TCP-IP networking. The kernel module for CIFS under Linux using a 2.6.9 kernel has a text segment size of 164 KB and data segment size of 4KB.

Filesystems such as Fuse that do not provide remote namespace access need to be exported indirectly through distributed filesystem mechanisms such as NFS [105], AFS [92] and CIFS [35]. NFS uses the generic VFS layer to uniformly export a variety of filesystems over the network. While NFS has been widely used to provide remote access to conventional files, it is not as well suited for device access. The initial NFS implementation (NFSv3) was stateless and failed to implement *ioctl* operations, which enable device specific control operations. While the latest version (NFSv4) supports *ioctls*, the intricate caching mechanism central to NFS's performance impedes its use in implementation of synthetic filesystems. The caching, performance and consistency features built into NFS leads to increased complexity in implementation as reflected in its memory requirements. The NFS server kernel module for the 2.6.9 kernel on an x86 platform has a text segment size of 147 KB and a data segment size of about 50 KB. Apart from the NFS kernel module, memory is also consumed by RPC modules required by NFS, such as *sunrpc*.

The synthetic filesystem infrastructure used in this dissertation is implemented in a distributed manner using 9P based filesystems executing in userspace. 9P is well suited for use as the remote resource access protocol in our work. The protocol's ability to support synthetic filesystems with diverse data sources is reflected by its use in Plan 9 to uniformly access a variety of distributed resources including devices, services and file systems. 9P supports non-caching RPC style operation which makes it well suited for hardware. The transport layer requirements from the protocol are in-order, reliable message delivery. This makes 9P usable in systems incapable of implementing TCP or UDP, which offer support for only basic communication infrastructure such as serial ports,

USB or low data-rate radio. The limited number of message types in 9P keeps the protocol simple. The maximum length of messages is fixed at 8192, and can be reduced further by appropriate usage semantics such as restriction of data count in write and read operations. The simple protocol design has allowed the implementation of 9P filesystems with code size of less than 15KB.

2.3 The Plan 9 Model

As described earlier, filesystems abstracting various resources and exported over communication links form the basic building blocks using which distributed systems are built in the Plan 9. Drawing from this idea, distributed filesystem abstractions provide the framework supporting various embedded software development tasks in our work. Filesystems exported from various entities within an embedded system are progressively composed to create higher level filesystems at the system and workstation level. Using these filesystems, client applications on the workstation are able to consistently use familiar file operations to debug, trace and configure software executing on the embedded devices.

Two ideas central to the Plan 9 model are the uniform treatment of resources as files and easy mechanisms to export filesystem contents over communication links. The non-typical use of file abstractions in Plan 9 is clasically illustrated in its handling of TCP-IP network stacks [101]. The interface to network protocols is encapsulated by the `/net` directory in the root file system, which has a namespace as shown below:

```
/net
   /tcp
     clone
     /1
      ctl  data  status  listen
   /udp
     clone
     /1
      ctl  data  status  listen
```

Each major network protocol class has a corresponding subdirectory with multiple 'connection' directories containing files for creating, accessing and managing connections using the protocol. To create a TCP connection, IP address and port number of the remote socket is written to the `ctl` file inside one of the connection directories. Once connected, the connection `data` file is written to and read from for sending and receiving data on the socket. The state of the connection can be monitored using the `status` file, while incoming connections may be accepted by reading the `listen` file. File representation enables network stacks to be exported and used remotely on machines that might not have them implemented locally due to security, computational or other reasons. The idea has been used [98] to implement controlled access to networks using gateway nodes, which export their */net* file trees for use by machines within the network to access the external internet.

9P protocol

[CHG:9p]Clients access exported filesystems using the 9P protocol. The protocol consists of a small set of request/response message pairs to support file system operations. The message pairs fall into the basic categories of connection establishment, navigation over the file hierarchy and file access. To start with, clients connect to filesystems using the *attach* message, which establishes an integer file descriptor as a pointer to the root of the tree. The integer file descriptors used to reference files (known as *fid*) are chosen by the client unlike in Unix wherein they are chosen by the server. The *fid* can be navigated through the exported namespace to point to the node of interest through a series of *walk* operations performed using 9P messages of the same name. *Clone* messages enable new *fids* to be created that are made to point to existing *fids*, specified as part of a clone operation. The file pointed to by *fids* may then be *opened* and *read* from, *written* to; each of these operations is performed through associated 9P messages, a complete list of which has been described by Presotto et al. in their network organization paper [101]. As mentioned earlier, the *fid*

pointing to the root is established by a client when it first establishes connection to a server through an *attach* transaction.

9P allows multiple simultaneous outstanding requests and provide tags to allow the server and client to correctly associate requests and responses; there is no requirement that a server respond to requests in order. The simple protocol design is reflected in directories being treated as conventional files that can be read to access contents, requiring no additional special messages. Similarly, device control is enabled through textual commands written to *ctl* files, thereby avoiding the need for device specific ioctls.

The server responds to all "walk" transactions relating to navigation with the *qid* value of the file being navigated to, which provides a definitive means by which to uniquely identify the file. *Qid* values are 8 bytes in length and consist of two parts - a 4 byte *path* field that serves as a unique identifier for the file and another 4 byte *version* field that is modified each time a file is changed. Two qid's are the same if and only they refer to the same underlying object. Two files within an exported namespace are the same if and only if they have the same qid. File attributes may be retrieved by the client using *stat* operations. These return serialized encodings of the file attributes including file owner/group information, creation/modification timestamps, file Qid, length and other attributes.

2.4 Implementing 9P filesystems

9P filesystems which are the core of the Plan 9 model implement file abstractions for various in-system resources and export the resulting namespace using 9P. While the Plan 9 operating system provides native support for implementing and hosting 9P filesystems, unavailability of ports for common embedded architectures and incompatibility of Plan 9 with support software (such as

device drivers) associated with hardware being abstracted making it unsuitable for being used within embedded systems for supporting file abstractions. Hence we use standalone 9P filesystems that concurrently execute at various levels of the embedded system hierarchy and in combination support a distributed filesystem abstraction.

[CHG:9pfs]9P filesystem implementations consist of three parts corresponding to - 9P message processing, namespace management, and filesystem specific operations . The three work in conjunction to provide the filesystem functionality. The message processing component manages 9P protocol related aspects, most significantly the marshalling/unmarshalling and input/output of messages. The namespace management component handles client namespace navigation through *walk* operations and also generic filesystem operations such as *stat*, *clone* and *clunk* - whose functionality is not specific to any particular device being represented through the filesystem. The filesystem specific component defines the layout of files and directories in the filesystem's namespace and the behaviour of the filesystem in response to the file operations of *read, write and open*. In implementing a 9P filesystem, a user is only responsible for implementing the filesystem specific component, while the rest of the filesystem implementation stays the same.

is what gives a filesystem its character, and has its implementation change across filesystems, while the rest of the filesystem implementation stays the same.

In the Plan 9 world, the layout of files and directories in a filesystem's namespace is specified in a compact and intuitive manner using directory tables. A directory table contains a series of entries, each corresponding to one file or directory in the namespace and containing the file name, a unique integer file identifier, parent identifier and file access permissions. Consider a 'test' device with a directory layout as shown below:

```
/testdev
```

```
a
/b
  c
  d
```

The layout may be captured using directory tables (implemented in C by an array of 'Dirtab' structures) as shown below.

```
const Dirtab testtab[]={
  /* name, unique-id ,parent-id, permission */
  "topdir", Qtopdir , QNobody, DEFAULT_PERM,
  "a",      Qa,     , Qtopdir, DEFAULT_PERM,
  "b",      Qb,     , Qtopdir, DEFAULT_PERM | DIR_FLAG,
  "c",      Qc,     , Qb,      DEFAULT_PERM,
  "d",      Qd,     , Qb,      DEFAULT_PERM
};
```

The Dirtab structure is defined to reflect the format of directory table entries. Qtopdir, Qa, Qb and Qc are unique integer constants used to generate the Qid path values for their respective files in the exported filesystem.

In implementing a 9P filesystem, a user is only responsible for implementing the filesystem specific component, which is done by presenting the namespace contents using directory tables and by defining handlers for *open, read* and *write* operations performed against these files. The rest of the filesystem implementation (9P message processing and namespace management) is drawn from base software sources.

Existing implementations of standalone userspace 9P filesystems are available from Plan9ports

[91] and NPFS [95] projects which provide adaptations of Plan 9 related software for Unix-like operating systems such as Linux, FreeBSD and Mac OS X. Using these directly within embedded systems is infeasible because of the resource requirements they impose and due to their inability to readily provide specific functionality required in the presented model. Consequently we use them as a basis to implement three filesystem blocks each of which fulfills a different requirement within the distributed filesystem model, as reflected in their roles and design emphases. The 'embedded filesystem' is used in embedded environments and is designed for judicious resource consumption. The 'mount filesystem' facilitates integration of embedded filesystems at the component level into filesystems at the system level thereby providing the filesystem model with its compositional capability. The 'clone filesystem' is designed to enable exporting of resources such as console IO, mass storage, and TCP-IP sockets from within workstations by abstracting them using file representations. The exported resources are then remotely accessed by resource impoverished embedded devices using file operations performed on files in the locally mounted clone filesystem.

Embedded Filesystem

At the core of the architecture is the embedded filesystem (EFS) that encapsulates a basic design unit within an embedded system (such as a SoC device) using a file interface. Our implementation of EFS is a stripped down version of the 9P fileserver from Plan9ports. The filesystem is exported over available generic communication links, which in our work has included serial connections, low power radio links and ethernet. t

Generating Filesystems Using EFS

[CHG:efs-use]The namespace exported by EFS filesystems consists of a collection of directories, one for each sub component (interchangeably called *device*) within the entity implementing the EFS. As

an example, for a filesystem abstracting a circuit board, each on-board chip could be represented with a separate directory. The listing below shows the namespace for an EFS filesystem within our prototype presented in Chapter 3, that contains two major sub components in the form of microcontroller devices from MSP and ARM families. Each device has its representative directory with files to program, control and debug it.

```
/EFS
  /mspdir
    registers
    memory
    status
    control
  /armdir
    registers
    memory
    status
    control
```

Every EFS instance contains a base 'tree' device which provides the directory framework onto which various device directories are attached. In the above example for instance, the tree device contains the *mspdir* and *armdir* directories onto which the devices attach their respective directories.

The directory layout of each device in the EFS is specified using a directory table (described earlier in this section), implemented as an array of Dirtab structures. Along with the namespace, each device specification also contains handlers defining behaviour corresponding to device initialization and device specifc file operations or *open*, *read* and *write*. The various devices with their

namespaces and file operation handlers lend each filesystem its character. To implement an EFS based filesystem abstracting the `test` device presented in the last section 2.4, a directory table representation of the device's supported namespace is created, as illustrated using the `testtab` array. A *device record* is defined which lists the device directory table along with functions handling device specific file operation performed on the files, as shown below:

```
/* device record for 'test' device */
Dev testdev = {
  'T',
  &testinChan, /* channel to receive requests */
  &testoutChan, /* channel to send requests */
  testinit, /* device initialization routine */
  testopen, /* handler for open */
  testread, /* handler for read */
  testwrite /* handler for write */
  /* other routines - omitted */
}
```

The `testdev` device record is registered with the namespace management component of the filesystem which then exports the device directory contents as part of the filesystem. Apart from file operation handlers, the device record also specifies a pair of channels (`testinChan` and `testoutChan`) over which respectively file requests for the device are received and responses sent out. Role of channels in the EFS implementation is described in detail later.

In performing a *read* operation as shown :

```
read(buffer, ''topdir/a'', 5);
```

a client starts by issuing a pair of *walk* requests to navigate a `fid` pointing to the root of the mounted EFS to first the `topdir` directory and then to file 'a' in the `test` device namespace; the walk operations are processed by the namespace management component of the EFS using the device's directory table to identify its namespace layout. Following this, the client issues a *read* request on the walked `fid`, which is identified as targetting a file associated with the `test` device by the 9P message processing component of the EFS, and ultimately handled by the *read* handler of the `test` device.

[CHG:EFS]An underlying assumption behind use of EFS filesystems is that both the set of devices being represented and files associated with each remains constant in the course of execution. As described in the next subsection, the static nature of its namespace contributes to the simplicity of the EFS design. Since the set of processor elements associated with each SoC device remains constant over its lifetime, representing them using an EFS filesystem based on a fixed set of device directories is warranted. While it is possible to conceive of scenarios wherein it would useful to create and destroy files within device directories in an EFS filesystem (to reflect process creation/deletion for instance), our EFS block does not support such functionality. The *mount* and *clone* filesystem building blocks in contrast, respectively have the ability to dynamically generate their namespaces and modify it at runtime.

EFS Implementation

In order to adapt the Plan9port fileserver design to the embedded domain, we limit the filesystem functionality and move to a single threaded implementation using cooperatively scheduled routines that is based on the CSP model [33]. The resulting filesystem has been implemented with a code size of less than 15KB.

The EFS implements a large subset of standard filesystem features, while excluding a few others

to limit complexity. The filesystem supports the core 9P file operations relevant to EFS functioning, namely: *attach*, *walk*, *clone*, *read*, *write*, *open* and *clunk*. The non supported operations are *create*, *remove* and *wstat*. The choice of operations to exclude is consistent with static namespace characteristic of EFS wherein files cannot be added or deleted from the namespace.

The filesystem limits the number of files that may be opened concurrently to a maximum number that is set at compile time. While the limit varies from one application to another, the rule of thumb is to ensure that it is larger than the number of distinct files in the exported namespace. Since open file descriptors for various files in the namespace can be reused, maintaining multiple open file descriptors to the same file is sometimes redundant. Thus limiting the number of open file descriptors is not always overly restrictive for clients. As another means of reducing operation state, the filesystem only allows a single active client connection at a time. The next section describes use of the overlying mount filesystem to deal with this restriction.

A standard procedure in server design is using multiple dynamically generated threads to concurrently process incoming requests. Liberal use of multi-threading is infeasible in the embedded domain because of the associated memory overhead. Our design of the EFS block - depicted in Figure 2.1 - is based on the CSP model [33] and uses a fixed set of interleaving threads communicating over uni-directional, singly buffered channels. Channels provide a mechanism by which threads can both communicate and synchronize in CSP based systems. The two operations that threads can perform on a channel are *chanrecv* and *chansend*, to receive and send data on the channel respectively. Since the channels are singly buffered and blocking, they follow rendezvous based semantics, which means they block until another thread is ready to perform the complementary operation on the same channel. The code below shows an example of a thread denoted by variable 't' performing a receive operation on channel 'c'.

```
if (chanrecv(&t, &c) < 0)
```

```
threadReturn();
```

The *chanrecv* function call returns a negative value if there is (are) no compatriot thread waiting to send on the same channel. In such a case the thread voluntarily yields using *threadReturn* and is blocked until when the operation can go through.

Each thread in the EFS implementation is responsible for a different aspect of filesystem operation, namely:

- input/output (IO)
- buffer management
- filesystem management
- device management (one thread for each device)

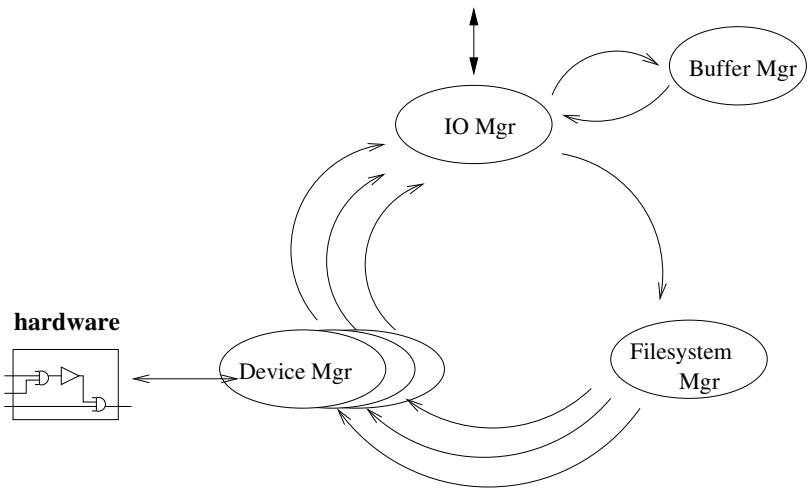

Figure 2.1: EFS Design

When a file request comes in, it is accepted by the IO thread and copied into a message buffer that is retrieved from the buffer management thread. The message buffer is passed on to the

filesystem management thread which deciphers the message to retrieve its type and associated arguments. Request types corresponding to generic file operations such as open, clone, attach and clunk are processed and responded to by the filesystem thread directly. Other device-specific requests such as read and write are passed on to the appropriate device manager as determined by the path/Qid of the file being targeted. This packet handoff occurs over the device's input channel which is registered with the filesystem core at startup time. The structure of the filesystem management thread's main loop is shown in Listing 2.1.

Listing 2.1: Filesystem Management Thread Loop

```
for(;;) {
  if (chanrecv(&inChan, &rx) < 0) {
    threadReturn();
  }
  switch(rx->type){
    case Topen:
        /*
          file open logic (omitted)- generates return packet 'tx'
        */
        if (chansend(&outChan, &tx) < 0)
          threadReturn();
    break;
    case Tread:
        devindex = lookupdev(rx->fd); /* find device being targetted */
        if (channbsend(devtab[devindex]->chan, &rx) != SUCCESS) {
          /* device busy; return error */
```

```
            tx->type = Rerror; tx->ename = EBUSY;
            if (chansend(&outChan, &tx) < 0)
              threadReturn();
          }
        break;
        /* handle other operations - omitted */
      }
    }
```

The filesystem first obtains an incoming request from the IO thread through *inChan*. Each request type is then handled within a switch statement. The *open* operation is handled locally by the filesystem manager and the response sent back to the IO thread for being returned to the client. The handling of *read* is more involved and begins by identifying the targetted device using the *lookupdev* function. The request is then sent to the appropriate device through its designated channel in a non-blocking manner. Use of non-blocking send enables a busy error to be returned to the client in case the device thread is busy processing an earlier request, so that the filesystem can continue serving new requests. Use of blocking send would in contrast result in the filesystem manager blocking until the targeted device's thread is ready to accept the new request, thereby making the whole filesystem unresponsive during the wait.

A device's software specification consists of 4 components namely: its directory table, table of function handlers for supported message types, a device thread and a pair of device channels, one each for receiving incoming requests and sending out responses. The thread waits on its device input channel to receive a request from the filesystem manager. The main loop of the device thread is similar to that for the filesystem manager with the difference that only the device specific messages

types (commonly read and write) are handled. Upon receiving a request, the thread engages in a series of interactions with its associated hardware to complete the request and passes the response back to the IO thread through the output device channel.

Multithreading in the EFS design is implemented using the *libthread* module from Plan9ports in a stack-less manner, along the lines of Protothreads [44] and Stackless Python. Threads are scheduled cooperatively, which means that shared data across threads does not have to be safeguarded using mutual exclusion constructs. When a thread yields, only the yield point in the execution is preserved and not the contents of the thread stack. The state that needs to be restored the next time the thread is scheduled needs to be stored within static variables. [CHG:singlthrd]While this means that only one 'instance' of a function can run within any thread (thereby making it non reentrant), the context associated with the thread is simply a memory address, which is 2 to 4 bytes long depending on the architecture. In contrast, storing context for a user level thread using *getcontext* in Linux takes up 348 bytes of memory, in addition to memory for storing contents of stack. By use of cooperative scheduling, pseudo threads comprising the EFS implementation are assumed to yield their execution responsibly when blocked on channel operations. Divergent behaviour on the part of any of the threads could cause other threads to be starved of execution time and therefore lead to the operation of entire filesystem to stop.

The CSP-based filesystem is designed in the form of a packet processing pipeline [75] consisting of a series of concurrently executing threads. The thread in each stage partially processes the request and hands it off to the appropriate next stage over a channel. Since blocking channels are used, threads automatically yield if there are no requests waiting to be processed. The system therefore automatically schedules itself based on the flow of requests through the pipeline. The pipeline is nonlinear since it fans out from the file system management thread to multiple device

managers, and converges at the IO thread from where responses are sent out. The IO thread simultaneously listens on all device output channels using the *alternative* operation, which blocks the thread until at least one of the channels has data available. When the operation returns, data is read from one of the possibly many 'ready' channels, the choice being made in a unspecified manner - in our case at random. Performing an 'alternative' operation first involves defining a table of the *Alt* structure entries each of which lists a participating channel, the operation to be performed on it (send or receive) and the variable where data is either sent from or stored into depending on the operation. The alternative operation is then run using the *alt* construct, which is structured like a switch statement with the various cases indicating completion of operations at corresponding indices in the table. Code from the IO thread using the alternative operation is shown in Listing 2.2 for a case consisting of a single device within the filesystem named 'test' apart from the standard 'tree' device:

Listing 2.2: Use of Alt in IO Thread to receive outbound packets from multiple devices

```
static Alt IOalt[] = {{&treeOutChan, &tx, CHANRCV},
                      {&testOutChan, &tx, CHANRCV},
                      {0, 0, CHANEND}};
while(1) {     /* forever .... */
    alt(IOalt) {
      default:
        /* code for sending tx out - omitted */
    }
    if (chansend( &rtnbufferChan, &tx) < 0)
       threadReturn();
}
```

The alt statement returns when a response packet is obtained from either of the devices (and stored in pointer location 'tx') which is then sent out. The buffer is then returned to the buffer management thread for reuse.

It may be recalled from the previous section 2.4 that integer identifiers for each entry in the directory table are used to generate 4 byte Qid path values for its associated file in the exported namespace. With multiple devices however, file entries across different directory tables could potentially share identifier values, thereby violating the unique Qid requirement. To address this issue, we divide the 32-bit path into two fields - a 3 bit device value and a 28 bit file identifier derived from its entry in the directory table. As per 9P protocol specification, the remaining MSB bit is used to specify whether the file is a directory . This technique for generating Qid values provides a basis by which the device associated with any given file may be identified merely based on its Qid. The Qid path format is laid out in the Figure 2.2 below:

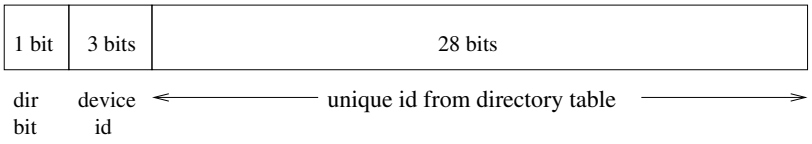

Figure 2.2: Qid path format

Mount Filesystem

Various EFS instances within an embedded system may be composed within a mount file system (MFS) that provides a single, uniform interface for the entire system. An MFS block is typically implemented in an entity that is hierarchically above where the EFS blocks reside. For instance, in an SoC based system, EFS units may be exported by each of the SoCs, which are integrated by an MFS implemented at the board level. In board based designs, the MFS may be implemented at the

workstation level.

The MFS block is responsible for providing the filesystem model with its compositional features. The key to this is its ability to mount lower level 9P based filesystems (both EFS and MFS) and use them in its own operation. The capabilities provided by the MFS are two-fold. It facilitates implementation of 9P filesystems that can mount remote filesystems and incorporate them in their own implementation. MFS also supports reexporting of various mounted sub filesystems all aggregated under a single 9P filesystem namespace.

Re-exporting Imported Filesystems

The challenge in implementing a union of sub filesystems under MFS is to ensure unique Qid values among files. It may be recalled from Section 2.3 that each file in an exported namespace has an associated Qid value that serves as a unique identifier. Since files across various imported EFS instances may share Qid values, they have to be made unique before being reexported as part of the combined MFS namespace. This issue has been investigated by the Plan9 community, most notably with regard to the implementation of *exportfs* [6]. The solution we use is to statically divide the MFS Qid space among various filesystems being aggregated and to map their file Qid's to within these spaces before re-exporting them.

Import of remote filesystems in MFS is supported through a separate device known as *mntdev*. Each instance of mntdev acts as a client to a single remote 9P filesystem. Unlike conventional devices wherein the contents of the device directory is specified using directory tables, with the mntdev device the information is retrieved from the external filesystem. All incoming requests are processed by forwarding them to the remote filesystem and returning the response packet. For walk operations the returned Qid value is altered to ensure uniqueness. This is done by setting 4 higher order bits of the Qid path value to an identifier that is unique to the mntdev instance. Along

with the device identifier bits, these ensure that Qid path for each exported file is unique. The 32-bit Qid path value thus has the format shown in Figure 2.3.

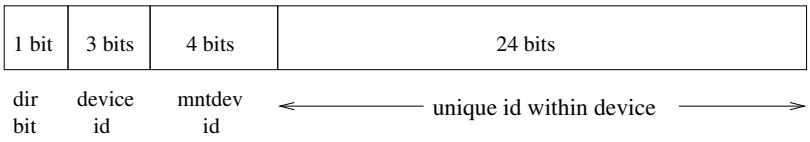

Figure 2.3: Qid path with mntdev bits

[CHG:mfs] This format extends the one presented earlier in Figure 2.2 with an additional 4-bit field that specifies the mntdev identifier. Use of this format imposes the restriction that a maximum of 16 EFS filesystems can be mounted together at one time, which is equivalent to making the reasonable assumption that no more than 16 SoC devices are present within an embedded system, whose filesystems need to be mounted together. The partitioning also restricts the number of files that can be associated with each device in an imported EFS to 2^{24}, which is far more than the number that would typically be required. An alternate approach that does not require equal division of the Qid space, maps mntdev Qid path values to files in imported filesystems on a need-be basis. This requires the implementation to explicitly store the mapping between Qid's in the imported namespace to those in the MFS.

Addressing EFS limitations

Since the MFS executes in computationally capable entities higher up in the embedded hierarchy, limitations in lower level EFS blocks can be tackled within it. We present two ideas in this regard that are proposed as future work. To address the limitation of EFS supporting only one active client, the MFS can multiplex a single connection [14] to each underlying EFS block among multiple clients. The MFS can also throttle incoming requests directed at EFS blocks that can only handle a limited number of concurrent requests. Protocol bridges can be incorporated within MFS to allow

indirect access of EFS using protocols other than 9P. Thus, by using these techniques end users are saved from having to account for the various limitations implicit in EFS blocks by handling them at the MFS level.

Clone Filesystem

[CHG:clonefs]A limitation of the two 9P filesystems (*embedded* and *mount*) described is that the number and type of devices they represent and also the files associated with each device is hard coded in their implementation. Clone filesystems relieve this limitation by supporting the addition and removal of device directories corresponding to the creation and destruction of resources being abstracted as part of the filesystem. Clone filesystems thus provide an effective means of implementing filesystem abstractions for transient entities associated with system execution.

An example of the use of clone filesystems is in the implementation of networking in Plan 9 through /net as described in Section 2.3. TCP connections (sockets) are represented by unique directories within the /net filesystem, which contain files to configure and use the connection. Clients create and destroy connections by adding *connection directories* and removing, which abstract the connection using file interfaces. After initialization, the TCP section of the filesystem contains a single file named *clone* as shown below:

```
/net
   /tcp
     clone
```

A new connection is created by opening the 'clone' file, which has the effect of creating a new *connection directory*. The file descriptor returned from the open operation points to the *ctl* file in the connection directory. When read, the 'ctl' file returns the connection number which can then be

used to open other files for configuring and using the connection. Apart from networking, clone filesystems are used in Plan 9 to implement a process filesystem [121] similar to /proc in Linux, which provides a file based interface to monitor, control and debug processes executing within the operating system. Process directories are continuously added and removed from the global namespace to reflect process creation and termination.

In this dissertation, the work presented in Chapter 5 uses 9P based clone filesystems to export workstation-based resources such as network stacks, peripherals etc. over communication links. 9P client applications executing within embedded devices import these filesystems and thereby gain indirect access to the exported resources. Thus the roles of clients and servers are reversed between the workstation and embedded sides as compared to the ideas presented thus far.

Extending NPFS

The clone filesystems in our work are implemented using NPFS [95], which provides a framework for building robust 9P filesystems in C for Unix-like operating systems. NPFS supports use of multi-threading to serve multiple concurrent client requests and direct mounting of the implemented filesystem within the host operating system. While 9P filesystems may be easily deployed using the Plan9port suite of tools, the approach requires use of a custom development tool chain consisting of a non-standard compiler, linker, debugger and thread library. NPFS in contrast, allows filesystems to be developed using familiar GNU toolchains.

We implemented two extensions to NPFS in order to make it suitable for our work. First we incorporated support for use of Dirtab representations to specify filesystem layouts. we then leveraged the Dirtab support to develop a framework for implementing clone filesystems within NPFS.

Conventionally the namespace for NPFS synthetic filesystems is derived at startup time with help from the entity being abstracted. As an example consider the 'mboxfs' filesystem available

within the NPFS sources that provides access to e-mail messages on a mail server through a file interface. The filesystem namespace is generated based on the messages retrieved from the server as shown in the code listing 2.3

Listing 2.3: Filesystem Namespace Generation

```
n = mailsession_get_messages_list(fld->folder->fld_session, &msg_list);
for(i = 0; we < carray_count(msg_list->msg_tab); i++) {
    msg = carray_get(msg_list->msg_tab, i);
    sprintf(buf, "%d", msg->idx);
    nf = npfile_alloc(dir, strdup(buf), 0500|Dmdir, ((u64)msg->idx)<<32,
                                                    &message_ops, m);
    add_file(dir, nf);
  }
```

The gist of the logic is to first download the list of messages from the server and for each message create a directory (using npfile_alloc) with the name given by the message index.

Use of this approach in our work is often infeasible as there may be no well-defined entity that is being abstracted with which to generate meaningful namespace information. Directory tables in contrast allow implementation of filesystems based on arbitrarily defined namespaces. Code generating the namespace information using directory tables is implemented in libraries and is reused with different filesystems. Associated file information such as access permissions is stored centrally within the directory table, making it easy to change the filesystem namespace contents.

Handling /textitwalk with Directory Tables

Implementation of the handler for 'walk' in Dirtab based filesystems is described, as this is where file descriptors get mapped to files based on path names and hence deals most closely with directory tables. The functionality of the walk operation may be stated as follows: given a file descriptor pointing to a directory and a name, to remap the file descriptor to a child file/directory with the given name. Using Dirtab representations, this is achieved by retrieving successive directory children and checking to see if their name matches with what is being looked for. The code in listing 2.4 illustrates this idea, wherein variable 'f' (of type `Fid`) is the file descriptor pointing to a directory and variable 'name' contains the name of the child being searched for.

Listing 2.4: Handling 'walk' with directory tables

```
offset = 0; /*start at the top of the table*/
while (!found) {
    dt = findNextDirChild(&offset, f->qid.path);
    if (dt == NULL)  /* no child found */
        break;
    if (strcmp(name, dt->name) == 0) {
        /*
            modify 'f' to point to child 'dt' entry - details omitted
        */
        found = 1;
    }
}
```

The helper function *findNextDirChild* retrieves the first entry within the filesystem Dirtab table beyond the given offset with a matching parent. This function is repeatedly invoked to retrieve successive directory entries until a matching one is found or no more entries are left.

Supporting clone filesystems using NPFS

Dirtab descriptions provide a convenient means by which to implement clone filesystems using NPFS. The namespace exported by every clone filesystem has a static part that remains unchanged during the course of filesystem execution and a dynamic part to which 'clone directories' get added each time a clone operation is performed. In the /net filesystem described earlier, the 'clone' file is the solitary member of the static part, while clone directories representing connections get progressively added to the dynamic part through clone operations.

Dirtab structures are suitable for implementing clone filesystems because in the course of operation their filesystem namespace always changes (grows or shrinks) by a fixed set of files each time. Accordingly, in our approach we maintain two Dirtab directory tables, one for the static section of the filesystem and another for the clone directory. Each time a clone operation is performed, a copy of the clone directory table is 'fused' with the static table at the appropriate location. This process is illustrated in Figure 2.4 for the /net filesystem. The center figure shows the namespace after two clone operations, as consisting of two copies of the clone directory table fused to the static table at directories named '/1' and '/2'. Deletion of a clone directory leads to its corresponding copy of clone table being 'unfused' from the static table, as shown in the figure on the right. While our implementation of clone filesystems does assume availability of dynamic memory, since these filesystems are intended to run on workstations the assumption is warranted.

Multiple clone filesystems can be easily supported using this approach by maintaining one directory table for each clone type. Since static and clone directory tables are isomorphic, nesting

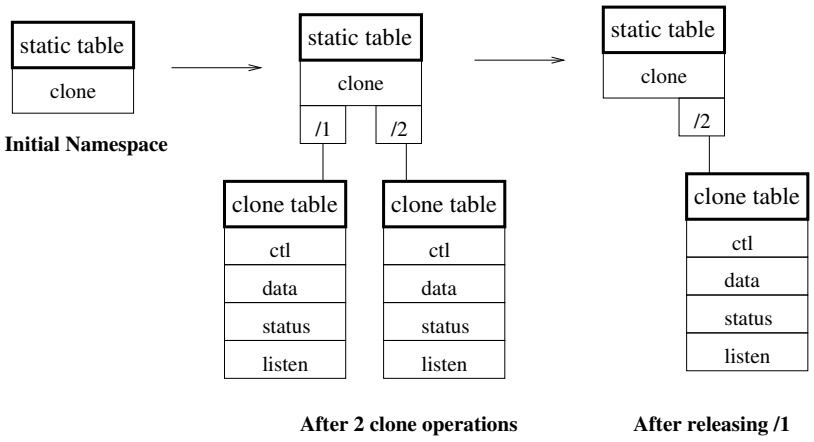

Figure 2.4: Clone operation example

clone filesystems within one another is also possible by fusing clone directory tables to one another.

An issue that needs to be addressed with this approach is that since directory tables are stateless, multiple instances of clone directory tables within a namespace cannot be readily differentiated. So we represent every clone instance in the filesystem by a value pair, consisting of the associated clone directory table as well as a clone handle in the form of a memory pointer. With the /net filesystem for example, the handle for each clone instance can be a pointer to a connection structure that stores socket number of the connection, IP address of the node connected to, the state of the connection etc.

NPFS filesystems supporting clone operations maintain a list of *transfer points*, which indicate locations in the filesystem where directory tables are fused together. The terminology signifies them as being the only locations in the namespace where file descriptors can 'transfer' from one filesystem directory table to another during walk operations. A transfer point can be defined using three values, namely:

- Qid of the parent directory containing the particular clone directories

- name of the cloned directory
- directory table corresponding to the clone directory along with the clone handle

The filesystem is able to associate every file descriptor with its cooresponding clone instance by encoding the relevant directory table and clone handle information within the file descriptor's Qid path bits. A similar approach is used to encode device and mount identifiers in the Mount File System as described in the previous section.

The 'walk' operation handler implementation is modified for clone filesystems as their directory contents are derived from both the respective filesystem directory table and the global list of transfer points. The implementation of *findNextDirChild* presented earlier is modified from the static filesystem case to reflect this. Where earlier the function returned a NULL value when no Dirtab entry was found with a matching parent Qid, now the function continues searching though the transfer point list based on the same criterion.

2.5 Host Access of 9P filesystems

Embedded software development tools for debugging, programming and trace analysis executing on workstations interact with remote embedded devices using files in the exported EFS and MFS namespaces. This section presents a technique implemented as part of our work that enables the exported 9P filesystems to be incorporated within the host operating system namespace under Linux. Software development tools can then access the exported filesystems through conventional file operations, without having to deal with any 9P-protocol related issues themselves. A related project, v9fs [90] implements in-kernel support for importing filesystems based on the 9P2000 protocol, which is a newer version of 9P. Our work targets importing of 9P filesystems, and contrastingly uses userspace software to do so.

The underlying idea behind our technique is to use Fuse [52] as a bridge between Linux and remote 9P filesystems. As described in Section 2.2, Fuse enables filesystems to be implemented in userspace and have their contents included in the overlying operating system namespace. When an operation is performed against a file in the bridged namespace, the Fuse kernel module forwards the operation to our userspace filesystem responsible for bridging 9P and Fuse, henceforth referred to as '9pbridge'. 9pbridge in turn resolves the issued operation into equivalent 9P requests that are sent to the targeted 9P filesystem. Once these requests have been completed, 9pbridge extracts the return value and passes it back to the kernel. The operation is depicted in Figure 2.5, which is based on an illustration from the Fuse documentation.

We present a brief comparison between 9P filesystems and Fuse, focussing on aspects relevant to implementing bridges between them. Both 9P filesystems and Fuse allow clients to create file handles using which various file operations may be performed. 9pbridge associates a unique file handle on the Fuse side for every file (indirectly) opened on the 9P filesystem by clients. In 9P

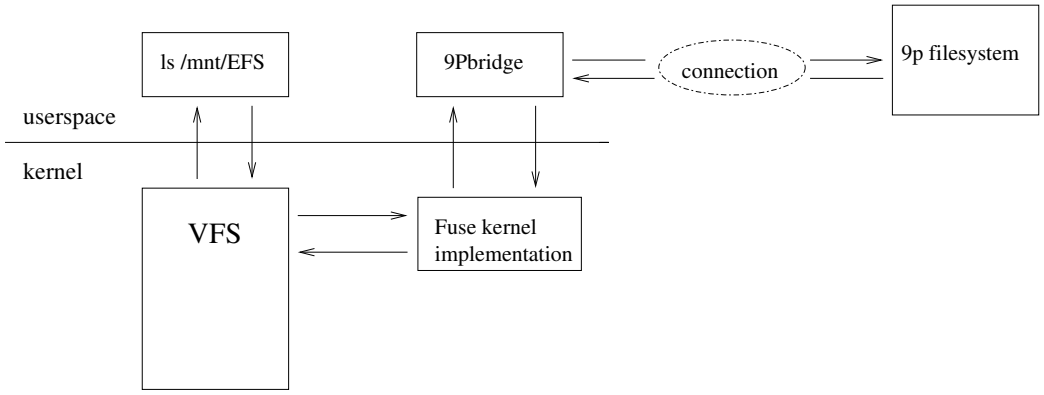

Figure 2.5: Request Processing

filesystems, handles are refered to using integer fid values that are picked by the client each time a new handle is created using a clone operation. Fuse constrastly allows clients to use integer fid's but these are concealed by the kernel module from the userspace filesystem, which instead maintains file handles using the *fuse_file_info* structure that is passed to the filesystem on every operation against the file. The handle contains a generic 64-bit value (named fh) to identify the open file instance, which we use to store its 9P side fid value. Like in 9P, file offsets are not stored within file handles in Fuse, but rather explicitly specified within every operation. This means the offset can be 'passed on' from a Fuse operation to its corresponding 9P operation. In the implementation of read within 9pbridge shown in Listing 2.5, the 9P fid is retrieved from within the fuse_file_info structure (fi->fh) and the read offset passed on directly from the Fuse to 9P call.

Listing 2.5: Fuse to 9P conversion of read operations

```
int 9pbridge_read(const char *path, char *buf, size_t size, off_t offset,
                  struct fuse_file_info *fi)
{
  if (fi->fh)
    {
```

```
        return styx_read(fi->fh, offset, size, buf);
    }
  else
    return -ENOENT;
}
```

Fuse treats directories and files differently unlike 9P, which allows directories to be read like conventional files to yield a series of *stat* structures listing directory contents. In Fuse, the names of constituent files within a directory are first obtained using the *readdir* operation, after which the files' attributes are obtained using the *getattr* operation.

Since Fuse is designed to be implemented locally, some aspects of its operation are inefficient when used in a distributed environment such as ours. We introduce optimizations in 9pbridge to address this issue. An open operation in Fuse reduces to three distinct 9P operations as listed below:

- *clone* of the Fid pointing to root to create a new Fid
- series of *walk* operations of the new Fid along the path of the file being opened
- *open* of new Fid

Since each of these operations is performed in a distributed manner, the resulting latency of the entire Fuse open operation can be considerable. To deal with this issue, we introduce an optimization whereby each file on the 9P filesystem is opened only once, which happens the first time it is opened through Fuse. The obtained file handle is reused to service all subsequent open requests issued on the same file.

A second optimization relates to client retrieval of file attributes, performed in Fuse using the *getattr* operation often in conjunction with open. This operation in Fuse results in a 9P stat operation being issued to the remote 9P server, which replies with a serialized encoding of a *stat* structure containing the file's attributes. Since the structure is 116 bytes in size, the operation is fairly communication intensive leading to a pronounced latency in low bandwidth links, while also consuming valuable power during communication. Reading directory contents leads to multiple stat operations being performed compounding the problem even further. we address this issue by caching stat entries for files and directories within 9pbridge. As with open, the stat operations are issued a single time to the remote 9P server and the returned structure is cached and reused from there on. This optimization is valid in our work as the only attribute of interest within the stat structure is whether the entity is a file or directory, which does not change with time. If stat attributes of interest get invalidated over time (such as modification timestamps), then more sophisticated mechanisms need to implemented to keep the cached copy in 9pbridge coherent.

2.6 Summary

This chapter presented the technology used in our work to build and utilize file abstractions for embedded systems. These abstractions are realized using a distributed, hierarchical organization of filesystems at the chip, system and workstation levels, as implemented using 'embedded', 'mount' and 'clone' filesystem building blocks. Each of these blocks fulfill a different requirement within the distributed filesystem model, as reflected in their roles and design emphases. On the workstation side, we provide a mechanism for mounting the exported filesystems within the local operating system namespace, thereby enabling the contained files to be accessed like any other 'conventional' file. The non-traditional use of filesystems as a standard means to access and control resources

is based on ideas espoused by the Plan 9 operating system. The Plan 9 model presents several advantages as compared to other popular distributed filesystem solutions such as NFS, including simplicity, less resource overhead, ability to be exported over a variety of communication links, and suitability for use with hardware. Chapters 3 and 4 present use of this technology to facilitate software development in two contrasting embedded application domains.

3

Application to Embedded Software Development

3.1 Introduction

In this chapter we describe use of the virtual filesystem technology to support debugging and tracing of software executing within multi-processor embedded systems. These tasks, which form an integral part of the software development process in general, are particularly challenging to support with embedded systems due to the resource constrained, heterogenous nature of their design and the concurrent nature of their operation. The factors in combination contribute to limited internal visibility thereby impeding debugging, and impose unique requirements on the various software development tasks involved, including debugging and tracing. Our approach is able to support these tasks in a resource conscious manner, and in contrast to traditional approaches is able to naturally accomodate the associated requirements unique to embedded systems. We focus on systems based on multi-processor system-on-chip (SoC) devices, which have come to be used as the principal building blocks in embedded design [21] [18].

A fundamental assumption behind the work presented in this chapter is that a scalable solution for programming and debugging systems built from SoCs can draw from distributed programming principles. The presented approach uses portable software modules at SoC level in the form of chip-level file systems as distributed building blocks. These are composed to create higher level file systems that provide a portable, high level interface to debug, trace and monitor software executing within SoC based embedded systems. We demonstrate the ability of the model to support requirements unique to SoC based systems that traditional methods are ill equipped to deal with.

The organization of this chapter is as follows. We first discuss requirements on the part of on-chip structures for supporting SoC software development tasks such as debugging and tracing. This is followed by a description of JTAG, which forms the primary basis of current on-chip debug support infrastructure, along with an account of its shortcomings when used with SoC based

systems. Then We introduce the use of filesystem abstractions to support standard software development tasks and present an architecture for implementing the idea within SoC based systems. Finally we illustrate use of the model to achieve concurrent debugging and tracing in a heterogeneous multi-processor environment using working prototypes.

3.2 Requirements for SoC software development

A system-on-chip is an integrated circuit (IC) that has several major components within a computing system such as processors, memories, peripherals and busses integrated into a single composite device. SoCs offer several advantages such as a high level of integration (resulting in minimal size), power efficiency and intellectual property (IP) reuse. Designing on-chip support for SoC software development presents some distinctive challenges. SoCs offer reduced internal visibility, as compared to system-on-board based design paradigms where the major components exist within distinct chips. In such older design techniques, access to signals and bus traffic was easily available through IO pins on chip packaging, external interconnects on circuit boards etc. With SoCs however, visibility is limited as these signals are buried within the chip along with their respective cores. Thus, basic on-chip support for software development entails providing mechanisms to probe and monitor the internal logic within an SoC.

The system design process using SoCs typically involves two parties: the chip provider who implements the SoC and the system designer who incorporates it within an embedded system [68]. Chip providers need to ensure that this separation does not affect the design process by providing system designers with effective mechanisms to program, debug and integrate their SoCs with systems in which they are being deployed.

Additionally, SoC programming and debug mechanisms must satisfy requirements imposed

by cores residing within the SoC. Concurrent access of the different on-chip processor cores is a requirement to support simultaneous debugging. The cores often have associated proprietary information that the mechanisms must conceal. Abstract interfaces are used to conceal the internal details by providing higher levels of interaction, at the cost of more complexity. Thus chip providers need to be equipped with methodologies to address these issues and make suitable tradeoffs, possibly by choosing the level of abstraction at which to present chip interfaces.

Aside from debuggin and programming of individual cores, on-chip structures should support development tasks for the system at large. An example of this is the cross trigger functionality [3], which enables events in one core set off actions (such as stopping execution) in another. System level monitoring [55] allow execution of a system to be observed without halting it. On chip debug support also enables resources to be shared among different cores during debugging. This is particularly useful in supporting multi-core tracing, by allowing trace buffers and trace ports to be shared between processor cores for storing and streaming out trace data respectively.

3.3 Present Practice

The prime motivation of our work comes from limitations of JTAG [107] (formally IEEE 1149.1 standard) that largely forms the basis of existing techniques to program and debug SoC based embedded systems. A brief overview of JTAG is presented in the following paragraphs followed by a discussion of its limitations when used with SoC based systems.

Overview of JTAG

JTAG was originally developed for chip testing, but has come to be used for other purposes such as programming and debugging. Its compelling advantage is being able to function with as few as

4 pins on the chip packaging, corresponding to the TDI (test data in), TDO(test data out), TCK (test clock) and TMS (test state machine control) signals. The typical layout of a chip supporting JTAG is shown in the Figure 3.1.

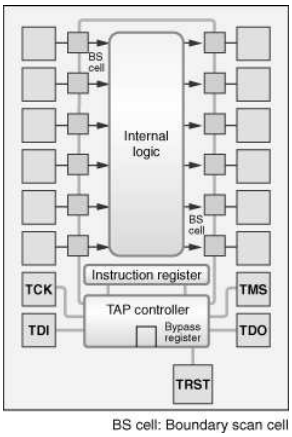

Figure 3.1: Typical on-chip JTAG architecture (courtesy: Sanyo Semiconductors)

Devices supporting JTAG have one or more chains of single bit boundary scan (BS) cells that run typically around the periphery of the chip. The chains start and end with external pins named TDI and TDO respectively. Each of these chains can be thought of as a data register (DR) whose size in bits is equal to the length of the chain. In addition, JTAG implementations include an instruction register (IR) which is connected to TDI and TDO in parallel with the data registers. To load data into a register, the required one is selected using a series of chip-specific operations involving TMS following which data is shifted in through TDI.

The basic operation sequence involved with JTAG entails loading the IR with an instruction opcode, DR with an instruction operand followed by an invocation of the instruction. The result of the operation is shifted out through TDO. The instruction set supported varies across devices

and is sometimes proprietary information. Using this mechanism, processor chips and cores allow examination of processor state, injection and execution of instructions and control of processor execution through start, stop, break and single step operations. The procedure is closely tied to the chip involved in terms of the on-chip core layout, length of scan chains etc. and is in general non-portable.

Limitations of JTAG

JTAG based techniques require significant information about hardware internals to be divulged. Descriptions of scan chains may have to be made available (eg. through BSDL files) for use by programming and debugging tools. This is undesirable in a SoC scenario, where chips may contain cores that have associated proprietary knowledge that chip providers must safeguard.

Processor chips and cores from different organizations often provide incompatible custom debug interfaces based on JTAG. For instance, the MSP and ARM based microcontrollers used to build prototypes presented later in this chapter provide JTAG interfaces, but whose usages are entirely different. These differences limit the interoperability of tools among processors. Presence of cores within an SoC that provide forms of debug support other than JTAG (such as Motorola's BDM) leads to a proliferation of debug interfaces exposed from within the chip.

JTAG requires physical access to a dedicated hardware port. The port along with the JTAG scan chains are shared resources among the various on chip cores. This makes concurrent access and debug of multiple processor cores in an SoC using JTAG non trivial. JTAG also suffers from being a bit serial protocol which compromises the speed at which data can be transferred into and out of the chip. Hence, often a higher speed link is used to support operations such as tracing which require greater throughput.

Traditional software development processes based on JTAG are ill-equipped to handle the partitioning of the SoC design task into creation of programmable platforms by chip provider's team of hardware-centric engineers and use of these platforms by system designer's team of application and software-centric designers. As an example depicting this partitioning, consider the MIPS based Broadcom universal optical disc (UOD) system-on-chip that is used by LG Electronics in a model of its DVD players [32]. The chip provider (Broadcom) needs to provide functionality in its SoC device (UOD chip) to enable the system designer (LG Electronics) to program and debug the device, while safeguarding proprietary details of the cores (MIPS processor) within the SoC. To do so, the chip provider needs to be able to choose the levels of abstraction at which to present interfaces for their platforms in order to manage complexity, to conceal internal proprietary details etc., while still providing the required programming/debug functionality. JTAG-based approaches are incapable of exposing higher level interfaces that chip providers may want to expose. This is partly because JTAG was designed in times when integrated circuits consisted only of a few thousand transistors, which meant on-chip debug support had to be extremely frugal in terms of the hardware resources consumed. In contrast, integrated circuits today have millions of transistors which makes accomodating on-chip logic to support debugging more feasible. The emerging trend is to move away from testing interfaces more towards dedicated system debug interfaces [62], such as those proposed by the Nexus standard [20]. This dissertation represents a step in the same direction.

3.4 Methodology

This section introduces the representation and use of file abstractions for SoCs. We present how the abstraction may be extended to support requirements specific to SoCs.

Filesystem Representations for System on Chip (SoC)

Consider a heterogeneous chip-level processing environment shown in Figure 3.2 with two conventional processors and a custom DSP. To manage a chip-level file system for such an SoC, our

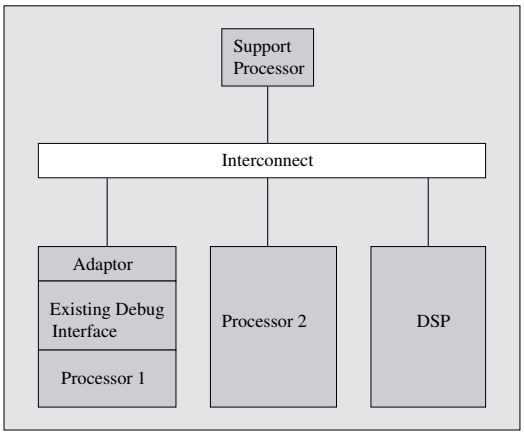

Figure 3.2: Model for chip-level filesystems

model introduces a support processor within the system design, that executes portable software modules to implement the file system. This file system is exported off chip over generic communication links and can be used directly by an end application (such as a debugger) or combined with file systems from other chips to create higher level file systems. Hardware adaptors can potentially be used to integrate processors with existing debug interfaces (processor 1 in figure) with the support processor.

The contents of the filesystem exported depends on the operations being supported by it. Debugging code on a processor requires mechanisms to access and modify registers and memory, to start and stop the processor, and to enable and disable breakpoints (watchpoints). A file system that supports debugging would enable these "run control" capabilities through conventional file operations on files in its exported namespace. An example chip level file system to support on chip debugging is shown below:

```
/System/
      processor1/
         control
         registers
         memory
         status
      processor2/
          ...
      DSP/
          ...
```

Each processor core is represented as part of the file system namespace and its corresponding directory contains files to access registers and memory, monitor status and control execution of the processor. With such a file system in place, basic run control (such as a register read) can be achieved through file operations (reading the *registers* file), which are reduced by the filesystem implementation into the required low level operations on the core.

Use of intermediary hardware to bridge software tools with core debug interfaces is common. For instance, ARM architectures use a dedicated hardware unit called Multi-ICE, while Motorola MCUs use BDM pods to achieve this objective. While these units exist external to the chip, in our model an on-chip support processor core fulfills an analogous role. With increasing transistor

densities, dedicating a fraction of the on-chip logic to support debugging is feasible [116].

Addressing SoC requirements

Filesystem abstractions provide internal visibility within SoCs by exposing processor state through files. Unlike with traditional JTAG based methods, in this model interaction with the SoC takes place over generic communication links. Use of high speed links such as USB can potentially yield much higher throughputs. While the actual value of throughput with JTAG depends on device characteristics such as scanchain lengths, the typical number is less than 10 Mbits/sec [9]. In contrast, USB 2.0 can support data transfer speeds of upto 480 Mbits/sec. While the overhead imposed by use of file abstractions would the limit the actual bandwidth that can be realized, generic communication links enable use of faster pipes for getting data into and out of an SoC.

The file model allows the chip designer to present interfaces for supporting SoC software development at varying levels of abstraction. This capability also allows for the concealment of proprietary details among on-chip core debug interfaces and to partition functionality across a system to manage complexity [99]. In SoCs containing cores with proprietary debug interfaces, the filesystem enables abstract namespaces allowing macro debug operations to be exposed without revealing how they are implemented. The knowledge of implementing the high level operations using the respective core debug interfaces rests within the filesystem. If conversely, the chip designer's objective was to limit the design complexity of the on-chip debug support structures, then the filesystem could expose an interface at a lower level of abstraction. In doing so functionality relating to use of core debug interfaces is transferred to external applications thereby reducing the filesystem complexity.

The filesystem model allows heterogeneity in debug interfaces among processor cores to be hidden behind filesystem abstractions. The interfaces across cores differ from one another a variety of

ways, ranging from signals in the physical interface to supported debug commands. By abstracting these heterogeneous interfaces using filesystems, uniform file operations may be used to interact with the processor cores. As with Nexus [20], standardizing the interaction between on-chip debug support and tools promotes interoperability between them. While Nexus implements this support fully in hardware, our work uses a hardware-software combination which makes it more easily extensible.

Distributed filesystem abstractions are able to naturally capture and utilize the hierarchical structure that is implicit in SoC based embedded systems. In such systems multiple IP cores are integrated into systems on chip, that are in turn used to build circuit boards, which are ultimately used to build systems. Analogously, filesystems exported by systems on chip serve as basic building blocks, and are progressively composed to create higher level filesystems at the board and system level as these entities is built up. The resulting hierarchy of filesystems provide mechanisms for debugging, tracing and monitoring the system at various levels in its hierarchy.

Composition of filesystems is performed using a hybrid of two broad approaches shown in Figure 3.3. The first is to organize the composed filesystem as simply a union of chip level filesystem namespaces as illustrated on the left in Figure 3.3. In this case the board level filesystem's primary purpose is to aggregate the different chip level filesystems (`/SOC1` and `/SOC2`) under one namespace and demultiplex incoming requests to the right one.

Alternatively the composed filesystem can be markedly different in functionality and namespace contents from the sub filesystems that it is built with. Using this approach, a *functional* file interface can be built at the board level using chip level filesystems that are primarily *structural*. Consider the example of a network router that is built using multiple SoC devices (such as network processors). Each of these SoC devices export filesystems that expose registers, memory and other hardware structures within internal processors as files. Using these chip level filesystems,

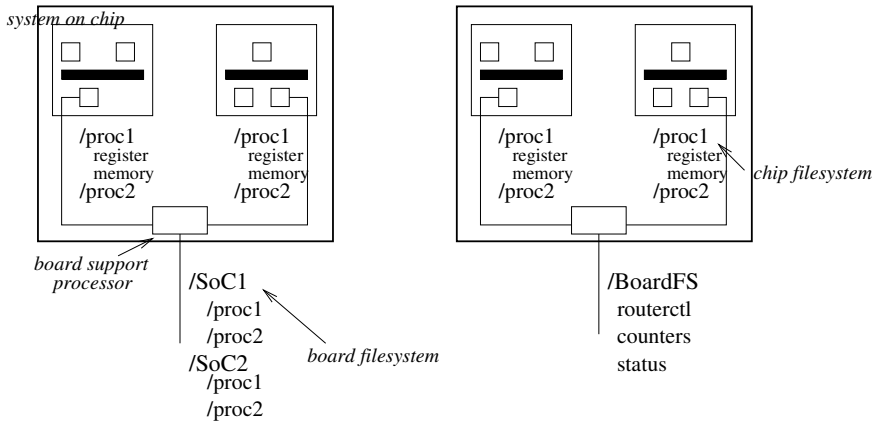

Figure 3.3: Strategies for composing filesystems

the router can implement a board level filesystem (such as `BoardFS` as shown on the right in Figure 3.3), which allows device level tasks - such as controlling router operation, managing routing information and accessing operational state - to be performed through the file interface.

A feature that distinguishes our model from Nexus is that ours combines the flexibility of a software based filesystem with performance and reusability of hardware components for interfacing with standard core interfaces. Industry standard interfaces defined for debugging [51] and tracing [62] ease integration of multi-core systems built using compliant cores, often with the use of hardware wrappers. By supporting these standards within the filesystem, possibly using hardware adaptors, a wide range of cores may be accomodated. Adaptors can also help offload implementation of debug operations from the support processor, by implementing them in hardware and thereby fulfilling a role analogous to debug coprocessors. Algorithms implemented in software on the filesystem can build on debug features provided by core interfaces hardware to create more complex features. An example of this is the use of instruction level single stepping feature provided by the cores to implement source level single stepping.

3.5 Implementation

This section discusses the implementation aspects of using hierarchical virtual filesystems to support software debugging and tracing in an SoC based embedded system. The architecture that we present uses filesystem building blocks presented in Chapter 2 to implement a hierarchical filesystem model that respects the relevant constraints and needs of the embedded domain, while enabling software debugging and tracing through a file based interface. we outline what these constraints and needs are and how our architecture is influenced by them.

Design objectives

In the workstation world, a distributed 9P filesystem is implemented by deploying 'standard' fileservers - either using operating system services or with standalone user level implementations - across the network constituting the distributed system. In the embedded domain however, this technique cannot be directly applied because of the resource constrained nature of the devices. Our primary design objective is adapt the distributed filesystem model for the embedded domain which implies recognizing and respecting the differences in computational capabilities of devices down the hierarchy of an embedded system. Correspondingly the various filesystems that execute at different hierarchy levels differ in sophistication.

The filesystem model in its basic form uses a request-response mechanism, wherein a client invokes a file operation (request) which is processed by a (possibly remote) filesystem, and the result of the operation sent back to the client as the response. The interactions of tools such as debuggers and trace port analyzers with an embedded device do not necessarily follow this paradigm. As an example a read on an *events* file to detect breakpoint or watchpoint hits may take an arbitrary time to complete depending on when the event of interest occurs. The filesystem cannot always afford

to block while waiting for these events to occur and this might lead to starvation of other incoming requests. The architecture needs to provide an acceptable means for handling such operations.

Another design objective is to use hardware to support filesystem implementation within the support processor. Use of hardware such as DMA can reduce the computational burden on the support processor while implementing the filesystem. Debug functionality implemented in software within the filesystem can be offloaded to hardware, as is often done with coprocessors. Using hardware adaptors , the support processor can expose standard interfaces [51, 62] to cores thereby facilitating their integration with the on-chip filesystem infrastructure.

An essential requirement on the workstation side is for tools such as debuggers and trace port analyzers to be able to use filesystem abstractions to interact with the corresponding embedded systems with 'acceptable' porting effort. While it is hard to generalize as to what constitutes being acceptable, the objective to is to avoid introducing low-level, filesystem related functionality within these tools, with the intention of making use of the filesystem approach minimally intrusive.

Architecture Layout

[CHG:embed-arch]The distributed filesystem architecture used to support embedded software development - shown in Figure 3.4 - uses a hierarchy of filesystems at chip and system levels that work together to support software development tasks for the system at large. The filesystems are based on the building blocks presented in the Core Technologies chapter 2 and vary in capabilities according to the level in the hierarchy they execute in.

At the core of the architecture is the embedded filesystem (EFS) that encapsulates an SoC device using a file interface. EFS is the filesystem in closest proximity to the hardware and exports a namespace that potentially supports debugging, tracing, configuration and monitoring of the

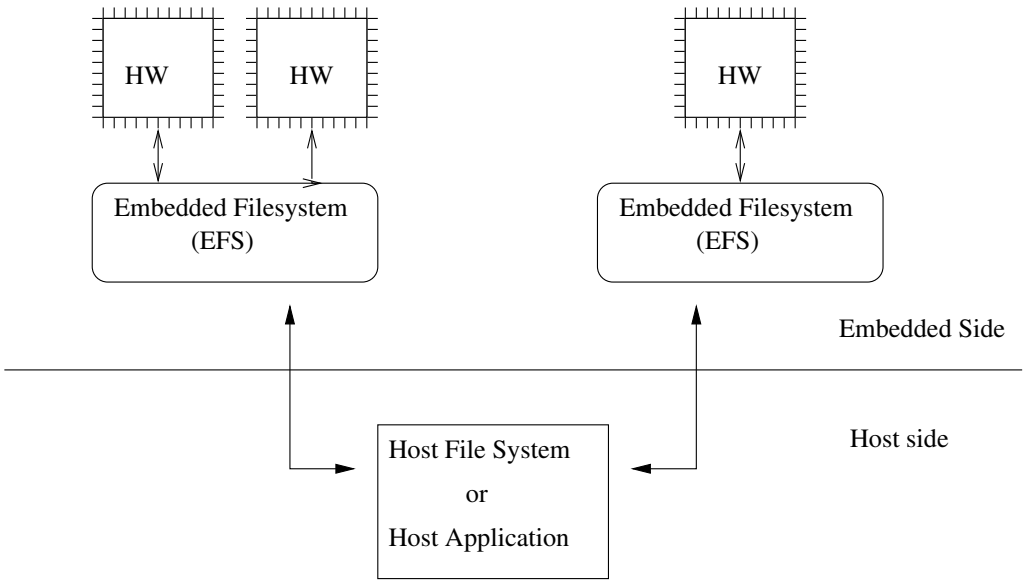

Figure 3.4: Architecture

processors within the SoC. The EFS is implemented in a designated processor core within the SoC known as the support processor, that may in addition be used for supporting the device's core functionality. In non SoC based systems, the EFS can be implemented within an equivalent basic design unit such as a circuit board.

Our implementation of the embedded filesystem is based on the EFS building block presented in Chapter 2. Each resource (cores) being abstracted has a corresponding directory within the EFS namespace. Processor resources provide files enabling workstations based tools to control their execution and to expose their internal state. Operations performed against these files are translated by the EFS into requisite operations on the debug interface corresponding to the targetted processor.

Multiple EFS instances at the SoC level are composed at the system level using mount filesystem blocks (MFS) presented in Chapter 2. In an SoC based system, EFS units may be exported by each of the SoCs, which are integrated by an MFS block implemented at the board level. The EFS and

MSF blocks in conjunction are naturally able to capture the hierarchical structure implicit in several embedded systems by representing them using a hierachical filesystem abstraction.

In our work, the EFS and MFS filesystems are mounted within the operating system namespace on the host side using the 9P to Fuse bridge described in Chapter 2. This enables debug and trace tools to control and access internal state of embedded devices through common file operations performed against the files in the global namespace. While existing implementations of the these tools would have to be modified to reflect this change, 9P protocol and transport layer specific details would not have to be introduced as they would be localized within the bridge. The modifications and additions are therefore clean and minimal.

Discussion

The implementation of debug and trace support using virtual filesystems exploits the thin server-rich client scenario found in systems targetted by our work. Servers execute on embedded adest-devices with constrained computational, memory and communication resources. Clients in the form of debugging, trace and programming tools execute on workstations which have more abundant resources. In recognition of this difference, server complexity is reduced by transferring resource intensive aspects of various operations to the client side.

An application of this strategy is in providing event monitoring capability through the filesystem. As described earlier, supporting event monitoring by having a read request block until the event occurs is acceptable only if other requests can continue to be processed in the meantime. This would however require use of either multi-threading to process each request individually or explicit storing and subsequent processing of the request, both of which consume memory . Our solution supports event monitoring through a two stage process, whereby clients first register for an event and then periodically poll an event file to check for its occurrence. This technique - based on

pull semantics - relieves the embedded server side of event notification responsibilities and instead transfers the responsibility of discovering about the event to the client side. The approach is used to support breakpoint functionality during software debugging as illustrated using a prototype presented in Section 3.6.

A key advantage of the filesystem approach is that each processor may independantly choose a desired level of abstraction corresponding to the file interface it implements. As described in the Application section **??**, chip providers can use this ability to choose the appropriate level of EFS abstraction in order to design for simplicity, for providing uniform debug interfaces among heterogeneous processors and to conceal processor debug interface usage information from external tools.

The 9P protocol used for providing RPC functionality within the filesystem framework is well suited for implementing the EFS. The maximum length of 9P messages is fixed at 8192, and can be reduced further by appropriate usage semantics such as restricting data count of write and read operations. The protocol supports multiple outstanding messages by its use of messages tags that enable requests to be matched to their corresponding responses. This allows multiple cores to be simultaneously debugged through the filesystem. Since the 9P protocol is transport independent, filesystems can be exported over a variety of physical links available to the SoC it resides in, including ethernet, USB or even on-chip JTAG scan chains used in conjunction with a higher level framing protocol like HDLC.

In any particular implementation, it is not necessary to include *all* the pieces in the system architecture. In simple systems requiring just a single EFS for instance, a useful improvization is to leave out the MFS block implementing filesystem abstractions at the system level. Instead, the EFS can be directly mounted on the host side and used by application software to interact with the embedded side.

3.6 Application to Debugging

In the next two sections, we use our filesystem infrastructure to perform embedded software debugging and tracing. We present prototypical applications that capture the salient features of the model and enable debug and trace operations through file interfaces. Debugging code on a processor requires mechanisms to access and modify registers and memory, to start and stop the processor, and to enable and disable breakpoints (watchpoints). Using a prototype, we illustrate use of the filesystem model to enable concurrent debugging in a heterogeneous multi-processor environment. Apart from demonstrating support for the mentioned "run control" capabilities through conventional file operations, objectives behind implementing the prototype are to illustrate other benefits associated with using the filesystem approach, namely:

- providing internal visibility through file interfaces
- creation of a logical interface that is independent of hardware layout
- presenting interfaces at different levels of abstraction to conceal proprietary debug interfaces and manage complexity across the system
- managing heterogeneity among on chip processor cores by means of uniform file interfaces
- use of a compositional file based interface to access various levels of the system hierarchy

Prototype Setup

Our prototype, shown in Figure 3.5, represents a heterogeneous, multi-processor embedded system for which a filesystem abstraction is implemented locally and exported over a communication link. The objective of the exercise is to use the filesystem abstraction to enable concurrent debugging of software executing on the various processing elements within the system. The system platform

consists of two dissimilar processors in the form of microcontrollers from Texas Instruments and Philips belonging respectively to the MSP430 and LPC product families; these devices are concurrently debugged on a workstation using the exported filesystem interface. The microcontrollers are based on processors with different architectures and debug interfaces. The LPC device is based on a 32-bit ARM core and offers an ARM-specific EmbeddedICE debug interface, while the MSP430 is based on a custom RISC architecture with a proprietary JTAG based debug interface. While the prototype is not a true system on chip environment, in being a heterogeneous, multi-processor environment it possesses the salient features of SoCs targetted in our work.

An embedded filesystem (EFS) encapsulating the two processors implements a filesystem abstraction for the prototype. The filesystem is implemented using a 32-bit soft processor from Altera named Nios residing within an FPGA. The EFS is exported using the 9P protocol over a serial connection, though our other prototypes have used radio and USB for exporting 9P based filesystems. On the host side debuggers for the two microcontrollers use the filesystem interface to debug software on the respective parts.

Filesystem Implementation

The filesystem implementation is based on the general technique for building EFS filesystems outlined in Section **??** of Chapter 2. The namespace exported by EFS to the host side is listed below:

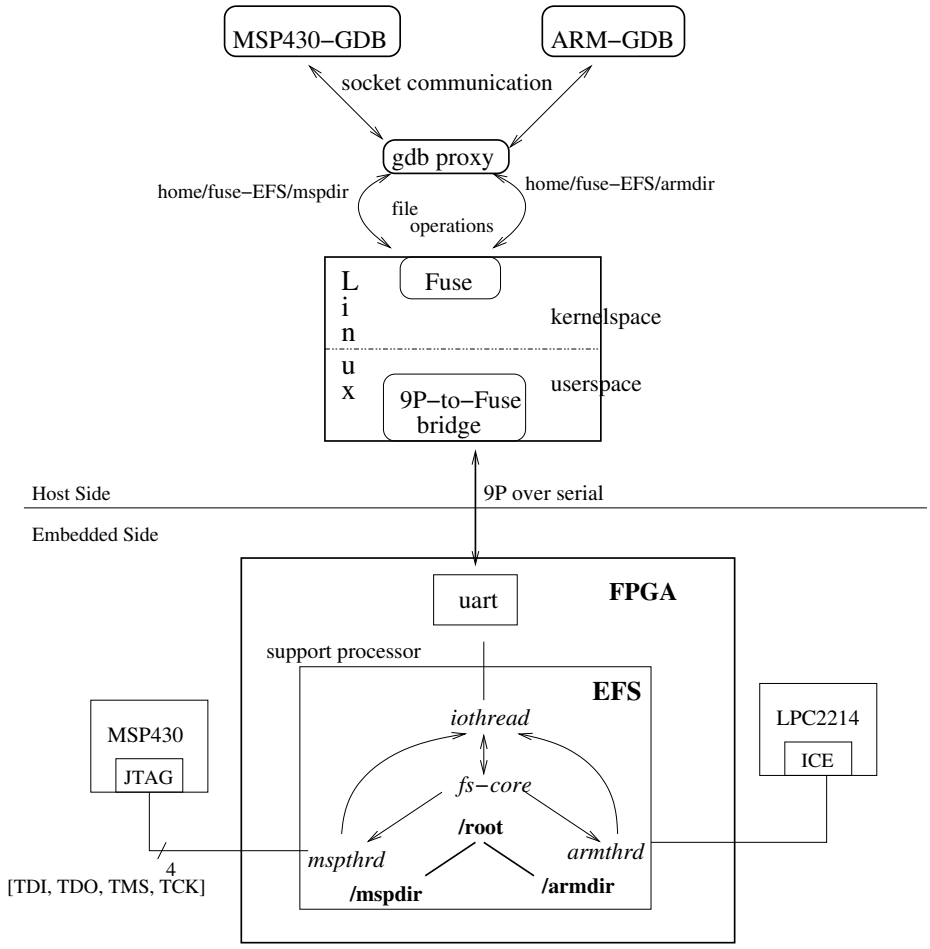

Figure 3.5: Prototype for Debugging

```
/EFS
  /mspdir
    registers
    memory
    status
    control
  /armdir
    registers
    memory
    status
    control
```

The filesystem captures the multiple heterogeneous debug interfaces among the two processes using identical file abstractions. The EFS namespace also illustrates use of file representations to compose individual device interfaces to create higher level ones at the system level. This is trivially done by organizing device directories for each processor as subdirectories within a common system level debug directory. The resulting system interface represented by the larger directory is isomorphic to the individual devices' interfaces represented by the two subdirectories. This isomorphism allows similar tools and techniques to be used to interact with various levels within the system hierarchy.

The EFS namespace is built by specifying directory layouts for each of the two devices using Dirtab structures as shown below:

```
const Dirtab armtab[]={
  {".",    Qtop,   Qtop,  PDIR | P_READ},
  {"registers", Qregisters,  Qtop,  P_RDWR},
  {"memory",    Qmemory, Qtop, P_RDWR},
  {"status",    Qstatus,  Qtop, P_RDWR},
  {"control",    Qcontrol, Qtop, P_RDWR}
};

const Dirtab msptab[]={
  {".",    Qtop,   Qtop,  PDIR | P_READ},
  {"registers", Qregisters,  Qtop,  P_RDWR},
  {"memory",    Qmemory, Qtop,  P_RDWR},
  {"status",    Qstatus,  Qtop,  P_READ},
  {"control",    Qcontrol, Qtop, P_WRITE}
```

```
};
```

Each line in the table defines a Dirtab element and represents one file/directory in the device directory. The entries list file name, unique file identifier, parent directory identifier and Unix style permissions which also specify whether the entry is a directory. *Qtop*, *Qregisters* etc. are unique integral identifiers for the various files in the table.

Our EFS implementation provides handlers defining the filesystem behaviour when supported file operations are performed against files in each device directory. The core filesystem handles the generic operations of attach, clone, clunk, and open, while device specific handlers are defined for read and write operations on MSP430 and LPC devices. Stackless pseudo threads are responsible for various aspects of the EFS implementation as given by I/O, buffer management, core filesystem operation and for serving device specific requests relating to the two devices within the system. The approximate size of machine code in the filesystem binary corresponding to these threads is listed below:

Thread	Code Segment Size(bytes)
Input/Output	1500
Buffer Mgmt	300
Core Filesystem Impl.	21000
LPC device	29000
MSP device	41000

The predominant contributors to the filesystem code size are the device threads. The rest of the implementation including core filesystem functionality, I/O and buffer management takes up about 25KB of code space, which is less than one third of the total size. This implies that by simplifying device thread logic it is possible to significantly reduce the filesystem size, thereby enabling

its implementation in devices with less memory.

On the workstation side the EFS is mounted within Linux by use of the 9P-to-Fuse bridge described in detail in Section 2.5 of Chapter 2. The bridge allows the remote EFS filesystem to be mounted locally within the Linux namespace using Fuse by forwarding all file operations on mounted filesystem as 9P requests to the EFS. As described earlier, techniques exist [90] [103] to similarly mount 9P filesystems within other popular operating systems such as FreeBSD and Windows. In our prototype, the mounted EFS namespace on the workstation side is used by debugger proxies to indirectly enable MSP and ARM debuggers to access their respective processors during software debugging. Proxies based on the filesystem model offer advantages over those implemented in traditional ways in terms of portability, reusability, support for multi-processor scenarios and concealment of embedded system design details as elaborated upon in the next section.

Debugger Hardware Access

In the embedded domain it is standard practice for debuggers to access embedded processors indirectly through debugging proxies. This allows the debugger implementation to be kept independent of the mechanism for accessing hardware. Debuggers interact with their respective proxies using a messaging protocol such as GDB's Remote Serial Protocol [**?**] or ARM's proprietary RDI protocol which capture the various debugger-processor interactions. Debuggers issue standard debugging requests which are then completed on their behalf by the proxies using debug interfaces available for accessing processor hardware.

The proxy model has two significant drawbacks when used with JTAG based processor debug interfaces. First proxies need to be aware of the usage of the available JTAG interfaces to perform various debug operations. To conceal implementation details, processor designers necessarily have to distribute debugger proxies as binaries and maintain confidentiality of the original source code

or run the risk of disclosing details of the processor's internal architecture. Another drawback is that since different devices (processors) typically provide dissimilar JTAG interfaces - in structure and or usage - it follows that a different proxy need to be used for each device, which makes the approach unwieldy for heterogeneous multi-processor systems.

In our prototype, a single proxy enables GDB based debuggers for both the LPC and MSP architectures to concurrently interact with their respective devices on the embedded side. Since the filesystem namespaces for both devices are similar in form and function, the same proxy implementation can be used to access either of them by simply changing the part of the filesystem namespace against which it operates. Consider the retrieval of register values from the target side, an operation requested by either of the debuggers and supported by the proxy. The proxy implements this functionality by means of a generic `register_read` function (partly shown in Listing 3.1 below) which when invoked with the prefix path to a device directory, retrieves register values from the appropriate device. The *prefix* variable specifies the device directory path while *registerindex* specifies the index of the register being read, which in turn in used as the offset of the read operation.

Listing 3.1: Implementing register read within debugger proxy

```
sprintf(registerfile_path, ''%s/registers'', prefix);
registersfd = open(registerfile_path, O_READ);
lseek(registersfd, registerindex, SEEK_SET);
retval = read(registersfid, registers_val, 8);
```

Use of a shared proxy allows hetergeneous multi processor systems to be debugged using a single communication link/port, unlike in traditional approaches where potentially multiple debugging ports (JTAG and BDM) may be required. Both the EFS filesystem and the 9P protocol

support multiple outstanding requests to be issued from the client side, which allows the debuggers to make requests concurrently without getting in each other's way.

Implementation of the proxy using the EFS interface is straightforward because of the natural mapping between request types in the GDB Remote Serial Protocol (RSP) and capabilities available through the EFS . Also, RSP uses the request-response mode of operation as does the filesystem. The proxy communicates with the LPC and MSP gdb debuggers through network sockets on which RSP messages and responses are exchanged. RSP messages supported in our filesystem proxy can be divided into four categories, namely: read/write of registers, read/write of memory controling and checking execution. They together provide debuggers with the necessary capabilities to debug software remotely executing on an embedded system.

Handling of register read requests was described earlier. The *register* file in each device directory when read at the offset corresponding to the index of the desired register returns the register contents in hexadecimal as a series of characters numbering twice the length of the register in bytes (since a byte takes 2 hex characters to be represented). Since the data is returned as text no byte order issues need to be addressed. Conversely, during register writes, the value being written is converted to a string and written to the register file at the appropriate offset, as shown in Listing 3.2 below:

Listing 3.2: Implementing register write within debugger proxy

```
int write_registers_totarget(uint32 writeval, int register_index) {
    sprintf(registerfile_path, ''%s/registers'', prefix);
    registersfd = open(registerfile_path, O_WRITE);
    lseek(registersfd, register_index, SEEK_SET);
    temp[0] = nibble_to_char((writeval & 0xF0000000) >> 28);
    temp[1] = nibble_to_char((writeval & 0xF000000) >> 24);
```

```
    temp[2] = nibble_to_char((writeval & 0xF00000) >> 20);
    temp[3] = nibble_to_char((writeval & 0xF0000) >> 16);
    temp[4] = nibble_to_char((writeval & 0xF000) >> 12);
    temp[5] = nibble_to_char((writeval & 0xF00) >> 8);
    temp[6] = nibble_to_char((writeval & 0xF0) >> 4);
    temp[7] = nibble_to_char(writeval & 0xF);
    write(registersfd, temp, 8);
}
```

Reads and writes to memory are handled similarly to the registers case. Proxies perform memory operations by reading or writing data to the memory file, using the offset to specify the memory address for the operation. With the EFS being implemented in a resource constrained processor, the filesystem write/read buffers are much smaller than the memories associated with each of the microcontrollers. The proxy explicitly ensures that the size of data being written/read is smaller than the EFS buffer sizes and chops up large sized operations into a series of smaller ones. While memories inside these microcontroller devices often have requirements on the alignment of addresses based on the size of data being written/read, the issued is handled locally by the EFS, thereby saving the proxy from having to deal with it.

Control operations supported by the proxy enable debuggers to start and continue execution, single step, and set/clear breakpoints. The operations are performed by writing control commands in the form of strings to the *control* file. For sake of simplicity, the commands supported by our EFS are a single character in length, listed as 'c', 's', 'b', 'r' refering respectively to the four operations mentioned above. Implementation of single stepping is shown in Listing 3.3.

Listing 3.3: Implementing single step within debugger proxy

```
  sprintf(controlfile_path, ''%s/control'', prefix);
  controlfd = open(controlfile_path, O_WRITE);
  retval = write(controlfd, ''s'', 1);
  if (retval < 1)
    return -1; /* unable to single step */
  while (!(ishalted()))
    usleep(100);
}
```

The device status execution is determined by reading the respective *status* files, which returns one of two single character status strings, namely : *h* or *r* indicating 'halted or 'running' states. Breakpoints use the previously described stategy of implementing event notification through synchronous means. After setting breakpoints by writing the [b]reak command to the *control* file along with an associated address, debuggers check for breakpoint hits by repeatedly reading the *status* file until the 'halted' state is indicated.

Partitioning Functionality

Unlike with JTAG based debugger proxies, the implementation of filesystems based proxies does not reveal information about hardware internals of the devices being debugged. The proxies can thus be distributed freely without fear of compromising proprietary architecture information. This property of the proxy is derived from the presence of a 'rich' EFS which exposes a high level functional namespace tailored to readily enable macro debug operations such as register access and control of execution. The 'intelligence' behind these operations is concealed within the EFS, which

has the result of increasing the complexity of the filesystem implementation as evidenced by the size of device-specific part of the filysystem binaries. Given the resource constained nature of embedded systems imposing this complexity may not be always acceptable. In order to manage complexity while implementing distributed filesystem abstractions, our model allows functionality to be partitioned among various levels of the system hierarchy. We illustrate use of this 'split filesystem' approach in the prototype to partition debug functionality between the host and embedded sides.

Being able to partition functionality is useful for reasons other than managing complexity as well. In SoCs containing cores with proprietary debug interfaces, the split file system approach can conceal details regarding use of these interfaces within the EFS implementation. It is possible to support time critical operations such as flash programming by implementing them within the EFS to meet latency requirements, as opposed to controlling them from the host side. Conversely, complexity of some operations may make them too heavyweight for being fully implemented within the EFS, thereby requiring them to be implemented on the host side. The 'split filesystem' approach offers the flexibility to expose interfaces at different levels of abstraction in order to address these issues.

As part of our prototype, we present three implementations of the MSP section of EFS, each representing a different partitioning of functionality between the host and embedded sides. These partitionings are based on software libraries available for debugging MSP430 devices using JTAG. The libraries may essentially be divided into two sets of routines. A set of high level routines (henceforth known as JEL for JTAG Emulation Library) is responsible for high level debug operations. JEL is built on top of a set of core functions (known as HIL for Hardware Interface Library) which provide the basic infrastructure for reading and writing JTAG registers and signals. The organization of these routines is illustrated in Figure 3.6.

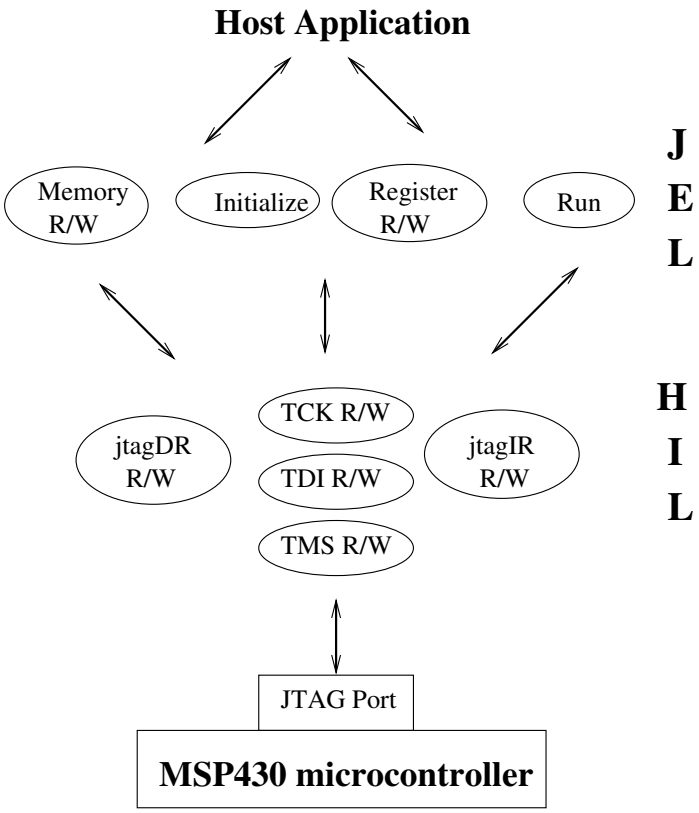

Figure 3.6: SW Routines for JTAG Emulation

Conventional debugging approaches used in practice implement both the JEL and HIL on the host-side with the HIL routines accessing the JTAG pins on the remote embedded device through a simple connector such as a parallel cable. This rigid partitioning necessarily confines all the debug functionality to the host which can be extremely limiting. Using the three implementations of the prototype we describe three contrasting partitionings of JEL and HIL between the EFS and host application, thereby demonstrating the superior flexibility offered by our approach.

EFS-centric partitioning: This type of partitioning is the basis of the filesystem implementation presented earlier in this section and implements both the JEL and HIL functionality within the EFS. The embedded side exports a rich file interface with files to directly control execution, access

registers/memory and check execution status. Such a partitioning conceals from the host details regarding the use of the in-system debug interfaces (JTAG in this case) by encapsulating the information within the EFS. Also, each high level operation (such as register read) involves a small file operations (reading the *registers* file) on the EFS. This results in minimal communication between the host application (such as debugger proxy) and the EFS thereby delivering good performance. However, the partitioning imposes considerable demands on the support processor, since the EFS contains a majority of the MSP device debug functionality. Using such a partitioning, size of MSP device logic with the EFS executable was about 41 KB, which (as shown shortly) is considerably bigger than the size for other cases.

Host-centric partitioning: A contrasting approach is to implement a lightweight EFS based on just the HIL that provides access to processor JTAG signals (TDI, TMS, TCK & TDO) and registers (JTAGDR & JTAGIR) through its namespace. The heavyweight JEL routines, implemented on the host side, access the MSP device JTAG interface through the EFS. The advantage of this approach is the limited complexity of the resulting EFS implementation. The size of the MSP part of EFS executable for this partitioning was less than 7KB, which is significantly less than the previous case. The disadvantage is that each high level operation (such as register read) involves numerous file operations on the EFS which compromises performance. Such a partitioning is best suited when dealing with lightweight embedded systems with limited computational and memory resources.

Hybrid Solutions: The two types of partitionings described represent the extremes where the debug functionality resides predominantly on either the embedded or the host sides. They reflect the tradeoff that exists between limiting EFS complexity and deriving acceptable performance. For most applications the optimal partitioning is a hybrid that lies somewhere between these extremes, that takes into account the requirements of the particular design. Using the hybrid model, select operations (such as flash programming) can be pushed down to the EFS to meet their specific needs

(such as timing requirements). An optimal set of operations would provide the greatest improvement in performance while imposing acceptable memory and computational requirements. The selection process can draw from the design of instruction sets for lightweight virtual machines [109] [79] and is in general guided based on frequency of use, latency for communication with host and availability of computational resources on the embedded side.

Optimizations

Splitting functionality between the host application and the EFS provides opportunities for optimizations, one of which we implemented on top of the host-centric partitioning model. It may be recalled that in this partitioning the EFS exports a basic filesystem that provides access to JTAG signals and registers through its namespace. The most common operation performed on host involves writing values to the two JTAG registers, namely JTAG Data Register (JTAGDR) and Instruction Register (JTAGIR) through the EFS. Each time a value is written to one of these registers by shifting it in through the scan chain, a value is simultaneously shifted out, which can be thought of as the return value of the operation. However the value that is shifted out is rarely used. A check of the JEL reveals that the value shifted out is used about 15% of the times, which means in most cases it can be discarded.

Based on the relative infrequency with which the shifted out value is used, we designed an optimization whereby individual writes from the host side to these registers are buffered up by the GDB proxy until an operation required the shifted out value. Then all the buffered write operations are sent together as one request to the EFS which unrolls it and executes the writes serially. The values being shifted out on all except the last write operation are repeatedly overwritten by subsequent writes. Once the writes complete, the value shifted out in the last write is read back through the EFS. Implementation of the optimization resulted in a five fold improvement in performance for memory read and write operations as compared host-centric partitioning approach.

The namespace exported by the EFS when using this partitioning changes as follows:

```
/EFS
    /mspdir
            cmd
            jtagDR
            jtagIR
    /armdir
      ......
```

Write operations are buffered by the proxy and ultimately written to the *cmd* file. The *jtagDR* and *jtagIR* files may then be read to retrieve values shifted out in the last write operation to the respective registers in the sequence.

Performance

We compared the performance of the prototype using EFS-centric partitioning with the standard approach used for MSP430 debugging based on host resident software communicating with the device JTAG interface through a parallel connection. The table below shows the latencies for memory access, single stepping and the observed rate of data writes to flash for the two cases.

	Our Prototype	Std. Approach
Memory Access	1 millisec.	7 millisec.
Single Step	750 millisec.	770 millisec.
Load code to flash	1330 bits/sec	4110 bits/sec

Our approach performs better during memory access and single stepping than the standard approach. For writing to flash it is slower by about a factor of 3. we suspect that the cause for this inferior performance is the algorithm that GDB uses when writing an executable to flash, whereby

it chops up the entire code into parts and loads these one after the other resulting in increased file operations. This can be remedied by implementing loading strategies with better loading strategies.

3.7 Application to Tracing

A second application of the filesystem infrastructure in the context of embedded software development is to support in-system tracing. Tracing is the process of generating traces of relevant runtime characteristics of a system over a period of execution. Traces provide a means to analyze system execution in a minimally intrusive manner. Tracing is especially important in the embedded domain where use of conventional debugging techniques (such as breakpoints) may be unsuitable as systems operate with realtime constraints and in environs where operation of surrounding entities cannot be finely controlled [23].

Many popular embedded architectures including those from Motorola [8] and ARM [2] provide support for tracing executed instructions and memory data accesses (respectively known as instruction and data traces). Triggers define the circumstances when tracing is enabled; they are specified typically based on memory access within address ranges, execution of specific opcodes and processor exceptions. In addition to triggers, trace infrastructure in processors often includes filters that enable the user to specify the data that needs to be collected after triggering. Configuration of trace triggers and filters is typically done through JTAG based mechanisms. Traced data is either streamed 'live' off chip through trace ports or stored locally in trace buffers and retrieved later. When trace data is stored locally, the same physical connection used to configure the trace is reused to retrieve trace data as well, while with streamed traces a separate higher bandwidth link is used.

The problem with this approach is that it scales poorly for heterogeneous, multi-processor SoC

devices. The trace trigger, filter and buffer in combination are often implemented within a dedicated trace module. In a multi-processor scenario each processor requires a unique trace module with its own interface and trace buffer. Further, as with debug interfaces considerable differences exist with respect to mechanisms and protocols for controlling and accessing the trace data generated among heterogeneous cores, thereby limiting the portability of trace tools. The goal of this work is to implement a system-wide trace interface that allows configuration and access of multiple, heterogeneous processor traces through uniform file operations, while sharing the trace buffer and interface among them.

While our implementation focusses on instruction level tracing, other forms of tracing are actively used in the embedded domain as well. Instruction level traces have the disadvantage of being potentially too low level, thereby rendering considerable amount of data which makes it hard to identify high level events within them and to correlate across traces from multiple sources. An alternative is to use traces of system level events such as context switches, bus operations etc., which present data associated with system operation at a higher level of granularity. These traces are particularly useful in multi-processor environments [122] for resolving concurrency related issues such as lock contention, for verifying operational correctness and for performance tuning. A technique used to generate system level event traces is through instrumentation of application level (userspace) or operating system software [119]. Software based trace generation may impose unacceptable delays for realtime systems, in which case a 'hybrid' scheme [55] using hardware for supporting tracing and monitoring may be adopted. The presented model for instruction level tracing can be applied to system level event tracing as well, wherein a common file based interface is used to configure and access multiple trace resources which generate event data collected in shared memory buffers.

It is our objective to demonstrate how the filesystem model can support tracing within a multi-processor SoC environment. we focus on supporting two core tasks, namely: trace configuration and access of generated trace data generated, both enabled through the filesystem framework. we show how the embedded filesystem implementation can be designed to support trace streams from multiple processor sources while sharing resources such as trace buffers, configuration interfaces and data ports among them. We shall also illustrate how the model enables system level debug features to be built using those provided by the individual processor cores. The ideas are presented using a prototype consisting of a multi-processor system from within which multiple trace streams are exported and configured through a unified file interface.

Prototype

The trace prototype models a dual processor SoC within which a filesystem abstraction is implemented and exported as shown in Figure 3.7. Each of the processors in the system are of the 32-bit ARM7 core family and are implemented using the Armulator instruction set simulator [4]. Choice of simulated processors was based on the complexity of support hardware required for using trace on real ARM cores. Two instances of the simulator execute as separate processes, representing the cores within an actual system. The embedded filesystem implementation executes as a third process and would be implemented within a support processor in a 'real' implementation as is customary in our model.

[CHG:armul]The ARM simulator was modified to generate a trace stream by outputting address of instructions as they are executed. A combination of the use of simulator based trace sources and our concious decision to keep the prototype simple implied that certain intricacies associated with conventional trace solutions are not addressed as part of our work. In order to limit the amount of generated trace data, conventional trace solutions include only indirect branches and

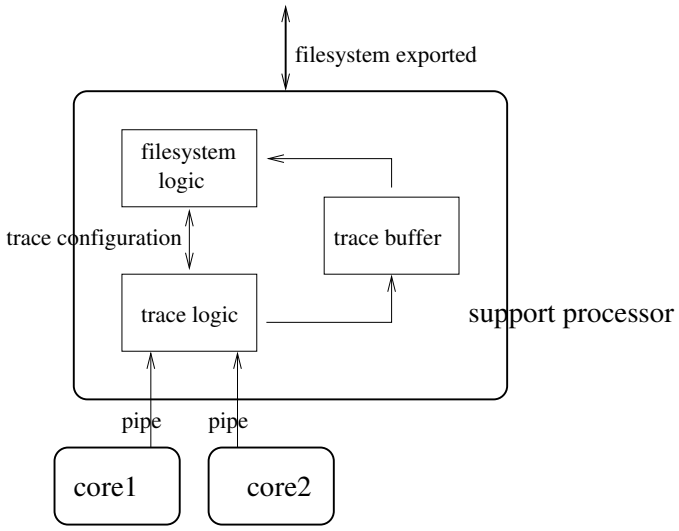

Figure 3.7: Prototype for Tracing

indirect calls in the outputted trace rather than the full address of every executed instruction, with linear execution being assumed otherwise. While our prototype currently traces out every executed instruction, the filesystem architecture can be extended to output only the required instructions during tracing by implementing the necessary functionality within the support processor trace logic. Certain trace solutions enable timing characterstics to be deduced using trace output. ARM ETM offers this capability by using a realtime pipeline status port (PIPESTAT) in addition to a higher latency trace data port. By corelating the two, trace analyzers are able to deduce the exact timing information associated with code execution. Use of the instrumented Armulator tool implies that cycle accurate pipeline information is unavailable to our filesystem block. Therefore no timing information is presented as part of our trace.

The transfer of trace data between the filesystem and each instance of the simulator occurs over a pipe. In real systems, dedicated point-to-point signals are used to provide trace blocks with access to processor execution information. The filesystem implementation in our prototype is based on

our EFS block, and in addition contains trace logic that maintains configurable triggers and a trace buffer to store the incoming trace data. The trace logic and filesystem core are implemented as separate threads within the filesystem process that access a shared trace buffer implemented as a large array. Upon receiving trace data from any of the in-system processors through their respective pipes (monitored through `select` operations), if the data conforms to any of the active triggers then the trace logic adds it to the trace buffer along with the associated metadata. In this work, the metadata consists of two pieces of information that includes: whether the data is a continuation of an earlier sequence of instructions or start of a new instruction sequence, and the identifier corresponding to the core to which the trace data belongs. Trace data is accessed by reading files in the exported namespace, in which event it is retrieved from the trace buffer by the filesystem core and returned to the user. Thus, a typical producer-consumer situation exists between the trace logic and the filesystem core.

In an actual SoC, the filesystem would be implemented within a support processor inside the SoC. Triggers are implemented using dedicated hardware blocks such as Coresight Embedded Cross Trigger [3], which apart from providing single processor triggers can also facilitate system level tracing through support for cross triggering. The support processor can memory map trace logic hardware registers within its address space, which would then allow triggers to be defined by setting these registers appropriately.

The filesystem configures the trace logic block by having the support processor memory map trace logic hardware registers within its address space; triggers are defined by setting these registers appropriately. Trace buffers can be implemented using memory blocks within the SoC, which is mutually accessed by both the support processor and trace logic.

The EFS consists of a single instance of a device named *tracedev* that is responsible for implementing file abstractions for the trace interface. The namespace supported by the *tracedev* device is

listed below:

```
/traceFS
    ctl
    tracedata
```

The *ctl* file is used to control execution of processors within the system as well as to configure the trace triggers. Triggers are defined based on address ranges which when executed from within cause collection of trace data. The ranges are specified by writing the 'trigger' command to the *ctl* file along with a processor number and an address range as shown below:

```
echo 'trigger 1 8100 810c' > ctl
```

This triggers tracing for processor 1 whenever it executes an instruction from within memory location address 0x8100 and 0x810c. Multiple such triggers may be defined for each of the processors within the system.

An advantage of the file based approach is that the filesystem may be used to implement system level debugging operations in multi-core SoCs on top of features provided by individual cores. An example of this as described earlier is cross triggering [3], which initiates tracing of data from multiple on-chip processors when a designated processor triggers. Our prototype supports cross triggers, which can be set by writing the 'crosstrigger' command to the *ctl* file along with the processor number and an address range. When any of the cross triggers fires it initiates tracing among all processors within the system.

Trace data may be retrieved through the file interface by reading the *tracedata* file. The data is formatted in the form of a series of trace packets each corresponding to a single instruction executed by one of the processors. In the event of multiple processors generating traces, the data

read out may be multiplexed with packets from different processors. The trace packet has a highly simplified format shown in Figure 3.8.

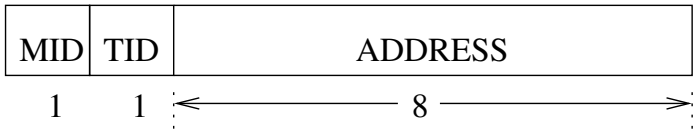

Figure 3.8: Trace Packet Format

Packets have a fixed length of 10 bytes, which makes packet processing easier [62]. Variable length trace formats such as those used with Nexus [20], while being more complicated, provide better compression. The Trace source IDentification `(TID)` field identifies the processor that generated the trace packet within the multi processor system. The Message IDentification `(MID)` field identifies the packet as being either a continuation of an earlier sequence of instructions (denoted by 'C') or start of a new instruction sequence (denoted by 'N'). As code is executed, once a trigger fires it generates an 'N' type packet denoting start of a new sequence followed by a series of 'C' packets corresponding to the stretch of execution where the trigger condition holds. The `ADDRESS` field indicates memory address from which the instruction was executed in that cycle. Tracing stops when execution moves onto a section of code that is not being traced and the whole process repeats itself for another period of code tracing. Listing below shows traces being configured and read for the two processors in the prototype using the filesystem interface. Trace data read off the *tracedata* file shows traces for processor '1' over two periods multiplexed with processor '2' trace data. The : and , punctuations have been added for readability.

```
> echo 'trigger 1 8100 810c' > ctl
> echo 'trigger 1 8208 821c' > ctl
> echo 'trigger 2 8100 810c' > ctl
> echo 'go' > ctl
```

```
> cat tracedata
N1:00008100,C1:00008104,C1:00008108,N2:00008100,C2:00008104,C1:0000810c
C2:00008108,N1:00008208,C1:0000820c,C1:00008210,C1:00008214
```

Discussion

The prototype illustrates three aspects relevant to using filesystem interfaces for tracing. First, it shows how multiple trace modules may be configured and accessed using a common file based interface that is exported over a single, generic communication link. The prototype also uses a shared trace buffer to store trace data from multiple sources. Apart from sharing the interface and storage buffer, our model offers the opportunity to share other resources such as trace compression blocks across processor trace modules which makes it particularly scalable to multiprocessor SoC devices. The second aspect illustrated by the prototype is the use of high level file operations for trace configuration and access which makes the filesystem based approach suitable for heterogeneous processor cases, since the differences in mechanisms for controlling and accessing the trace are concealed behing the filesystem implementation. Finally, the *crosstrigger* feature provides an example of the use of on-chip filesystem to implement system level debug features on top of features provided by individual cores.

Support hardware facilitates the implementation of an embedded filesystem that supports tracing. Use of DMA-like components would enable the support processor to simply setup streaming of trace data from internal trace buffers out through available communication peripherals when the data is read externally. To enable compression of traces before they are streamed out, the support processor may be supplemented with hardware based trace compression blocks [89]; these may acheive compression through standard approaches such as differential compression [47] [104], or through architecture based approaches such as value prediction [87]. To facilitate integration of the

filesystem trace logic with on-chip processor trace modules, standard interfaces for tracing [62] [20] could be supported. In this regard, McDonald-Maiser et al. [62] propose a trace infrastructure that includes a standardized trace interface (including signals and messaging protocol for trace data) between processor cores and the rest of the debug support; this allows arbitrary on-chip interconnects to be integrated with the trace infrastructure through use of suitable wrappers.

3.8 Related Projects

Various projects have tried to adapt JTAG for use in an SoC environment. The most basic challenge is to make an SoC with multiple cores - each with its own JTAG infrastructure - operate as a compliant JTAG chip, which implies among other things using no additional JTAG signals. Oakland[49] implemented compliant means to access debug registers within various cores of an SoC by introducing an additional chip-level instruction register R0, that was a part of the scan chain. By loading specific *access codes* to IR0 external tools could explicitly control the chip JTAG behavior without the need for additional external signals. Vermuelen et al.[26] proposed a compliant core access technique of their own that additionally permitted concurrent debugging. Their approach is based on a two level JTAG module setup, one at the chip level that is daisy chained to those at the core level. JTAG compliance is achieved by handling bypass operation as a special case. While these approaches succeed in making JTAG better suited for the SoC domain, they suffer from the basic deficiencies associated with JTAG of not being scalable or portable and exposing proprietary knowledge about core debug interfaces[54].

The Nexus initiative[20] addresses the issue of portability in JTAG. It defines a uniform debug interface that compliant tools and processors adhere to, so that they can be guaranteed to work together. The physical debug port of the processors is standardized as are the interactions that take place over it. A drawback of Nexus is that since the set of interactions are fixed in hardware, it cannot be easily customized or extended to support other software development operations than the ones implicitly supported. Our model in contrast uses a software based filesystem whose functionality can be reconfigured, extended and customized more easily. This flexibility of a software based filesystem can be combined with the performance and reusability of hardware based debug adaptors to facilitate integration with existing debug and trace interfaces. Also, just as with JTAG, Nexus uses a dedicated debug port to which physical access has to be provided externally. In

our model, debugging is enabled through interfaces exported over generic communication links as available within the system.

While Nexus standardizes interactions between tools and debuggers, another line of work has looked at introducing uniformity in the interfaces exposed by the cores to the rest of the SoC, in order to promote core reuse. The IEEE P1500 standard[51] defines a methodology for equipping cores with necessary hardware so that SoCs using them can be tested in a JTAG compliant manner. Hokpins and McDonald-Muier[62] propose core interfaces that standardize configuration and access of trace data in multi-core environments. Significantly, their work is not scan-chain based and aims to decouple the SoC debug support infrastructure from the debug interfaces provided by the individual cores. The VSIA alliance[86] addresses the larger issue of defining standards for various aspects of system chip development including testing, verification, bus protocols, analog-mixed design etc.

An alternative option while debugging system on chips is to use simulation. The key advantage of using simulation is that systems can be debugged and tested for functionality, performance and power without having access to hardware. Simulation also allows obscure internal state to be accessed far more easily than in real systems. System on chip simulations are based on simulation models [16, 4] of constituent processor cores along with those for memories, buses and peripherals. These models are integrated and executed in system simulation environments [34, 115] while examining operational characteristics. The utility of simulation though, is limited to cases where operational conditions are reproducible at the time of running the simulation.

[CHG:verification]As Dijkstra pointed out[40], software testing and debugging can help weed out bugs existing in programs, but can never be used to prove their absence. The need for software debugging can be obviated by use of verification techniques to write provably correct software. However generating models that accurately model complex software systems is hard. Further,

with the shortening times to market associated with embedded products today accomodating the additional verification step in the software development process might not be feasible or cost effective. While formal methods are an effective means of reasoning about system operation at the algorithmic level, use of debugging techniques in conjunction is required to enable correct implementation of these algorithms.

Commercials vendors in both hardware and software domain offer tools to support system on chip software development. ARM, a leading processor IP provider, offers processors with in-core support for runtime debugging through a JTAG based interface known as EmbeddedICE. Select cores provide a trace interface called ETM, that streams trace data using a dedicated trace port while being configured through EmbeddedICE. ARM's Coresight technology[3] provides system level debug features such as cross-triggering and sharing of trace port across multiple cores by leveraging debug support on individual cores. ARM's flagship debugger named RealView supports debug of systems with multiple heterogeneous core combinations such as an ARM based RISC core and a DSP cores. Software executing on the different processors can be simultaneously debugged allowing for synchronized start/stop and single step operations. The debugger can use chip cross trigger capabilities to enable breakpoint events in one core to cause multiple cores to break execution. Technologies like Coresight form ideal building blocks for implementing ideas presented in this chapter.

3.9 Summary

This chapter described use of the virtual filesystem technology to support debugging and tracing of software executing within multi-processor embedded systems. we present a technique for implementation of filesystem abstractions within SoC based embedded systems using building

blocks described in Chapter 2. These abstractions capture debug interfaces of the various (potentially) heterogeneous processors within the system using uniform file namespaces. Motivations for use of our approach are presented vis-a-vis limitations of existing techniques, such as the debug interface being closely tied to hardware characteristics of the device. Advantages are illustrated by means of prototypes for debugging and tracing that capture the salient aspects of using filesystems to support software development for SoC based systems . The debugging prototype consists of two hetergeneous processors in the form of ARM and MSP430 microcontrollers for whom a filesystem interface is implemented locally and exported over a communication link; software on these devices is concurrently debugged on a workstation using the exported filesystem interface. Apart from demonstrating support for basic "run control" capabilities through conventional file operations, the prototype illustrates other benefits of this model including the ability to partition debug functionality at different levels of the system hierarchy, being able to support debug interfaces at different levels of abstraction and the management of heterogeneity among various on chip processor cores. The prototype for tracing demonstrates use of our filesystem model to implement a system-wide trace interface that allows configuration and access of multiple processor traces through uniform file operations. Since the on-chip resources for implementing tracing, such as the trace buffer and trace interface are shared among the processors, the approach naturally scales to support multi-processor systems.

4

Application to Sensor Networks

In this chapter we describe use of our virtual filesystem framework for use and deployment of sensor networks. Sensor networks are distributed collections of heterogeneous, resource constrained sensor nodes used for monitoring and environmental data collection over large geographical areas. Availability of lightweight abstractions for sensor networks allows applications to be developed unmindful of low level sensor issues. we present a distributed virtual filesystem based abstraction that provides a common, scalable and high level interface for enabling various sensor related operations.

4.1 Introduction

Inexpensive low-power processors and wireless transceivers have made creation of widespread networks of sensor nodes possible. These networks have been used for data acquisition and monitoring purposes in diverse application domains including ecology, security and structural engineering as a cost-effective means of observing large areas[22, 76, 12]. Our objective is to design a distributed filesystem architecture for sensor networks that caters to requirements associated with the network use and deployment while being compatible with the structural characteristics of the devices within the network. Using the filesystem framework, we focus on facilitating two application scenarios in which sensor networks are commonly used, namely - sensing and recording information relating to ambient conditions and event detection and notification for monitoring purposes. In recognition of the unequal computational capabilities of devices within the network, the filesystems distributed across the network vary in sophistication as do the protocols used to interact among them.

In addition to its use during network operation, the virtual filesystem infrastructure can be

reused during configuration and deployment phases. Heterogeneity among sensors within a network requires software to be manually configured to suit sensor architectures before compilation, and resulting binaries to be deployed using programming mechanisms appropriate to the various sensors again chosen manually. Using our approach in contrast, virtual filesystems implemented within the network enable extraction of sensor architecture information using file operations. Similarly sensors can be programmed by 'copying' binaries to their file abstractions, thereby saving the end user from having to keep track of intricacies and variations in programming techniques.

On the workstation side, we provide mechanisms to mount sensor filesystems within namespaces of popular desktop operating systems such as Linux and Windows. This opens up opportunities to use conventional tools and techniques such as GUIs, spreadsheets, and project configuration tools such as 'make' with sensor networks without having to implement overlying middleware between the tools and the network. Researchers using sensor networks who have limited expertise in computer science would especially benefit from being able to access the network through conventional tools described above.

The chapter is organized as follows. we start by discussing the significant structural and functional aspects of sensor networks that influence our work. we then introduce the idea of using file representations with sensor networks and describe how the abstraction facilitates network usage and deployment in our model. Our implementation for realizing distributed filesystem abstraction within sensor network architectures is then presented. This is followed by an overview of related sensor middleware techniques that share similar objectives to our work. we conclude by presenting a basic evaluation of the implementation.

4.2 Sensor Networks Background

The large numbers in which sensor nodes within networks are deployed requires the cost of individual sensor nodes to be economized. Sensor node cost is controlled by optimizing the set of included resources, and even the resources that are present are minimalistic. Typical sensor platforms from popular vendors such as Crossbow[126] and Moteiv(telos), for instance, use processors operating at clocks speeds less than 20MHz with a few 10s of KB of available RAM. In addition to the processor, a sensor node contains a set of sensors (probably as a sensor array) as required by the application, signal processing hardware and a wireless transceiver with limited range.

Our work is based on the cluster based model[59] where the entire sensor network is divided into distinct clusters of sensor nodes. Each cluster is managed by a cluster head node, which has superior computational and networking resources as compared to lightweight sensors constituting the cluster. A cluster head class device from Crossbow known as Stargate, used with the sensor nodes described earlier is based on a 400 MHz Intel XScale processor that has 64MB of RAM at its disposal. Cluster based sensor networks use software at the sensor and cluster head levels which work in conjunction with data analysis, network monitoring and configuration tools at the workstation level to provide network functionality. Software at each level has to respect the computational and memory constraints relevant to the entity it is executing on.

Two scenarios in which sensor networks are commonly used are for passively recording information relating to ambient conditions relating to soil, atmosphere etc., and for actively monitoring wide areas for surveillance, operational correctness, and disaster detection purposes, sometimes deployed in conjunction with a respond system. These usages may respectively be categorized as relating to data-centric and event based applications. Data-centric applications use the network as a data source from which sensor readings are retrieved either individually or aggregately by

workstation based applications. In event based applications, monitoring tools executing on workstations define events of interest within the network based on sensor readings, and get notified as and when the events occur. An example of such an application is the use of widely deployed temperature sensors to detect forest fires[24].

Sensor networks posses certain defining architectural and usage characteristics that make them different from conventional distributed computational systems. The networks typically comprise a diverse set of hardware and software elements. Hardware relating to sensing , communication and supporting processor types range from commercial-off-the-shelf (COTS) components to highly-specialized, one-of-a-kind parts. Software elements draw from numerous disciplines, including embedded systems, artificial intelligence and various natural sciences reflecting diverse application domains that these networks are used in. To cater to the large number of scientists and researchers using sensor networks whose expertise is not computer science, familiar interfaces such as those based on databases[128] and spreadsheets[125] have been proposed and implemented for using the network.

The resource constrained nature of sensor nodes introduces idiosyncrasies unique to their operation, in the form of short duty cycles to conserve power, unreliable communication links and operational limitations. Use of abstractions such as those presented as part this work saves user level applications from consciously having to deal with these low level sensor issues. Since the sensor nodes are spread over large geographical areas, issues synonymous with distributed systems such as naming and discovery have to be addressed, in a resource constrained environment lacking standard capabilities such as TCP-IP based end to end networking.

The predominant practice used in deploying sensor networks is to first program sensors and then release them into their field of operation. With increased programming and access speeds, low power consumption and high reliability, flash memory based microcontrollers have come to

be used extensively with popular sensor platforms from Crossbow(mica), Moteiv(telos) and Intel(mote). Programming these sensor nodes involves writing to the on-chip flash memory of the microcontrollers through either hardware based (JTAG), or software based mechanisms. Telos sensors for instance, are based on the MSP430 device from Texas Instruments, which ships with a boot strap loader software[7] that can be communicated with over serial or USB links to program flash as shown in Figure 4.1. The programming process is thus dependent on the hardware associated with the sensor node, communication link and memory write mechanism being used.

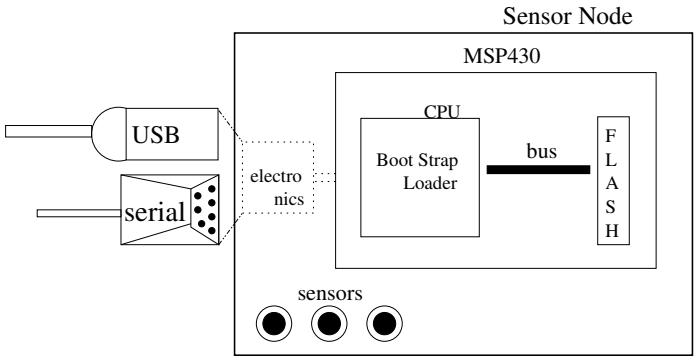

Figure 4.1: Flash Programming through Bootloader

Network programming saves the need of having to program each sensor individually by enabling them to be concurrently programmed over radio. The XNP[66] and its related Deluge[63] project support network programming of TinyOS based sensor nodes. The basic principle involved is similar to that used in flash programming over serial or USB links, with the difference that a radio link is used to to communicate with the resident flash loader. Network programming has the advantage of supporting heterogeneous sensor nodes since a common radio communication based protocol is used with all of them. Its limitation is that at least a part of all nodes to be programmed need to within radio range of the programming source. Our work uses network programming as an enabling technology on top of which richer functionality is built.

The deployment techniques widely used today are adequate to program local sets of sensors. More complex networks are built on a larger scale[70] and consist of heterogeneous sensors grouped into clusters distributed over wide areas, where manual configuration and deployment would be impractical[37]. In such scenarios remote deployment of applications on sensors is a necessity, complete knowledge of sensor architectures may not be known apriori, and different deployment techniques might be required among the sensors. Our objective is to support these non-trivial sensor deployment requirements through the virtual filesystem framework by leveraging existing deployment techniques.

4.3 Filesystem Representations for Sensor Networks

Consider the sample network depicted in Figure 4.2, used for soil monitoring. Sensors spread over a vast geographic area help monitor characteristics like moisture content, chemical concentration (contamination) etc. The network consists of two clusters, each managed by a cluster head. The cluster head communicates with sensors using short range, low power radio often based on the 802.15.4[65] standard, and with an upstream network using a sophisticated communication mechanism such as ethernet.

Such a network may be abstracted using a virtual file system with a namespace as shown in Figure 4.3. This filesystem is implemented in a piecewise manner with each cluster contributing a section that is exported over communication infrastructure available to the respective cluster head. The various pieces are assembled on remote workstations which then gets a composite file based interface to access the entire network.

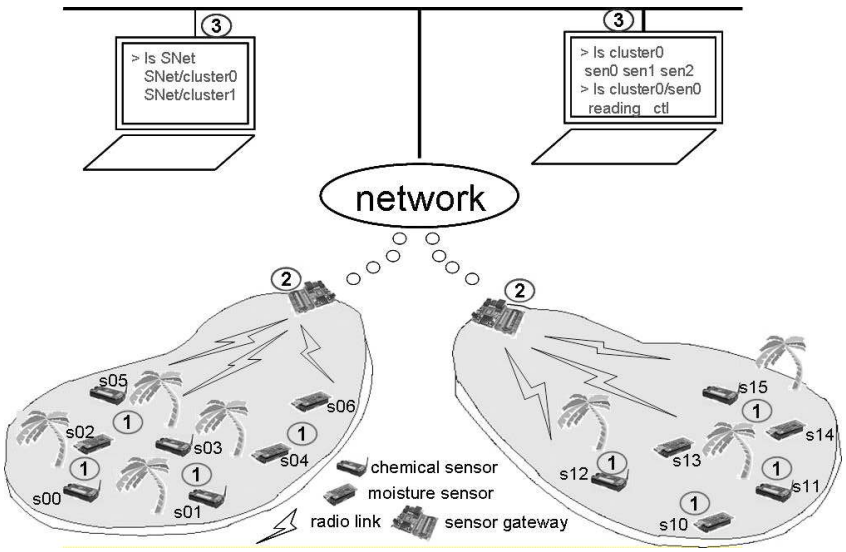

Figure 4.2: Cluster Based Sensor Net.

Each cluster is represented by a top level 'cluster directory' containing subdirectories for its constituent sensors. These 'sensor directories' contain files for accessing sensor readings, verifying status of node operation, and configuration and delivery of sensor events, as given by the *reading*, *event* and *status* files in the example. Once this basic namespace is in place, data centric operations can be implemented on workstations solely using file operations. This idea is illustrated in the script shown below that logs readings from all sensor nodes that are 'ON', by iterating through each sensor directory.

```
foreach sensor (s*)
  if ((cat $d/status | grep "ON" | wc -w) <> 0)
    cat $sensor/reading >> log
  endif
end
```

The resulting code is elegant and intuitive since it uses the familiar file metaphor while leaving

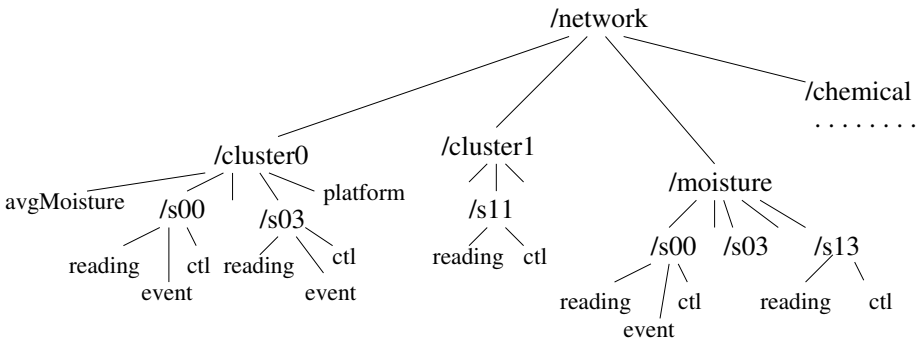

Figure 4.3: Filesystem Namespace

out low level sensor-specific details. Furthermore, the script operates on files without any realization that their data is generated on the fly.

Filesystems are hierarchical and hence share a structural correspondence with sensor networks, which makes the use of their representation natural. File abstractions exported by the various clusters within a network are integrated in the creation of abstractions encapsulating the entire network by mounting individual cluster filesystems under common directories. Use of file abstractions reduces the task of locating and naming a sensor device to finding the path for its corresponding file in the namespace. Sensor 1's value from cluster 0 in the example, for instance, is read from `/network/cluster0/s1/reading`. The uniform file interface abstracts heterogeneity among sensors within the network. The networking and protocol details relevant to access of sensors from remote workstations is localized within the filesystem implementation and concealed from the user. Since the entire filesystem is implemented in a distributed manner using software at the workstation, cluster head and sensor levels, the complexity at each layer can be managed by redistributing it among higher layers in the filesystem hierarchy. This idea is embodied in the implementation of event support within the filesystem infrastructure as explained later. Individual sensors provide

basic single event support based on sensor reading thresholds, which is built upon at the cluster head level in allowing multiple clients to each define distinct events that all are concurrently supported.

In our work network file representations are implementated using a two level filesystem hierarchy executing within the network at the cluster head and sensor levels respectively. Filesystems within the sensors nodes (henceforth called sensor filesystems) provide access to the minimal data and event resources available within the node, which is built upon by the overlying filesystem within the cluster head (henceforth called cluster filesystem) to provide robust file abstractions for the entire cluster. The various cluster filesystems are exported over communication links available to the cluster head, and mounted remotely by workstations into respective their operating system namespaces using techniques presented in Section 2.5 in Chapter 2.

4.4 Usage

Features provided by the two filesystems in combination may broadly be classified into four categories, namely: sensor data access, event configuration and notification, application configuration support and application binary deployment within the network. This section describes the usage aspects of these features.

Data Centric Applications

The data centric functionality supported by the cluster filesystem is encapsulated by three distinct sections of its exported namespace - shown in Figure 4.4 for a cluster consisting of 2 each of temperature(s0, s2) and photo (s1, s3) sensor nodes. The first section, as described earlier is through individual sensor directories, one of which exists for each sensor node within the cluster. These sensor directories contain files to control sensor operation, access sensor readings, and to define and retrieve events within the sensor.

An alternate means to expose sensor nodes is through logical interfaces, that emphasize properties associated with the nodes such as associated sensor type. A data-centric application might not be concerned about *which* sensor node within a cluster the readings are being retrieved from as long as its sensor is of a certain type. Similarly, applications might want to retrieve readings from *all* nodes with sensors of a certain type. In such cases providing ready means by which to locate and target nodes with required type of sensors would be handy. Filesystem namespaces can fulfill this requirement by supporting group directories which associate sensor nodes based on custom criteria. The example shows one such grouping based on sensor type yielding group directories */temp* and */photo*. The groupings can be based on dynamically changing properties, in which case sensor nodes potentially change group membership in the course of operation. Maintaining sensor

node groups based on remaining battery power allows incoming requests to be easily targeted towards ones with maximum battery life. Sensor network based applications for observing wildlife migration [85] can benefit from location based grouping of GPS sensors mounted on the animals.

Often in sensor networks, data of interest is not individual sensor readings, but a collective value derived across all sensors in the cluster, often called aggregate property. In recognition of this, sensor abstractions commonly provide support for aggregate properties in some form. Projects based on database [127, 84] abstractions, for instance, have enabled application of common aggregation functions available in query languages such as `MIN`, `MAX`, `AVERAGE` and `SUM` to sensor data. Aggregation files implementated within the cluster filesystem in our model serve the same purpose. Read operations performed against these files results in a corresponding aggregation function being applied to sensor readings from all nodes within the cluster. While fulfilling these requests, cluster filesystem may retrieve sensor readings from the nodes each time the aggregate value is produced, or use local caches that are updated periodically.

Event Based Applications

In addition to providing access to sensor network data, the cluster filesystem enables users to register, manage and get notified of events generated within sensor networks using a file based interface. Data retrieval operations of the sort supported in our work fit into a conventional request response model. Event based operations in contrast, typically involve an initial configuration phase wherein a client (an application on the workstation) defines network events of interest with a 'registration' entity (cluster filesystem in this case), and from there on gets notified asynchronously whenever the events occur. This interaction does not comply with the request response mode of operation and cannot naturally be supported using filesystem abstractions. Therefore, as with our handling of breakpoints in the embedded debugging application presented in Section 3.6

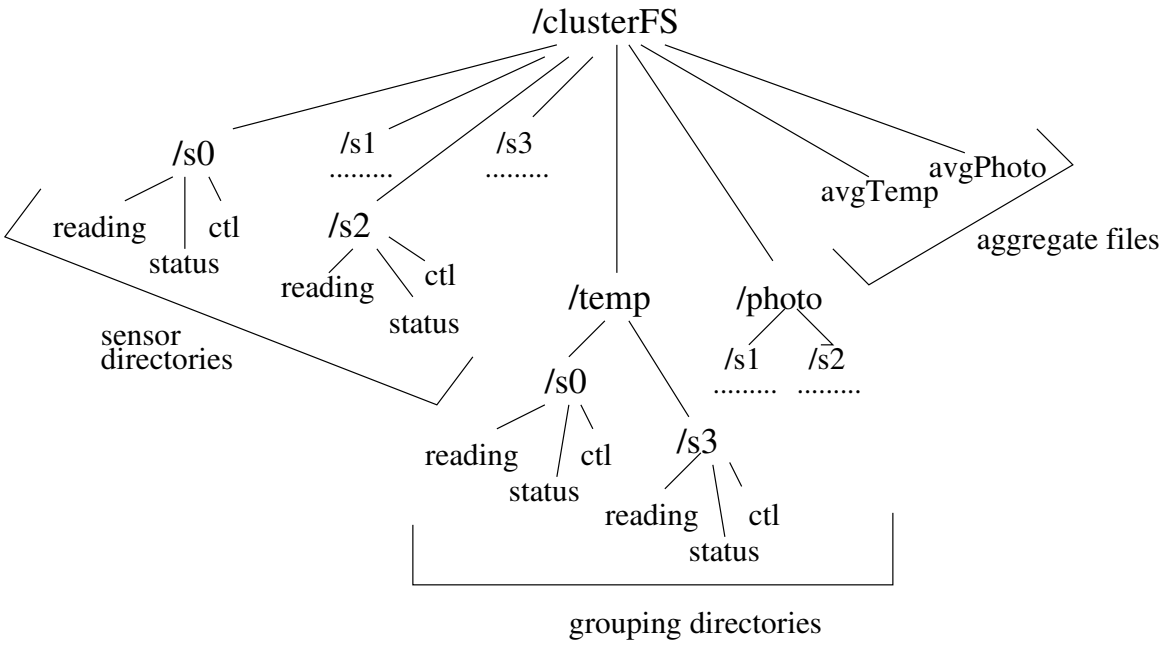

Figure 4.4: Data-centric Section of Cluster Filesystem Namespace

of Chapter 3, event notification is implemented in a synchronous manner, by having the client read a designated file on the server to learn about event occurrence.

The cluster filesystem namespace contains event files that allow workstation-based applications to configure and get notified about events within the cluster. In our implementation, a designated file type named (named *event*) exists (created during discovery) within each of those sensor directories whose corresponding sensor node contains event resources of either the temperature or photo type. Values written to these files are used to configure threshold values for corresponding event resources on the sensor node. When readings higher than the threshold are detected during sensor node's periodic sampling of its sensor, the node issues an event notification message that is received by the cluster filesystem, which makes note of the event occurrence. Workstation-based applications receive these notifications by reading the sensor node's event file. The read operation

blocks until when a suitable event occurs.

Sensor Application Configuration & Deployment

While the previous two scenarios represent filesystem usage during network operation, file abstractions can help during the sensor application configuration and deploy phases as well. They do so by enabling the three operations at the core of building and deploying software, namely: configuration, compilation and installation as is done conventionally using *configure*, *make* and *make install*. The objective is not necessarily to apply these tools directly, but instead to draw associated principles.

An issue to be addressed in using this approach for sensor networks is that the software configuration and installation operations target a distributed network of remote sensor nodes rather than the very workstation (henceforth called host) on which they are being run as is conventionally the case. The underlying principle behind our work is that a filesystem abstraction specially tailored to support configuration and deployment can bridge the gap between the host and the network sides. It is worth noting that the software build operation remains unchanged from present practice whereby the host generates binaries for sensor architectures by using standard cross compiling techniques.

Software Configuration

The inevitable differences in system configuration relating to architecture, hardware, operating systems and compilation tools have motivated the development of techniques to make software sources usable across a wide configuration spectrum. A popular technique among these involves use of the *configure* tool, which draws necessary system information and uses it to fine tune software

sources so as to suit the system configuration.

Applying a similar technique, our work uses the filesystem to configure sensor software to match a network's characteristics. Sensor software compilation phase is preceded by a configuration process which retrieves necessary sensor information through the filesystem to fine tune the software sources and guide compilation. This can help automate configuration of software sources for different sensor architectures thereby making them usable with a variety of networks.

In our implementation, cluster filesystems provide a file by the name of *platformtype* which when read returns the sensor platform type as either *tmote*, *mica* or *telos*. Given that sensor application Makefiles are often written to accept sensor architecture as a build target, this information can be obtained from the network automatically as shown below, instead of being set manually apriori.

```
>cat /sensornet/cluster0/platformtype
    tmote
>make 'cat /sensornet/cluster0/platformtype' -C /sensorapps/RedBlink
```

Software Deployment

The approach used for deploying software on sensor nodes draws from the recursive technique often used with 'make' for building software with multiple subsystems. The top level Makefile in these packages has little idea on how to build the subsystems. When *make* is invoked at the top level, the task of building the subsystems is delegated through repeated re-invocations of itself on Makefiles within the subsystems' directories. This approach to application building has the advantage of separating and confining the subsystem build logic to within their respective directories.

Analogously in the filesystem model, subsystems within a sensor network in the form of clusters export filesystems with mechanisms to program the constituent sensors. Hence the more complicated task of remotely programming a large network is reduced to one of programming numerous localized clusters using the filesystem abstractions they export, which is more tenable. The cluster head responsible for programming the various sensors under its purview is aware of the programming technique(s) supported by the sensors and applies them accordingly. The user level interface provided to deploy sensor applications consists of generic file operations which allows the technique to be naturally integrated with the application build phase.

Control files (named *ctl*) within sensor directories identically support the *program* command which may be used to deploy binaries into sensor nodes. A binary may be deployed into a sensor node from a remote workstation by writing the command along with a repository number (explained below) into the control file within the appropriate sensor directory, as shown below:

```
echo 'program 3' > /sensornet/cluster0/sensor0/ctl
```

Doing so invokes the appropriate mechanism to program the sensor with image stored in repository 3, thus providing polymorphic behavior.

Cluster file systems contain cluster wide 'repository files' which are repositories of sensor binary images on the cluster head. These repositories act as local caches within the cluster head into which binaries may be copied from the deployment node and disemminated across various sensor nodes within the cluster. Since these repositories are represented as files, sensor binaries can be directly copied into them from the deployment node as shown below:

```
cat /sensorapps/app1/main.ihex > /sensornet/cluster0/image3
```

Logical sensor groupings supported by cluster filesystems play a useful role during network

programming. Within a sensor network, application binaries vary depending on sensor architecture, functionality etc. When the filesystem provides sensor groupings along the same criteria, it provides a simple means by which to decide on the binary appropriate for each sensor. This can potentially reduce the complexity necessary in scripts used for deploying software across a network.

We illustrate the concepts involved in sensor software deployment using a prototype, shown in Figure 4.5 and consisting of two clusters for which applications are configured, compiled and deployed from a remote deployment node in the form of a workstation. The two clusters use dissimilar programming mechanisms that are respectively based on radio and USB links. The role of cluster head is performed by a workstation located in close proximity to the clusters. Sensor nodes in each cluster are partitioned to execute two applications, namely RedBlink or BlueBlink wherein they toggle LED's of the respective colors; each cluster thereby has multiple (two) binaries executing within itself.

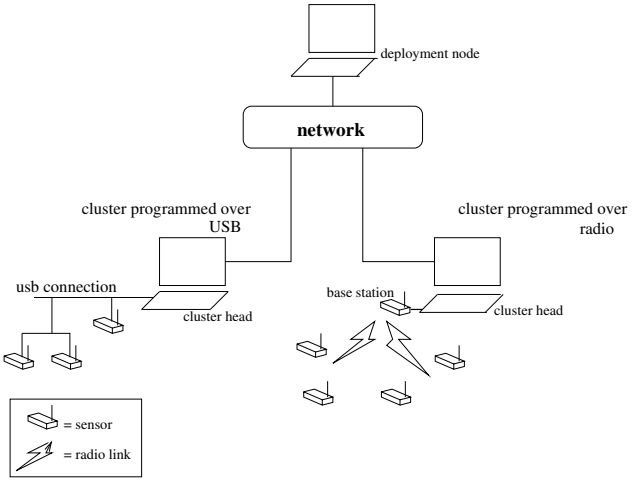

Figure 4.5: Prototype

Each of the clusters exports a filesystem over TCP-IP which is mounted by the remote deployment workstation. The filesystem namespace is shown below:

```
\cluster0
  \s0 \s1 \s2
  \RedBlink
     \s0  \s2
  \BlueBlink
      \s1
  platformtype
  image0
  image1
```

We present a deployment script that programs the multiple binaries (RedBlink and BlueBlink) within cluster nodes using heterogeneous links (USB and radio), all using common file operations. The *RedSensors* and *BlueSensors* directories group sensors based on the executing application, thereby providing an easy basis by which to decide on the binary appropriate for each sensor. Use of loops enables operations to be repeated over multiple clusters and sensors, which makes it feasible for large scale networks to be programmed with this technique. Note that the intricacies and differences in programming sensor nodes has been completely concealed behind the generic file interface.

```
# for each cluster
for(c in /sensornet/cluster*) {
  cd $c
  # transfer both Red & Blue sensor binaries to the repositories in cluster
  cat /sensorapps/RedBlink/main.ihex > image1
  cat /sensorapps/BlueBlink/main.ihex > image2
  # first program red sensors
```

```
   for (s in $c/RedSensors/s*){
       cd $s
       # program binary in image 1 into red sensors
       echo  'program 1' > ctl
   }
  # then program blue sensors
   for (s in $c/BlueSensors/s*){
       cd $s
       # program binary in image 2 into blue sensors
       echo  'program 2' > ctl
   }
}
```

4.5 Distributed Filesystem Implementation

In this section, we present an implementation that realizes a virtual filesystem framework for supporting data-centric and event based applications along with sensor software configuration and deployment. The objective is to demonstrate the implementation of a hierarchical, distributed filesystem spanning the sensor and cluster head levels that is compatible with computational capabilities of these devices. We illustrate the ability of the virtual filesystem to support optimization features such as logical groupings and aggregate functions, and to partition complexity in implementing data and event related functionality across constituent filesystem hierarchies.

Sensor Filesystem

Using the filesytem model, the primary means by which sensor nodes interact with the outside world is through the lightweight filesystem abstractions they implement. At any point of time, sensor nodes may operate in one of two modes - namely application mode and configuration mode. In each of the two modes, sensor nodes export a lightweight sensor filesystem that serves requests from the overlying cluster filesystem to access sensor resources. On starting up, sensors operate in configuration mode, thereby allowing users to configure applications by pulling necessary architecture information through the sensor filesystem. Once applications have been configured and compiled, binaries are loaded and invoked on the sensors through the cluster filesystem. In the process of executing the binaries, sensors transition to application mode. During subsequent software builds, the sensors are brought back to the configuration mode through the cluster filesystem and the configuration/compilation/deployment process repeats itself. The transition of the sensor nodes across the two modes is shown in Figure 4.6.

The sensor filesystem units are designed to be compatible with the resource constrained nature

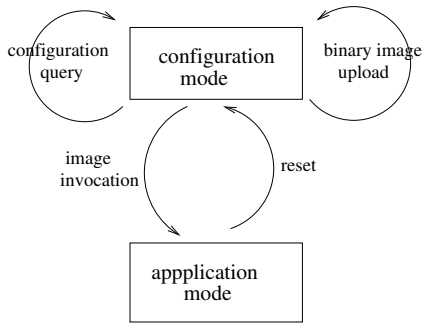

Figure 4.6: Sensor State Transition

of sensor nodes, which we ensure by making a series of concessions in terms of their functionality. Minimal session state is maintained by each sensor filesystem in its interactions with the overlying cluster filesystem. Each incoming request is self contained in terms of its associated parameters and is processed in isolation.

The namespace exposed by sensor filesystems consists of a collection of resources each specified by an integer identifier (known as resource identifier) - string names are avoided for sake of simplicity. The application mode filesystem exported from a sensor node exposes data and event resources for each of its on-board sensor types. Our implementation work used sensors nodes with two sensor types for temperature and/or light (photo) sensing. In combination, this gives 4 distinct resource types with resource identifiers as shown in the table below:

Resource Type	Resource Identifier
Temperature Sensor Data	1
Temperature Sensor Event	2
Photo Sensor Data	3
Photo Sensor Event	4

The configuration mode filesystems in contrast universally have two fixed resources in their

namespace, namely *sensorarch* and *sensorapp*. These two files allow users to extract architectural information from within the sensor through its filesystem interface during the configuration phase.

The resource namespace is flat (non nested), which obviates having to interpret complex paths through an explicit navigation operation. Since the overlying cluster filesystem accesses sensor node resources using their resource identifiers, client file descriptors are done away with. This approach does impose the limitation that only one resource of each kind can be present within each sensor node, which is a compromise we make for sake of simplicity.

The only two operations supported by the sensor filesystem are *read* and *write*. Appropriate semantics define the behavior when the two operations are performed on various resources in the namespace. In our implementation, reading data resources returns the current value from the corresponding sensor. Values written to event resources define upper bounds, beyond which when sensor readings are detected, notifications are sent to the overlying cluster filesystem. Resource identifier '0', known as the discovery resource, is reserved specially for filesystem namespace discovery. This resource when read at increasing offsets returns information about various resources available on the sensor node, and hence helps the cluster filesystem during the sensor discovery process as elaborated upon later. In the configuration filesystem, the *sensorarch* file when read returns the architecture of the sensor node as one of a few standard values (tmote, telos, mica, etc.); *sensorfunc* returns a scenario specific application name which in the case of the prototype was either 1 or 2, corresponding to RedBlink or BlueBlink.

Cluster head devices access resource namespaces exported from associated sensor nodes using an RPC based protocol designed as part of our work. Low bandwidth of the radio based communication link between the cluster head and sensors, as well as the resource constrained nature of sensors motivate us to keep the protocol design simple. The sensor filesystem accepts input messages of type *Iread* and *Iwrite*, corresponding to the *read* and *write* operations issued by the cluster

head. The filesystem respectively responds with output messages of type *Oread* and *Owrite* which contain either return data or a success flag. Apart from these, the sensor filesystem uses output messages of two other types, namely: an asynchronous *Oevent* message to notify the cluster head of events, and *Oerror* in case of error during completion of read or write operations. Input message are uniformly 5 bytes long, and consists of a series of single byte fields as shown in Figure 4.7

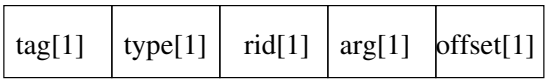

Figure 4.7: Sensor filesystem protocol packet format

The *tag* field serves as a message identifier which can be used by the sensor filesystem to distinguish between duplicate incoming messages, and to match requests and responses on the cluster head side. The *type* and *rid* fields respectively specify the message type as being either read or write and the resource that is being targeted. Finally *arg* and *offset* specify a 1-byte argument and an offset for relevant operations. Output messages have an identical format, only without the offset field. The protocol requires reliable, non-duplicate message delivery between the cluster head and sensor devices. This requirement may be fulfilled over lossy radio links by use of a low overhead technique such as the alternate bit protocol [27], which however restricts the effective communication throughput that may be achieved.

Our implementation of the sensor filesystem uses the TinyOS programming model [114]. The filesystem has been deployed on Moteiv sensor nodes from Crossbow, which is based on the TI MSP430 microcontroller operating at 8 MHz and consisting of 10 KB of RAM. The TinyOS model (described in the Related Work section 4.6), uses a set of software 'driver' components for on-board hardware peripherals that are wired with components defining application logic to implement sensor applications. This wiring (known as *configuration*) for the sensor filesystem is shown below:

```
configuration SensorFS {}
implementation {
  components Main, SensorFSM, GenericComm as Comm, PhotoTemp as Sensors, TimerC;

  Main.StdControl -> SensorFSM;
  SensorFSM.CommControl -> Comm;
  SensorFSM.ReceiveMsg -> Comm.ReceiveMsg[AM_INMSG];
  SensorFSM.SendMsg -> Comm.SendMsg[AM_OUTMSG];
  SensorFSM.EventTimer -> TimerC.Timer[unique("Timer")];
  SensorFSM.SensorControl -> Sensors.TempStdControl;
  SensorFSM.SensorControl -> Sensors.PhotoStdControl;
  SensorFSM.PhotoADC -> Sensors.PhotoADC;
  SensorFSM.TempADC -> Sensors.TempADC;
}
```

The filesystem core (as specified in the `components` statement) is implemented in the `SensorFSM` software component, which uses a set of hardware peripherals relating to radio communication (`Comm`), temperature and light sensing (`PhotoTemp`) and timing (`TimerC`). Each of these components has associated 'uses' and 'provides' interfaces that are connected according to the wiring shown in the listing. For instance, the `PhotoTemp` component *provides* the 'PhotoADC' interface for sensing and retrieving light sensor readings as digital data. This interface is used by SensorFSM to obtain sensor values as and when required. The 'Comm' component allows access to an on-board wireless transceiver based on the IEEE 802.15.4 standard [65] and provides physical layer (PHY), media access (MAC) and checksum capabilities.

The configuration filesystem is additonally equipped with relevant Deluge [63] components to

support network reprogramming and the dual mode of operation. Deluge is component available with TinyOS that allows a limited number of images to be loaded, stored and invoked within a sensor node through commands issued over wireless; in our work the commands are issued by the overlying cluster filesystem. Sensor nodes store binaries corresponding to the configuration and application filesystem within Deluge images slots on the sensor nodes. Switching execution between these binaries (modes) is achieved by the cluster filesystem issuing necessary Deluge commands to the sensor nodes.

Cluster Filesystem

The cluster filesystem uses underlying sensor filesystems to build file abstractions for using and deploying clusters. Cluster head devices have more abundant computational, memory and communication resources than the individual sensor nodes. This relative abindance of resources enables the cluster filesystems to be designed with emphasis on providing rich functionality, in contrast to sensor filesystems wherein design focus is on compatibility with sensor nodes' resource constrained nature.

The role of the cluster filesystem is analogous to that of the Embedded Filesystem block presented in the Technology chapter, which is to implement and export 9P based synthetic filesystem abstraction for hardware in the form of a cluster of sensor nodes. Our discussion of the cluster filesystem design concentrates on aspects that are germane to sensor networks and bypasses the conventional issues in 9P fileserver design, already described in Chapter 2. Ideas are presented using cluster filesystem implementations from our work. The implementations are written in Java, since the software available for remotely interacting with TinyOS based sensor nodes from the cluster head is Java based. We use existing filesystem implementations in Java from the JStyx project [71] as a basis, on top of which cluster filesystem functionality is built.

The cluster filesystem namespace can be hardcoded in its implementation, or generated at runtime. We incorporated a simple sensor resource discovery process within the cluster filesystem whose namespace is generated at startup time. The discovery technique is inspired by the enumeration process used in USB, whereby whenever a device is attached to a host USB port, the host iteratively retrieves information about the various configurations supported by the device (denoting capabilities), and correspondingly updates its local USB device database . Similarly, when discovering the set of resources exported by a particular sensor node , the cluster filesystem reads the corresponding sensor filesystem's discovery resource as identified by resource id '0', at increasing offsets (starting from 1) to iteratively retrieve information about all resources exported from each sensor node. The cluster filesystem supports files of two types namely *EventFile* and *DataFile*, corresponding to event and data resource types within sensor filesystems. Each time a record of a sensor resource is read back, a new file of the appropriate type is created within the cluster filesystem. The Java code implementing discovery within the cluster filesystem is shown below:

```
/* create new directory for sensor node with name 'sNN' */
sensor_dir = new StyxDirectory("s" + String.valueOf(sensorno));
while (true) {
 outpacket.set_offset(ridindex);  /* incremented by 1 each time */
 outpacket.set_type(Iread);  /* define operation to be 'read' */
 outpacket.set_rid(DISC_RID);  /* target the discovery resource */
 mote[sensorno].sendrecv(outpacket, returnpacket);
 if (returnpacket.get_arg() == NO_RSP) /* no more resources? */
   break;
 else {
   if (isEventResource(returnpacket.get_arg()))
```

```
      discoveredFile = new EventFile(returnpacket.get_arg(),sensorno);
    else
      discoveredFile = new DataFile(returnpacket.get_arg(), sensorno);
   }
 sensor_dir.addChild(discoveredFile);
}
```

In this code segment, the sensor node being targeted is identified by a unique network number as given by variable `sensorno`. At the start of the discovery process for each sensor node, a sensor directory is first created by instantiating a new object of type `StyxDirectory`. Read requests are sent to the sensor filesystem targetting the discovery resource at offsets starting with 1 and incrementing each time. Each response packet contains information about the resource within the sensor filesystem namespace corresponding to the read offset, based on which an appropriate file (*Eventfile* or *DataFile*) is created in the cluster filesystem. Once the discovery process completes, the sensor directory associated directory will have files mirroring data and event resources on the sensor node, as well as 'static' files such as *ctl* that are present for every node. The discovery process is engaged in for each sensor node within the cluster, at the end of which sensor node directories for the entire cluster are in place.

[CHG:sensor-links] The cluster filesystem implementation supports sensor groupings and aggregation files by maintaining sensor node collections created during the initial discovery process. Corresponding to the two sensor types in our implementation, the filesystem maintains two groups, as represented by the *temp* and *photo* directories in Figure 4.4. These group directories in turn contain sensor directories for those nodes with sensors that match the group's type. Every time a resource type is detected of a certain type within a sensor node, the nodes' sensor directory is added to the appropriate group directory. Each sensor directory is thus 'visible' in two places -

in the top level cluster filesystem directory and the group directories, representing hard links. Aggregate files are implemented using the polymorphism feature in Java. An abstract *AggrFile* class exists that has associated with it a set of sensor files whose data is being aggregated. For each specific aggregate operation (such as MAX, AVG etc), a class is derived from *AggrFile*, that implements read semantics appropriate to the operator. As part of our work, we have implemented classes corresponding to average and maximum operator file types.

Polymorphism is also used to support multiple programming mechanisms using identical command interfaces with control files. A base *CtlFile* class exists using which subclasses named *USBCtlFile* and *RadioCtlFile* are defined, corresponding to programming mechanisms based on radio and USB links respectively. These subclasses override the default handling of the *program* command with the required set of operations necessary to program a sensor node using their corresponding programming mechanism. Depending on the mechanism supported by a particular sensor node, a control file of the appropriate subclass type is defined within the node sensor directory.

Splitting filesystem functionality between the cluster head and sensor node levels enables us to provide event support that is superior to that provided implicitly by the sensor nodes. Specifically, the technique allows us to support blocking semantics on read operations against event files and multi client event support. The sensor filesystem is ill-equipped to maintain the state associated with blocked read requests, as this can progressively consume considerable runtime memory on sensor nodes. Read operations on event files are thus not passed on to the sensor filesystem, but instead made to block within the cluster filesystem, and processed once an appropriate event notification is received from the sensor node. Also, each event resource within a sensor node can support only a single event at a time. The cluster filesystem mitigates this limitation by reusing that one available event to serve multiple client event requests each with possibly different configurations. The cluster filesystem does this by registering on the sensor node an event that is the

composition of all events requested by various clients. Each time the sensor node issues an event notification, the cluster filesystem examines the occurred event to determine which of the client event configurations it satisfies and processes blocked read requests corresponding to those. Since in our implementation, an event configuration is defined by the sensor reading threshold, composing multiple events reduces to identifying the least among these and using it to set the threshold on the sensor node.

4.6 Related Work

In this section we present work relating to three central topics in sensor network application design, namely: programming paradigms, sensor network middleware and sensor binary deployment techniques. Our work uses available programming paradigms to implement filesystem abstractions that offer middleware services such as sensor data access and event configuration/notification, while also supporting application deployment on sensors within the network,

A basic task in implementing a sensor network is to program individual sensors with the desired functionality . TinyOS [114] is an event driven operating system that provides a popular programming paradigm for low end sensors. The tool includes software component 'drivers' supporting common hardware resources on sensor platforms such as timers, sensors, radios etc. Sensor applications are created by 'wiring' together these hardware drivers to software modules implementing application functionality. The applications are then compiled and programmed among various sensor nodes within a network. TinyOS is an enabling technology in our work, using which filesystems are implemented within sensor nodes. The Emstar [19] project provides software environments for Linux-based higher end sensor platforms such as cluster-head devices. The presence of a richer set of resources within these devices enables Emstar to offer more sophisticated features than TinyOS for user applications such as inbuilt neighbor discovery and support for hosting web servers to disseminate sensor data.

A contrasting approach used with large scale sensor networks is macro programming which enables software design from a network wide perspective. [CHG:sensormacro]Using this approach a user specifies the high level network functionality through global behaviour descriptions. The reponsibility of realizing these descriptions in terms of applications executing among the various sensor nodes distributed across the network rests with the macro programming environment. Details

regarding distributed code generation/instantiation, sensor node coordination and remote data access are hidden from the user. Programming of sensor network applications has been attempted using functional [94] and object oriented programming paradigms [112]. Functional languages are able to treat sensor readings as data streams and perform functional 'macro' operations such as *map* and *fold* to operate on entire datasets of sensor data. As part of more language independent work, Kairos [57] enables programming of entire networks using global behavioral descriptions, while the PIECES framework [82] provides programmers with a network state abstraction with can be used to observe and interact with the network.

Various middleware tools provide data centric capabilities to aggregate, query and filter generated sensor data. TinyDB [84] provides a traditional query based interface for TinyOS applications. SQL-like queries issued on a workstation are processed within the network consisting of sensors executing TinyOS based software components that support TinyDB. Cougar [127] and SINA [106] respectively provide distributed database and object models for interacting with sensor networks. They are specially tailored toward long running queries, which they achieve by tuning the sensor sampling and data reporting rates according to remaining battery life. TinyLIME [38], which has its roots in Linda [53], provides data sharing among the entities of a sensor networks by use of a shared tuple space that may be read from and written to.

Event support in middleware is provided by Impala that offers an event based programming model which supports a fixed set of events relating to on-board timer, sensor data and radio communication. DSWare [80] provides fault tolerant event support services in the presence of node and link failures by using sensor groups to incorporate redundancy. Events are specified using a sophisticated description mechanism and notified along with an associated confidence value that is representative of the certainty of the event's occurrence. Cortex [29] and Mires [108] middleware solutions provide event notification functionality using publish/subscribe semantics thereby

enabling multiple clients to be supported.

Projects such as Agilla [50] and Smart Messages [72] have tried to use code movement based techniques to implement middleware, while MiLAN [60] uses network reconfiguration for meeting performance and QoS requirements. The Sentire [31] project shares objectives with our work in terms of providing a framework on which to build sensor network middleware. Other common programming models that could support the sensor applications include technologies for distributed systems such as CORBA and SOAP [36, 15] which have the significant disadvantage of being relatively heavy weight.

The ability of the presented file model to draw architecture information from within the network to configure sensor applications has not been previously investigated to our knowledge. The very need for configuring software though, arises due to differences in platform architectures, which can be avoided by using virtual machines which offer a 'write once run anywhere' model. Virtual machine implementations for sensors have been proposed with Maté [79] and MagnetOS [81] to address sensor programming and code distribution. However functionality of these virtual machines in resource constrained sensor devices is severely limited which has affected their acceptance.

A difference between a majority of the projects described and our work is that, while they rely on rich API's and programming models creating a high entry barrier for application developers and users, we provide a simple, familiar file interface with which to interact with networks. File abstraction provides a reduced, core set of capabilities meant for use either directly in applications, or to implement richer middleware tools as required. Limiting the number of features supported allows the abstractions to be implemented in severely resource constrained sensor nodes. Further, the application software implementation is decoupled from sensor access mechanisms, thereby making the software interoperable with other means of access. The presented approach reuses existing, popular technologies to implement various pieces of the virtual filesystem infrastructure,

including TinyOS for implementing filesystems at the sensor level and Fuse [52] for importing filesystem at the workstation level. The implementation would continually from improvements as these class of tools are developed over time.

4.7 Evaluation

We present a basic evaluation of our implementation of the virtual filesystem model for sensor networks. We address the feasibility of the approach in terms of memory requirements on sensor nodes. Using empirical means we ascertain the intrusiveness of the approach within existing sensor applications. Finally, we measure and account for the latency of read requests made from clients to the cluster filesystem.

Memory, processor and radio resources consumed by the sensor filesystem supporting data and event based operations are shared within a sensor node with the core network application being executed in the node. Among these, processor and radio resources are only used when serving a request, so their utilization is dependent on rate at which requests are received. A constant price is paid with regard to code and data memories however, as they are shared by the filesystem and network application throughout the course of the node's execution. This makes it imperative that the filesystem consume memory judiciously. The binary segment sizes for a minimal sensor application using the sensor filesystem providing data/event support and compiled for 16-bit MSP430 based Tmote sensor nodes are: a little over 20KB for code segment (residing in flash) and 700 bytes for BSS+Data segments (residing in RAM). The filesystem supporting configuration and deployment was larger due to the use of Deluge; its code segment was 31KB in size and BSS+DATA segments 1450 bytes. These sizes are acceptable considering that the Tmote device has 48 KB of flash memory and 10KB of RAM, which is typical of TinyOS compatible sensor mote devices.

Each time a request is served by the sensor filesystem, the processor within the sensor node has to spend clock cycles processing it. Since the request processing time could potentially have been spent working on a network application related task, the filesystem operation is intrusive to some degree. To estimate the worst case intrusiveness of the filesystem in our prototype, we ran a

computationally intensive task on the sensor node in conjunction with a sensor filesystem, which was repeatedly targeted with read requests. The difference in execution times of the computational task with and without the sensor node concurrently serving read requests gives an indication of the level of intrusiveness of the filesystem. The computational task chosen was the continuous incrementing of a counter from 0 to a large value (0x11fffe), implemented as a TinyOS process that adds one to the counter each time and re-schedules itself, as shown in Listing 4.1 below.

Listing 4.1: Script for programming heterogenous multi-cluster sensor nodes

```
task void counterincr() {
  if (counter < 0x11fffe) {
     counter++;
     post counterincr();
  }
  else
     count_over = 1;
}
```

At the same time that this sensor application is executing, a client application continually makes read requests to the sensor filesystem, indirectly through its overlying cluster filesystem. Since each request is made immediately after the previous one finishes (and the sensor filesystem processes only one request at a time) this is close to the maximum achievable rate at which requests can be made to the sensor filesystem by clients using the communication infrastructure on hand. The execution time of the computation task without the filesystem running concurrently was 96953 milliseconds; the task execution time while concurrently serving read requests was 105204 milliseconds, thereby indicating an overhead of 8251 milliseconds. This shows that the maximum

overhead that may be incurred by network applications while concurrently serving sensor filesystems = (8251/96953)*100, or 8.51%. This overhead is directly proportional to the rate at which requests are presented to the sensor filesystem, which in turn is bound by the communication protocol and link utilized.

Latency of read requests issued by a client and targeting sensor files through the cluster filesystem was 78 milliseconds. This latency (T_{lat}) consists of three distinct parts, namely:

- communication delay between client and cluster head ($T_{cl<->CH}$)
- communication delay between cluster head and sensor node ($T_{CH<->SN}$)
- request processing time on the sensor node (T_{req})

Through measurements, T_{lat} was determined to be = $T_{cl<->CH}$ + $T_{CH<->SN}$ + T_{req} = 78.3 milliseconds

T_{req} was measured as the difference between latency of read requests and round trip communication time from client to sensor nodes, and was observed to be less than the smallest measurable time interval in our setup, which was 1 millisecond.

$T_{cl<->CH}$, which is the round trip communication delay from client to cluster head was measured to be = 2.1 milliseconds.

$T_{CH<->SN}$, which is the communication delay between cluster head and sensor node = T_{lat} - $T_{cl<->CH}$ = 78.3 - 2.1 = 76.2 milliseconds

The results show that the delay in communication between the cluster head and sensor nodes accounts for about 97% of the latency in completion of a read operation. Since this is the round trip delay, it represents the time for transmitting both the request and response packets to the sensor

nodes, equal to 10 bytes of data + 10 bytes of header. The effective bandwidth of the sensor radio communication involved = 160 bits/76.2 milliseconds = 2.12 Kbps. Given that the radio used in the Tmote sensor node is rated to support a maximum transfer rate of 250 Kbps, the effective bandwidth observed reflects the sub-optimal use of the communication infrastructure due to communication protocol, among other less significant reasons.

4.8 Summary

This chapter described application of distributed virtual filesystem based abstractions to enable usage and deployment of sensor networks. We present a technique for implementing filesystems of varying capability at the sensor and cluster head levels, which work in conjunction to support file abstractions for the entire network. An implementation of a lightweight filesystem for sensor nodes is described, that is designed to support the generation and operation of overlying filesystems in cluster head devices, while using the limited resources available. We illustrate use of file abstractions in two common sensor network usage scenarios, namely to enable data-centric and event based sensor application classes. Data-centric applications are enabled by allowing individual sensor readings to be directly accessed by readings files in the exported namespace and through more sophisticated features such as sensor grouping directories and aggregate function files. Event support is provided by enabling both event registration and discovery through file operations; clients discover about events by performing blocking read operations on event files which complete when the event occurs. Implementation of this blocking functionality provides an example of how filesystem logic may be split between the cluster head and sensor nodes in implementing

complex features. In addition to its use during network operation, use of virtual filesystem infrastructure during sensor configuration and deployment phases is also described. During the configuration phase, sensor node architectural information drawn from within the network through the file interface can be used to guide the sensor software configuration and compilation process; resulting binaries may then be deployed within sensor nodes by 'copying' them to appropriate files in the sensor network filesystem namespace. These ideas are illustrated using a prototype involving a heterogeneous, multi cluster sensor network for which software is configured and deployed automatically from the host side using shell scripts.

5

Enabling Proxy Based Resource Access for Embedded Devices

The third and final application presents use of virtual filesystems to provide computationally impoverished devices such as sensors and actuators the ability to access significant software and hardware resources such as mass storage, network access, and console for input/output by importing them over generic communication links. In contrast to previous applications, the roles of clients and servers are reversed between the embedded and workstations sides. In addition to providing access to significant resources, the model allows incorporation of richer functionality and logic within software executing on embedded devices, which often makes the code more portable. The contribution of our work is to use the core filesystem technology to provide a framework for lightweight devices (called clients from here on) to import and use resources from a remote workstation. We also describe possible extensions to the basic usage model including its adoption in a system on chip scenario. These ideas are illustrated using an example application.

5.1 Introduction

The embedded market today is replete with devices that may be categorized as high end microcontrollers, which find utility in fields as diverse as automotives, electronics, machinery and sensor/actuator networks. These devices (such as MSP430 series from TI [11], LPC series from Philips [1]) are typically based on processor cores capable of operating at a few 10s of Mhz, contain a few tens of kB of ROM (code memory) and less than 10 kB of RAM (runtime memory). With limited resources at their disposal, it is often infeasible for these devices to use conventional software solutions such as TCP-IP stacks, which have code footprints of about 200KB. As a result, projects have attempted to adapt software tools such as network stacks [43] and filesystems [39] for use in the lightweight embedded device domain. While resulting solutions provide acceptable but often limited functionality, using more than one such tool in conjunction is cumbersome. Aside from

software tools, providing embedded devices with access to hardware resources (such as a console) directly is infeasible for a number of reasons including the manner in which devices are deployed (out in the field), the numbers and cost involved, and architectural limitations.

In this methodology devices get access to resources resident on remote workstations by importing them over generic communication links. Since the communication and support software infrastructure is reused, the method imposes a nearly constant price for access to an indefinite number of resources. The latency involved in accessing the resources depends on the communication link and the frequency and extent of the resources might have to be tailored accordingly. The methodology can be used only until the development phase of a product when features such as console output and logging to mass storage might be useful for debugging, or retained during deployment as shown with the example in Section 5.3.

As is the case throughout this dissertation, a key concept that is exploited is that anything can be a file – a file system is simply a way to organize a name space as a hierarchical set of objects that respond to well understood commands (`open`, `read`, `write`, etc.). Requisite resources on a capable 'host' device (such as a workstation) are bundled within 9P filesystem abstractions and exported over available communication links for resource contrained embedded devices to mount and use. Since the requirements imposed by 9P protocol are fairly basic, most connection types commonly associated with embedded devices such as serial links, USB and low power radio (zigbee) may be used.

A significant advantage to this approach is that cost of importing resources is independent of the number and type of resources being imported, since the cost of importing a filesystem is largely independent of the size and content of its namespace. Also important for power starved embedded devices, the model follows a request response mechanism which means the client and host interact only at the client's behest thereby precluding unnecessary communication.

5.2 Methodology Adoption

In this section we discuss the two steps involved in adopting the presented methodology within a system, namely exporting resources from a host entity and accessing them from within embedded devices.

Resource Export

The utility of the presented approach is hinged upon being able to encapsulate resources required within embedded devices using filesystems on the host side. To this end we describe how hosts may export hardware resources such as console, peripherals etc., software resources such as network stacks, and custom resources based on application specific needs.

Given that the standard 9P protocol is used for messaging, it follows that embedded devices can readily interface with the filesystems exported out of Plan 9, or its closely related Inferno [41] operating system, which can be run in 'hosted' mode under Linux or Windows. Use of either Plan9 or Inferno puts at the user's disposal a variety of resources useful to embedded devices exported through file servers. These are mentioned below along with the directory path they are exported from in Plan 9:

- console device providing IO terminal and clock (/dev/cons)
- serial and parallel port for communication (/dev/eia0, /dev/lpt)
- network protocols (/net/tcp, /net/udp)
- mass storage (any part of the file tree can be exported)

However, dedicating a node to run these operating systems is often not possible. In that case standalone 9P filesystems can be created using the NPFS framework [95] and exported off the host

node. An advantage in using NPFS based filesystems is that they carry minimal software dependencies, which makes the filesystems highly portable across various Unix-like operating systems. We have used NPFS to create reusable filesystems for exporting console IO, TCP-IP and mass storage resources, thereby allowing each of them to be exported to the client sides. These filesystems also serve as references to help users build custom variants to suit their specific needs.

Clone filesystems described earlier in Section 2.4 are particularly useful for encapsulating and exporting logical resources such as TCP-IP sockets . Instances of such resources are created on the fly by clients, utilized for a period of time and subsequently released. Using clone filesystems new resource instances are created by reading the *clone* file in the filesystem namespace, which has the dual effect of creating a new instance of the resource (such as a socket) and adding a resource directory to the filesystem namespace containing files to use the newly created resource. Use of clone filesystems to support TCP-IP networking was described in Section 2.4.

Applications may in general, use NPFS to develop custom filesystems (clone or otherwise) to support requirements specific to their application. The ability to package any necessary data through the file system and present it to the embedded side provides system designers with a powerful tool. Implementing a file system with NPFS leaves the user responsible for just defining the file layout of the filesystem using a table and for providing callbacks that define the filesystem behavior when data is read and written.

Embedded Resource Access

Embedded devices can mount and access resource filesystems using a library we implemented as part of this work, whose functions mirror Unix file operations. The library defines 'standard' file operations such as *open*, *read* and *write*, apart from 9P specific ones such as *attach*,*clone* and *walk*, with each function name given by prefixing '9P_' to the operation name. The first operation a client

performs in interacting with a remote file system is to mount it using the *9P_attach* function also has the result of associating the file system root with a client specified integer valued file descriptor. In the Plan 9 world, (integer) values for identifying file descriptors are specified by the client side, unlike in Unix where the values are returned by the server during an *open* operation. The root file descriptor is duplicated to another client specified integer file descriptor (and thereby preserved) using the *9P_clone* operation; the duplicated file descriptor is made to point to the required file through a series of single step *9P_walk* operations. Hence a typical Unix operation such as:

```
fd = open("/tcp/ctl", OREAD)
```

would translate to a sequence of steps:

```
9P_clone(rootfid, newfid)
9P_walk(newfid, "tcp")
9P_walk(newfid, "ctl")
```

The usage of *9P_read* and *9P_write* is conventional whereby data can be read from or written to a file descriptor, with the return value of the function call being the number of bytes read or written.

In its simplest form the library is single threaded, which means file operations block until they return from the remote fileserver. While this solution may be adequete in basic embedded systems, blocking while operations complete would not be acceptable in more sophisticated systems, which have multiple functionalities implemented in software operating concurrently. Therefore lightweight stackless concurrency mechanisms are used within embedded clients, based on the *libthread* module also used to implement EFS(presented in Chapter 2), so that the embedded system continues to function while waiting for file operations to complete in the background.

Embedded clients accessing remote filesystems use threads corresponding to: message transmission, message reception and system applications (as many as required). The transmission

thread receives 9P messages from application threads and sends them out over available communication links. Receiver thread waits for application threads to request incoming 9P messages, upon on which it listens on the communication link for the next incoming message, which is then returned to the client. The threads communicate using CSP-based blocking channels [33]. Performing a file operation involves separate send and receive stages. The client application thread first generates a 9P message based on the operation needed to be performed and sends it to the transmitter thread over *sndChan*. It then sends a message request to the receiver thread on *rcvReqChan*, and then waits on *rcvChan* for the receiver thread to send it the response message. This allows the rest of the embedded system to continue execution while the application thread waits for the response for a file operation.

The size of the core library was a little less than 10 KB when compiled for a 16 bit RISC (TI MSP430) processor. This excluded the communication code that is responsible for data transfer to and from the communication link being used, which is highly dependent on the peripheral and reliability mechanisms being used. When using serial communication with a plain UART this would amount to less than 1 KB, while using a radio connection with basic acknowledgment/retransmission mechanism would push the code size up by another 10 KB or so.

In capable embedded devices, more sophisticated mechanisms can be used to access the host filesystem. In systems running operating systems that support threads, libraries can use separate, preemptively scheduled threads for completing each request from client applications thereby making the entire system more error tolerant. In an embedded environment with multiple processing elements (cores), a point-point or broadcast network may be shared among the various processors to access a single resource server through their respective libraries. If the devices run Linux, then kernel modules such as v9fs [61] may be used to mount host filesystems within their local Linux namespace, thus doing away with the library.

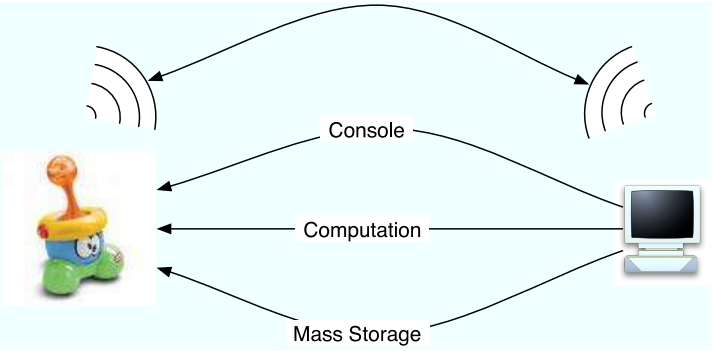

Figure 5.1: Resources exported to Goofy

5.3 An Illustrative Example

This section presents an example that reinforces the basic utility of idea presented in this chapter to concurrently import multiple useful resources into a resource impoverished embedded environment.

The basis of the example is an instructional embedded platform developed in the computer science department using the Goofy Giggles toy (shown in Figure 5.1) that has in its core a TI MSP430 microcontroller (MCU). The MCU can operate at upto 8 MHz, has on chip flash memory of 32 KB and RAM of 5 KB. Using the various peripherals available on the MCU, the platform has been equipped with motors for the wheels, sensor, and a low data rate 802.15.4 radio.

The objective is to equip the platform (henceforth called Goofy) with resources that are hard to come by in an embedded environment by importing them from the workstation. Using these resources Goofy is programmed to accept commands off the keyboard and act accordingly, a task commonly assigned to radio equipped mobile robots [120]. Use of console I/O allow users to enters commands at the workstation keyboard to navigate Goofy, which periodically logs values from its on board sensor. This focus in this prototype is on importing two resources as illustrated in Figure 5.1 and listed below:

- console I/O
- mass storage

The platform represents a concurrent embedded system as the tasks associated with running the motor, sensing and communication need to run simultaneously over time. As described earlier, use of a lightweight concurrency model based on cooperative, stack-less threads allows each of these tasks to access their respective resources through blocking file operations without freezing the entire system.

The communication link for talking to host based file servers used a low powered radio network based on IEEE 802.15.4 standard [65]. The radios involved supported basic checksum capability on top of which acknowledgment and retransmission features were built in software using the alternating bit protocol [27]. This satisfies the 9P protocol's two transport requirements of reliable delivery and preservation of byte order.

The filesystem on the host side was developed using NPFS and exported a namespace as shown below:

```
\GGfs
    console
    log
    computeclone
```

The *console* file is used by Goofy to display its status on the host's standard out and accept commands from the keyboard. Single character motor commands such as: 'f' for faster, 's' for slower, 'r' for turn right and 'l' for turn left could be defined to enable remote control of Goofy. The *log* file can be written to by Goofy to record its operation state (such as speed).

The code size for the prototype was less than 15KB in size, while the data and bss sections combined were less than 1400 bytes. Goofy was able to perform up to 25 file requests per second. As expected, up to 95% of the latency for a file operation was spent in waiting for the response message. This suggests that implementing more aggressive data transfer mechanisms than the alternating bit protocol would result in increased throughput by keeping more packets in flight.

5.4 Discussion

The lightweight proxy approach offers utility beyond simply being able to access remote resources from microcontrollers. We briefly discuss other benefits and extensions of the work presented in this chapter.

Writing Richer code for small devices

Software written for small devices is to varying degrees limited in functionality, interactivity and expressiveness. This is because the price to be paid for accessing resources providing interactivity (such as the console for IO) and functionality (computation, mass storage for logging,) are prohibitively high. The proxy approach alleviates the situation by imposing a constant price for access to an indefinite number of resources. More functional, expressive code can then be written by leveraging the imported resources. This is reflected in the main loop for the Goofy application shown below, where reading input from the keyboard and displaying status are naturally incorporated into the application.

```
9P_read(consolefd, data, 1)
/* react to command  - ommited */
```

```
9P_write(consolefd, GG_status, 1)
9P_write(consolefd, &motor_speed, 2)
```

Moving to a SoC scenario

System on chips devices often have multiple processor cores with a robust 32 bit RISC processor executing general purpose operating systems, along with specialized cores such as DSP's running application specific algorithms. This model can be applied to such an environment whereby the general purpose core exports software resources such as network stacks as filesystems for other on chip processing elements to use. Such an idea can also be applied to circuit boards built around a central processor and consisting of lightweight processing elements such as microcontrollers. Since in these cases, the host and client entities are in the same clock domain, synchronous protocols such as HDLC [10] can be used to provide reliable transfer of 9P messages.

5.5 Related Work

Several projects have investigated the converse of this idea, which is to export abstractions out of an embedded system and use that to interact with it. The closest of these is the 'Styx on a Brick' project [83] which exported the various resources of a Lego Mindstorms robot (motor, sensor) within a filesystem that could be accessed remotely. The emWare emNet [45] and OMG Smart Transducers (ST) [96] systems both provide extremely lightweight protocols that enable hosts to access resources on embedded devices using CORBA and RPC based mechanisms respectively.

An alternative to using this approach is to implement lightweight version of software tools tailored for embedded devices. uIP [43] is a lightweight TCP-IP stack, which as implemented on the TI MSP430 required 4.2k program memory, and 700 bytes of RAM, but allowed only one TCP

session at a time and did not provide assembly of fragmented IP frames. ELF [39] is a lightweight log-structured file system developed for flash memories that is highly tailored towards sensor mote operations.

Access to data networks through proxies is well established practice. With the advent of distributed and network computing, projects have attempted to transparently provide remote access to peripherals such as console [129] and even audio [78]. Of late a similar problem is being addressed for virtualized environments such as VMWare [111], in order to share I/O devices between host and guest operating systems.

5.6 Summary

This chapter presented a technique by which lightweight embedded devices can import multiple remote resources over generic communication links. The technique imposes a constant overhead that is largely independent of the number of resources being imported. The basis of the approach is to universally encapsulate all resources as files, which makes it applicable to software, hardware and custom resources. The technique offers the opportunity to implement richer, more interactive software on embedded devices while imposing nominal demands on them.

6

Conclusion

In conclusion, we recap the motivations for using filesystem abstractions to support embedded software development and present our contributions in realizing and using these abstractions. We present a case for why the use of our approach would remain feasible and relevant in the longer term in the highly dynamic semiconductor industry. We also present ideas for future work, both in the form of concrete objectives and as broad research ideas arising based on the completed work.

6.1 Motivations for Using Filesystem Abstractions

Filesystem abstractions enable use of uniform, well understood interfaces to interact with the heterogeneous processing elements that coexist within embedded systems. The uniformity saves the user from having to be cognizant of the differences in the debug, configuration and other interfaces among the processors. File abstractions effectively allow separation of interface from the supporting implementation, which gives system developers a means by which to safeguard intellectual property associated with the various components within the system. File based interfaces offer familiarity to human users and also a convenient back-end on which to base wide a range of tools for debugging, programming, task automation, and data analysis. By virtue of their compositional nature, file-based interfaces from various constituent components within a system can be naturally assembled to progressively create higher level interfaces with which to interact with the entire system. Since file interfaces can be exported over communication links, they can easily be accessed remotely.

Based on these advantages, our distributed filesystem abstraction is best suited for systems exhibiting certain specific characteristics including: hierarchical and/or locally distributed architecture, heterogeneous composition, autonomous operation among sub-systems. In systems designed based on a distributed architecture, the pieces comprising the filesystem can be naturally

be implemented within the various constituent entities in the architecture distribution; similarly with hierarchical systems, filesystems implemented in the lower levels can be composed hierarchically thereby progressively creating richer filesystems. Filesystem abstractions are well suited for systems exhibiting heterogeneous composition as in these cases uniform file interfaces can help conceal the heterogeneity within the system. File operations issued in our model are serviced in a distributed manner and hence inherently have a high latency. Therefore for performance reasons, our model is better suited for architectures consisting of autonomous subsystems whose cross interactions in the form of file operations would be sufficiently infrequent.

6.2 Summary of Contributions

The central question addressed in this thesis is as to how the workstation centric idea of a distributed filesystem can be implemented and effectively applied to facilitate various software development tasks in the embedded domain. As part of our work we have implemented filesystem building blocks that can be used to hierarchically assemble file abstractions within resource constrained embedded architectures. The hierarchy includes the 'embedded' filesystem block that encapsulates a basic embedded system design unit (such as a SoC device) using a file interface in a resource conscious manner. The various embedded filesystem instances within a system may be composed together using a mount file system (MFS) block which then provides a holistic interface using which to interact with the entire system; the MFS block is thus responsible for providing our model with its compositional features. We provide means to access these filesystems on the workstation side by mounting them within the local operating system namespace, thereby enabling them to be accessed like any other 'conventional' file.

We apply our virtual filesystem technology to facilitate software development in two distinct

embedded application domains. The first application addressed is concurrent debugging and tracing of software executing within multi-processor SoC based embedded systems. File abstractions capture debug interfaces of the constituent (potentially) heterogeneous processors within the system using uniform file namespaces. We present a technique for implementing these abstractions within SoC based embedded systems using filesystem building blocks described earlier. Using prototypes we demonstrate use of these abstractions to support several issues central to debugging and tracing in SoC based embedded systems including:

- providing internal visibility of constituent processor state through a generic file interface
- creation of a logical interface that is independent of system hardware layout
- presenting interfaces for debugging at different levels of abstraction and thereby having the ability to to conceal proprietary debug interfaces among processor cores
- integration of disparate processor debug/trace interfaces into unified system level debug solutions and in the process sharing on-chip resources (such as trace buffers, communication ports) among them

A second application for our filesystem framework is to enable usage and deployment of sensor networks. In contrast to the first application wherein the filesystem framework was used to control and check status of execution of embedded processor elements, in the sensor network domain the file abstraction facilitates event management and efficient sensor data retrieval from within the network. Distributed filesystem abstractions for sensor networks as implemented in our work use filesystems of limited functionality at the sensor node level that support more capable filesystems at the cluster head level, both of which work in conjunction to support file abstractions for the entire network. We demonstrate use of file abstractions to support two common sensor network usage scenarios, relating to data-centric and event based application classes. To complement its use

during network operation, the virtual filesystem infrastructure can also be used to draw requisite sensor architectural information to enable sensor software configuration and to deploy resulting binaries within sensor nodes using generic file operations after compilation. Thus our framework enables creation of a single, central interface by which to access, control and deploy hierarchical networks of distributed, heterogeneous sensor nodes.

6.3 Long Term Relevance

We argue that use of embedded, virtual filesystems for supporting software development would remain feasible and relevant in the longer term. With increasing transistor densities, dedicating a fraction of the on-chip logic to support debugging is affordable [116]. Presence of these debug support structures would be warranted by the increasing volume and complexity of embedded software - as indicated the fact by 2011 it is estimated that a new car will run 100 million lines of code, which more than twice as much as in the Windows operating system [25]. Incorporating dedicated logic within system-on-chip devices for purposes other than supporting core device functionality has been actively considered recently. For instance, Zorian et al [130] have investigated use of 'Infrastructure IP' embedded within SoC devices as an instrumentation technique to collect manufacturing (defect) information which is then used to diagnose yield problems. The automotive sector is beginning to accept the Nexus standard [20] that uses on-chip debug logic to standardize message based interactions between processors and debugging tools.

To complement the inexorable increase in the amount of software in embedded systems, the number of processor cores within designs is increasing as well [18]. Reasons inducing this trend include the broadening range of features being supported by embedded devices and the power advantages incurred in using multiple processors running at lower clock rates rather than using a

single higher speed processor [77]. In the presence of multiple processors, there is a need to support concurrent and 'global' debugging of software across multiple processors. The emerging trend is thus to move towards dedicated system debug interfaces [62, 20] which support features such as cross triggering [3] and synchronized start/step operations. The work presented is a step in the same direction.

There is definitive shift in the system design process from using custom built application specific intergrated circuits (ASICs) to using third party SoC devices. This trend is motivated by the costs in designing and manufacturing ASIC's which according to estimates by the International Technology Roadmap currently range between $10 million and $20 million for an SoC within a PDA. These costs are estimated to increase as feature sizes shrink because of the increase in complexity and number of masks required as part of the chip fabrication process [28]. To counter these prohibitive costs system design companies resort to buying the SoCs backing their products from third party vendors specializing in semiconductor design. SoC based design also allows reuse of pre-built hardware blocks which helps designers roll out their products within ever shortening times to market. As systems with increasingly rich functionality come into fore, designing complex chips at their core require expertise in diverse technology areas such as microarchitecture, communication, multimedia etc. [58], all of which is hard to find in a single organization; the solution is to buy off the shelf SoC's which can readily be integrated into systems.

As the use of SoC devices in system development gains ground, system development teams have to contend with the disconnection from SoC development teams and consequently with the limited available design details of the SoC device. As the number of SoCs available in the marketplace proliferate, the differentiator among these devices will be the software and debug support built into the device that allows system development teams to use, program and debug the software

within the device at the time of integration. Our approach allows SoC providers to be expose programming and debug interfaces for their chips at the desired levels of abstraction, while providing them with the opportunity to conceal proprietary details of the internal cores.

6.4 Future Work

In this section we outline specific opportunities for future work that extend the implementation supporting this dissertation. We also discuss broader research questions related to the virtual filesystem model arising based on the presented work.

Our existing filesystem infrastructure has minimal fault tolerant mechanisms incorporated within it. To address this issue, filesystems need to be able to detect and handle errors in hardware that it they are abstracting. A related issue is the detection and handling of failures among one or more filesystems themselves in the distributed filesystem hierarchy. The embedded filesystem block presented has its associated namespace statically compiled into its implementation. A useful extension would be to allow the namespace content information to be specified at startup time; handlers for file operations performed on these files can be implemented as libraries dynamically linked into the filesystem core at execution time.

Our work used the filesystem technology to enable concurrent debugging in a multi processor prototype representing a system-on-board scenario; realizing the model in a system-on-chip environment would help better illustrate its potential. Quantifying the computational requirements imposed by the filesystem on the support processor would help SoC designers make an informed choice of processor hardware for fulfilling the role of the support processor. Implementation of

filesystems as services within popular embedded operating systems (such as VxWorks) can promote acceptance of our model. Use of hardware debug adapters for interfacing the support processor with standard processor debug interfaces can facilitate easy integration of on-chip hardware with the filesystem and also help offload a part of filesystem debug functionality from within the support processor to dedicated hardware.

The implementation of our filesystem model within sensor networks assumes a single-hop communication between the cluster head and sensor nodes. Extending this to support multi-hop scenarios though use of more sophisticated protocols such as RMST [110] and PSFQ [118] can broaden the applicability of our work. Use of higher throughput protocols than the currently used alternating bit protocol would improve the effective latency seen for file operations targeting sensors.

A general question that arises based on our work is whether the distributed filesystem should be built using 9P, or more generally any other request-response based protocol for enabling interactions between client and server sides and across filesystems. While use of 9P allows its associated suite of tools derived from Plan9 to be used with our filesystems, it does not naturally support asynchronous behavior (for event support), and its usage contains inefficiencies in terms of the amount of data transferred. Another intriguing consideration relates to the dependency of our filesystem model on the underlying communication infrastructure. This issue may be addressed by investigating the feasibility and usability of our filesystem infrastructure over specialized communication links found in the embedded domain, such TTCAN - a time triggered bus used in automobiles, and bit based links such as scanchains through use of framing protocols such as HDLC. An ambitious endeavor would be to automatically generate filesystems for SoC devices with a given hardware (core) layout, in order to support existing chips.

Bibliography

[1] 16/32-bit lpc2000 family. http://www.nxp.com/products/microcontrollers/32bit/index.html.

[2] ARM extended trace macrocell (etm) technical reference guide. http://www.arm.com/documentation/TraceDebug.

[3] ARM's coresight on-chip debug and trace technology. http://www.arm.com/products/solutions/CoreSight.

[4] Armulator, ARM. http://www.arm.com/support/ARMulator.html.

[5] Device file system guide. http://www.gentoo.org/doc/en/devfs-guide.xml.

[6] exportfs, srvfs - network file server from plan9 man pages. http://plan9.bell-labs.com/magic/man2html/4/exportfs.

[7] Features of the msp430 bootstrap loader (rev. d). http://focus.ti.com/lit/an/slaa089d/slaa089d.pdf.

[8] Freescale, MPC565 user's manual, 2002.

[9] Introduction to on-board programming with intel flash memory. http://www.intel.com/design/flcomp/applnots/29217902.pdf.

[10] Iso 13239 : High-level data link control protocol.

[11] Msp430 : Ultra low power mcu from texas intruments. http://www.ti.com/msp430.

[12] National ecological observatory network. http://www.neoninc.org.

[13] OCP-IP: Open Chip Protocol International Partnership. http://www.ocpip.org.

[14] Providing asynchronous file i/o for the plan 9 operating system. http://pdos.csail.mit.edu/papers/plan9:jmhickey-meng.pdf.

[15] Simple Object Access Protocol (SOAP). http://www.w3.org/TR/soap.

[16] Simulavr: an AVR simulator. http://savannah.nongnu.org .

[17] The two percent solution. http://www.embedded.com/story/OEG20021217S0039.

[18] What processor is in your product? http://www.embedded.com/columns/showArticle.jhtml?articleID=1931

[19] Emstar: A software environment for developing and deploying wireless sensor networks. In *Proceedings of the USENIX 2004 Annual Technical Conference*, 2004.

[20] The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface, 2004. http://www.nexus5001.org.

[21] Guest editorial: Concurrent hardware and software design for multiprocessor SoC. *Trans. on Embedded Computing Sys.*, 5(2):259–262, 2006.

[22] D. E. L. G. M. H. A. Cerpa, J. Elson and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In *Proceedings of the 2001 ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean, April 2001*, 2001.

[23] K. M.-M. A. Mayer, H. Siebert. Debug support, calibration and emulation for multiple processor and powertrain control socs. *IEEE Trans. Comput.*, 55(2):174–184, 2006.

[24] C. G. A. S. Tanenbaum and B. Crispo. Taking sensor networks from the lab to the jungle. *IEEE Computer Magazine*, 39(8):98–100, 2006.

[25] D. F. Bacon. Realtime garbage collection. *Queue*, 5(1):40–49, 2007.

[26] T. W. Bart Vermeulen and S. Bakker. Ieee 1149.1-compliant access architecture for multiple core debug on digital system chips. In *Proceedings of the International Test Conference*, 2002.

[27] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5):260–261, 1969.

[28] S. Bhattacharya, J. Darringer, D. Ostapko, and Y. Shin. A mask reuse methodology for reducing system-on-a-chip cost. In *ISQED '05: Proceedings of the 6th International Symposium on Quality of Electronic Design*, pages 482–487, Washington, DC, USA, 2005. IEEE Computer Society.

[29] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In *2nd IEEE International Conference on Pervasive Computing and Communications (PerCom 2004*, March 2004.

[30] Bluetooth.com : The official Bluetooth Technology Website. http://www.bluetooth.com/bluetooth/.

[31] J. W. Branch, J. S. Davis II, D. M. Sow, and C. Bisdikian. Sentire: A framework for building middleware for sensor and actuator networks. In *The 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, March 2005.

[32] Lg electronics selects broadcom platform for blu-ray and hd dvd player. http://www.broadcom.com/press/release.php?id=947700.

[33] S. Brookes, C. Hoare, and A. Roscoe. A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31:560–599, July 1984.

[34] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. *Readings in hardware/software co-design*, chapter Ptolemy: a framework for simulating and prototyping heterogeneous systems, pages 527–543. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[35] CIFS. http://www.microsoft.com/mind/1196/cifs.asp.

[36] OMG's Corba website. http://www.corba.org/.

[37] D. Culler, D. Estrin, and M. Srivastava. Guest editors' introduction: Overview of sensor networks. *IEEE Computer*, 37(8):41–49, August 2004.

[38] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco. TinyLIME: Bridging mobile and sensor networks through middleware. In *3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, March 2005.

[39] H. Dai, M. Neufeld, and R. Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the 2nd international conference on Embedded networked sensor systems(SenSys)*, 2004.

[40] E. W. Dijkstra. Chapter i: Notes on structured programming. pages 1–82, 1972.

[41] S. M. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. W. Trickey, and P. Winterbottom. The inferno operating system. *Bell Labs Technical Journal*, pages 5–18, Winter 1997.

[42] A. Dunkels. Full TCP/IP for 8 Bit Architectures. In *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, San Francisco, May 2003.

[43] A. Dunkels. Full TCP/IP for 8 Bit Architectures. In *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, San Francisco, May 2003.

[44] A. Dunkles. Protothreads: lightweight, stackless threads in c. http://www.sics.se/∼adam/pt.

[45] emNet technical overview. www.emware.com.

[46] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM Press.

[47] M. B. Z. Eric E. Johnson, Jiheng Ha. Lossless trace compression. *IEEE Transactions on Computers*, 50(2):158–173, 2001.

[48] R. Faulkner and R. Gomes. The process file system and process model in UNIX system V. In *USENIX Association. Proceedings of the Winter 1991 USENIX Conference*, pages 243–252, Berkeley, CA, USA, 1991. USENIX.

[49] S. F.Oakland. Considerations for implementing ieee 1149.1 on system-on-a-chip integrated circuits. In *Proceedings of the International Test Conference*, 2000.

[50] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *The 24th International Conference on Distributed Computing Systems (ICDCS'05)*, 2005.

[51] L. W. K. A. R. K. Francisco DaSilva, Yervant Zorian. Overview of the ieee p1500 standard. In *International Test Conference 2003*, page 988, 2003.

[52] Fuse: Filesystem in userspace. http://fuse.sourceforge.net.

[53] D. Galernter. Generative communication in Linda. *ACM Computing Surveys*, 7(1):80–112, 1985.

[54] D. R. Gonzales. M*CORE architecture implements real-time debug port based on Nexus consortium specification, 1999. http://www.nexus5001.org/archive/pdf/northcon99.pdf.

[55] M. M. Gorlick. The flight recorder: an architectural aid for system monitoring. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 175–181, New York, NY, USA, 1991. ACM Press.

[56] N. A. Gray. Comparison of web services, java-rmi, and corba service implementation. In *Fifth Australasian Workshop on Software and System Architectures*, Melbourne, Australia, 2004.

[57] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2005.

[58] R. Gupta and Y. Zorian. Introducing core-based system design. *IEEE Design and Test of Computers*, pages 15–25, 1997.

[59] W. Heinzelman, A. Chandrasekaran, and H. Balakrishnan. Application-specific protocol architectures for wireless networks. *IEEE Transactions on Wireless Communications*, 1(4), October 2002.

[60] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18(1):6–14, 2004.

[61] E. V. Hensbergen and R. Minnich. Grave robbers from outer space : Using 9p2000 under linux. In *Proceedings of Freenix*, 2005.

[62] A. B. T. Hopkins. Debug support strategy for systems-on-chips with multiple processor cores. *IEEE Trans. Comput.*, 55(2):174–184, 2006. Member-Klaus D. McDonald-Maier.

[63] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM Press.

[64] GNU Hurd. http://www.gnu.org/software/hurd/hurd.html.

[65] Ieee 802.15.4 standard. http://standards.ieee.org/getieee802/802.15.html.

[66] C. T. Inc. Mote in-network programming user reference. http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf.

[67] Intel: Product brief of intel ixp2850 network processor. http://www.intel.com/design/network/prodbrf/25213601.pdf.

[68] International technology roadmap for semiconductors 2001 edition: Design, 2001. http://public.itrs.net/Files/2001ITRS/Home.htm.

[69] Jini architectural overview. http://www.sun.com/software/jini/whitepapers.

[70] L. L. Johannes Gehrke. Guest editors' introduction: Sensor-network applications. *IEEE Internet Computing*, 10(2):16–17, Mar/Apr 2006.

[71] Ifs kit - installable file system kit. http://www.microsoft.com/whdc/DevTools/IFSKit/default.mspx.

[72] P. Kang, C. Borcea, G. Xu, A. Saxena, U. Kremer, and L. Iftode. Smart messages: A distributed computing platform for networks of embedded systems. *The Special Issue on Mobile and Pervasive Computing, the Computer Journal*, 2004.

[73] T. J. Killian. Processes as files. In *USENIX Association. Proceedings of the Summer 1984 USENIX Conference*, pages 203–207, Berkeley, CA, USA, 1984. USENIX.

[74] S. R. Kleiman. Vnodes: An architecture for multiple file system types in sun UNIX. In *USENIX Summer*, pages 238–247, 1986.

[75] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[76] V. Kottapalli, A. Kiremidjian, J. Lynch, E. Carryer, T. Kenny, K. Law, and Y. Lei. Two-tiered wireless sensor network architecture for structural health monitoring. In *SPIE Annual Meeting*, 2003.

[77] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 81, Washington, DC, USA, 2003. IEEE Computer Society.

[78] T. M. Levergood, A. C. Payne, J. Gettys, G. W. Treese, and L. C. Stewart. Audiofile: A network-transparent system for distributed audio applications. In *Proceedings of the USENIX Summer Conference*, 1993.

[79] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM Press.

[80] S. Li, Y. Lin, S. Son, J. Stankovic, and Y. Wei. Event detection services using data service middleware in distributed sensor networks. In *Second International Workshop on Information Processing in Sensor Networks*, 2003.

[81] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *The International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2005.

[82] J. Liu, M. Chu, J. Liu, J. E. Reich, and F. Zhao. State centric programming for sensor and acutator network systems. *IEEE Pervasive Computing*, 2(4):50–62, 2003.

[83] C. Locke. Styx-on-a-brick. http://www.vitanuova.com/inferno/rcxpaper.html.

[84] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1), 2005.

[85] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM Press.

[86] H. S. Mark Birnbaum. How vsia answers the soc dilemma. *IEEE Computer*, 32(6):42–50, 1999.

[87] S. J. J. J. K. P. R. N. B. S. Martin Burtscher, Ilya Ganusov. The vpc trace-compression algorithms. *IEEE Transactions on Computers*, 54(11):1329–1344, 2005.

[88] D. F. McMullen, T. Devadithya, and K. Chiu. Integrating instruments and sensors into the grid with c90ahmst-ima web services. In *The APAC Conference and Exhibition on Advanced Computing, Grid Applications and eResearch*, Gold Coast, Australia, 2005.

[89] A. Milenkovi and M. Milenkovi. An efficient single-pass trace compression technique utilizing instruction streams. *ACM Trans. Model. Comput. Simul.*, 17(1):2, 2007.

[90] R. Minnich. 9p2000 file system support for unix/linux/*bsd. http://v9fs.sourceforge.net.

[91] A library of plan 9 client software for unix. http://swtch.com/plan9port/.

[92] S. M. C. M. H. R. D. S. Morris, J. H. and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 1986.

[93] Ifs kit - installable file system kit. http://www.microsoft.com/whdc/DevTools/IFSKit/default.mspx.

[94] R. Newton and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[95] npfs : Library for writing 9p2000 compliant user-space file servers. http://sourceforge.net/projects/npfs.

[96] OMG Object Management Group. Smart transducers interface specification, 2003.

[97] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

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

[99] B. Pisupati and G. Brown. File System Interfaces for Embedded Software Development. In *Proceedings of the International Conference on Computer Design*, 2005.

[100] B. Pisupati and G. Brown. File system framework for organizing sensor networks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 935–936, New York, NY, USA, 2006. ACM Press.

[101] D. Presotto and P. Winterbottom. The organization of networks in Plan 9. In *USENIX Association. Proceedings of the Winter 1993 USENIX Conference*, pages 271–280 (of x + 530), Berkeley, CA, USA, 1993. USENIX.

[102] A. Rubini. Kernel korner: The "virtual file system" in linux. *Linux J.*, 1997(37es):21, 1997.

[103] Samba - opening Windows to a wider world. http://www.samba.org.

[104] A. D. Samples. Mache: no-loss trace compaction. In *SIGMETRICS '89: Proceedings of the 1989*

ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pages 89–97, New York, NY, USA, 1989. ACM Press.

[105] S. Shempler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system NFS version 4 protocol. Technical Report RFC 3530, Network Working Group, April 2003.

[106] C.-C. Shen, C. Srisathapornphat, and C. Jaikaeo. Sensor information networking architecture and applications. *IEEE Personal Communications Magazine*, 8(4):52–59, 2001.

[107] I. C. Society. Ieee standard test access port and boundary-scan architecture, 2001.

[108] E. Souto, G. Guimares, G. Vasconcelos, M. Vieira, N. S. Rosa, C. Andr, and G. Ferraz. A message-oriented middleware for sensor networks. In *Workshop on Middleware for Pervasive and Ad-hoc Computing*, 2004.

[109] P. Stanley-Marbell and L. Iftode. Scylla: a smart virtual machine for mobile embedded systems. *wmcsa*, 00:41, 2000.

[110] R. Stann and J. Heidemann. Rmst: Reliable data transport in sensor networks. In *Proceedings of the 1st IEEE International Workshop on Sensor Net Protocols and Applications (SNPA)*, 2003.

[111] J. Sugermann, G. Venkitchalam, and B. H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, USA, 2001.

[112] Q. C. Y. C. D. E. J. G. S. G. L. G. T. H. S. K. L. L. S. S. J. S. R. S. A. W. T. Abdelzaher, B. Blum. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, pages 582–589, 2004.

[113] S. Tilak, B. Pisupati, K. Chiu, and G. Brown. A File System Abstraction for Sense and Respond Systems. In *Proceedings of the Workshop on End-to-End Sense and Respond Systems*, 2005.

[114] TinyOS: An operating system for wireless sensor networks. http://www.tinyos.net.

[115] B. L. Titzer and J. Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[116] J. Turley. Nexus standard brings order to microprocessor debugging. http://www.nexus5001.org/nexus-wp-200408.pdf.

[117] Upnp - universal plug and play forum. http://www.upnp.org.

[118] C.-Y. Wan, A. T. Campbell, and L. Krishnamurthy. Psfq: a reliable transport protocol for wireless sensor networks. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 1–11, New York, NY, USA, 2002. ACM Press.

[119] D. Wilner. Windview: a tool for understanding real-time embedded software through system vizualization. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 117–123, New York, NY, USA, 1995. ACM Press.

[120] A. Winfield and O. Holland. The application of wireless local area network technology to the control of mobile robots. *Journal of Microprocessors and Microsystems*, 23(10):597–607, 2000.

[121] P. Winterbottom. Acid: A debugger built from a language. In *Proc. of the Winter 1994 USENIX Conference*, pages 211–222. USENIX, 1994.

[122] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2003. IEEE Computer Society.

[123] W. Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41st annual conference on Design automation - Volume 00 (DAC '04)*, pages 681–685. ACM, 2004.

[124] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the java system. *Computer Systems*, 9(4):265–290, 1996.

[125] A. Woo, S. Seth, T. Olson, J. Liu, and F. Zhao. A spreadsheet approach to programming and managing sensor networks. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 424–431, New York, NY, USA, 2006. ACM Press.

[126] Crossbow technology. http://www.xbow.com.

[127] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 2002.

[128] Y. Yao and J. Gehrke. Query processing for sensor networks. In *Proceedings of the 2003 CIDR Conference*, 2003.

[129] G. Zimmermann, G. Vanderheiden, and A. Gilman. Universal remote console - prototyping the alternate interface access standard. In *Center on Disabilities Technology and Persons with Disabilities Conference*, 2000.

[130] Y. Zorian. Embedding infrastructure ip for soc yield improvement. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 709–712, New York, NY, USA, 2002. ACM Press.