File System Interfaces for Embedded Software Development

Bhanu Pisupati & Geoffrey Brown
Indiana University
Department of Computer Science
Bloomington, Indiana, USA

Abstract

We present a scalable architectural approach which aims to simplify embedded software development by supporting key development tasks like debugging, tracing and monitoring. Our approach uses filesystem interfaces to represent various entities in the embedded system hierarchy that are merged and exported to clients using a messaging protocol. Unlike many JTAG based debug solutions, ours is compositional and supports heterogeneous, concurrent debugging.

1. Introduction

The predominant trend for embedded systems to use SoC devices comprising multiple, diverse cores has brought in its wake a new set of constraints and demands such as the need to support concurrent debugging, tracing and monitoring in a heterogeneous environment. While solutions have been proposed[4],[5] that try to adapt JTAG based approaches for multi-core system on chip debugging, these are not scalable and also depend upon proprietary knowledge about debug interfaces[7]. The Nexus standard[2] aims to define a uniform debug interface that can inter operate with diverse types of cores but does not address multi-core debugging. Traditional software development processes are also ill-equipped to handle the partitioning of responsibilities in creation and application of programmable platforms that is central to the SoC design task[8]. JTAG-based approaches are limited in their ability to expose interfaces at higher levels of abstraction and correspondingly often require disclosing of internal design details.

We believe that a scalable solution for programming and debugging systems built from SoC's that addresses the needs described above can draw from distributed programming principles using chip-level file systems as distributed building blocks. These are composed together to create higher level file systems that provide a portable, high level interface which hides the heterogeneity that may exist inside the SoC. The architecture supports concurrent access to cores inside the SoC and can support a diverse set of operations including debugging, tracing, monitoring and configuration. Deeply embedded functional blocks may be managed, programmed and monitored using a familiar file interface exported over communication links. Conventional access methods including Nexus, in contrast, typically piggy back upon IEEE 1149.1 (JTAG) scan chains which requires physical access to a dedicated hardware debug port. Our idea is largely inspired by the Plan 9 operating system[9] which extended the file metaphor to device control with file system operations providing a common interface.

To support chip-level file systems for SoC's we are investigating the model shown in Figure 1. The figure depicts a heterogeneous chip-level processing environment with two conventional processors and a custom DSP. To manage a chip-level file system for such an SoC our model introduces a support processor executing portable software modules that implement the file system. This file system is exported off chip and can be used directly by an end application (such as a debugger) to interact with the system or combined with file systems from other chips to create higher level file systems. An example file system namespace that supports software debugging is shown below.

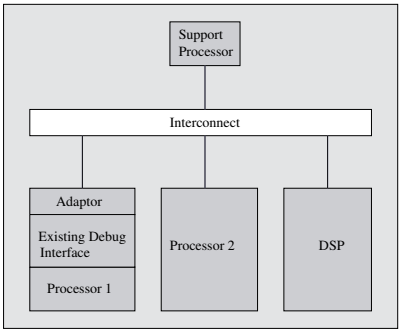

Figure 1. Model for chip-level filesystems

Proceedings of the 2005 International Conference on Computer Design (ICCD'05)
0-7695-2451-6/05 $20.00 © 2005 IEEE

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

Each processor core is represented as part of the namespace with a separate directory which contains files to access registers and memory, check status and perform basic run control. The heterogeneity among the cores in the SoC is concealed by the uniform file interface used to interact with them.

2 Architecture

The architecture that we propose is shown in Figure 2. At its core is the embedded file system (EFS) that encapsulates a part (chip/board) of the system using a file interface. Various embedded file systems may be integrated in the implementation of a host file system (HFS) or directly used by a host application. The EFS may itself be built using specialized file systems provided by sub components. In any particular implementation the various pieces in the system architecture are optional. For instance, a design may leave out the specialized file systems and have the EFS interact with sub components directly. Similarly, a design may do without a HFS whose role may be subsumed by the host application software. Specialized file systems enable computationally lightweight devices to implement a basic file interface that provides limited, device specific functionality while depending on a higher level file system to implement more sophisticated features (such as managing client state). The host and the embedded sides communicate using a message based protocol called 9P[9], whose message set essentially consists of encodings of file operations.

The architecture supports the split file system approach whereby debug functionality can be split across file systems (or applications) on the host and embedded sides. This enables confining of proprietary information within the EFS implementation thereby concealing it from the host side. Since complexity of embedded debug functionality and performance while debugging are often directly proportional, split file system paradigm helps achieve the required tradeoff between limiting complexity and obtaining desired performance through suitable partitioning across the host and the embedded sides. The architecture supports concurrent debugging of cores by using the embedded file server's ability to support multiple outstanding (blocking) requests from the host side. As requests are issued to the EFS, it directs

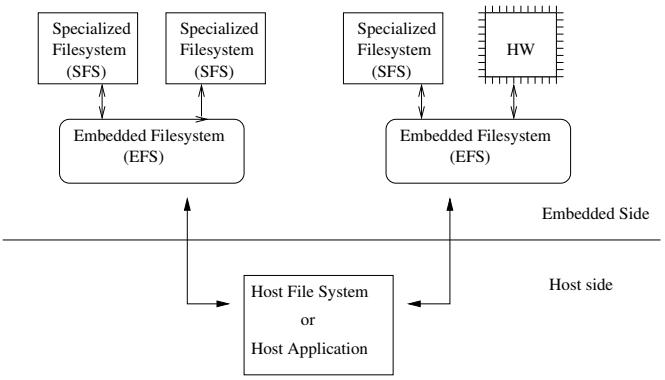

Figure 2. Architecture

them to the appropriate component and immediately returns to process new incoming requests.

As part of our implementation effort, we have developed a skeleton EFS module that handles processing of incoming 9P requests, manages client state and provides a framework for file namespace generation within the EFS. In the creation of a design, the namespace can be built by attaching devices to various parts of file tree. For each device users simply need to provide a set of call back routines corresponding to various file operation types along with a table driven directory structure listing the files that go with the device. The EFS module automatically generates and manages the namespace based on the devices added. Size of the EFS module implementation for the 32-bit Nios architecture[3] was about 10 KB. We have also implemented specialized filesystems for low-end devices with minimal RAM, which required about 150 lines of code and 800 bytes of code memory for the 16 bit MSP430 microcontroller architecture from TI.

3 Application

In this section we demonstrate the application of our architecture to concurrent debugging of multiple TI MSP430 microcontrollers and also to support on-chip tracing. Our prototype, shown in Figure 3, consists of two TI MSP430 microcontrollers being concurrently debugged using an embedded file system, along with an on-chip processing core generating simulated trace data. The support processor interacts with the MSP430's using their JTAG emulation feature, that allows run control of these devices using the JTAG port. The objectives of the section are to demonstrate use of the filesystem model to support concurrent debugging and tracing, to show flexibility in partitioning that the approach offers and compare performance to that obtained with conventional debugging techniques.

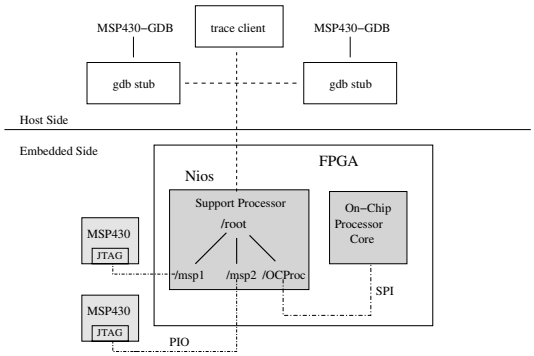

Figure 3. Prototype for Debugging & Tracing

3.1. Concurrent Debugging

The host side of our prototype includes two instances of the gdb debugger that concurrently debug code executing on the microcontrollers. Each instance interacts with its respective device by means of a stub that mounts and uses the EFS to execute operations requested by and on behalf of the debugger. Even when file requests result in blocking operations on individual devices (as with waiting for breakpoint hit), the file system continues to accept and process requests, while waiting for the earlier one to complete. In the context of our prototype, this means that the two debuggers can interact with the microcontrollers concurrently and not get in each other's way, with their requests being completed in parallel.

3.2 Partitioning of Debug Functionality

Software libraries available for debugging MSP430 devices using JTAG emulation provide high level routines (henceforth known as JEL for JTAG Emulation Library) to achieve run control. These routines in turn are 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 on the device. Conventional debugging approaches implement both the JEL and HIL on the host-side with the HIL routines accessing the JTAG pins through a simple connector such as a parallel cable. In contrast, the split filesystem model offers great flexibility in partitioning debug functionality between the host and embedded sides. We implemented three of these partitionings in our prototype.

EFS-centric partitioning implements the JEL as well as the HIL functionality inside the EFS. The embedded side exports a 'rich' file namespace (similar to that presented in Section 1) with files to directly access registers/memory and control execution, which conceals details of the underlying debug infrastructure (JTAG in this case). However, the partitioning increases the complexity of the EFS (75 KB in size) by implementing all the debug functionality within it.

Host-centric partitioning implements the JEL inside the host application (gdb stub) and the HIL within the EFS. The namespace exported as part of EFS includes files to access the JTAG registers and signals of the microcontrollers. The resulting EFS is simple (20KB in size), but each high level operation (such as register read) involves numerous file operations on the EFS which compromises performance.

Hybrid partitioning 'pushes' down frequently used operations into the EFS while implementing others on the host, thereby improving performance. Select operations with stringent timing requirement (such as flash programming) can be pushed into the EFS as well.

3.3 Performance

The table below compares the performance of our prototype using EFS-centric partitioning with the standard approach used for MSP430 debugging based on host resident JTAG emulation software communicating with the device through a parallel connection.

	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, which we believe may be attributed to the algorithm that gdb uses in 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 alternate loading strategies.

3.4 Tracing

Our prototype addresses two issues key to on-chip tracing, namely on-chip storing of generated trace data and mechanism to access it off-chip. The support processor uses DMA to store the stream of trace data generated by the on-chip core into memory buffers, thereby keeping itself free to manage the file system. The trace data is accessed off-chip by reading the appropriate file in the exported filesystem. The fileserver is able to stall a read request until trace data is available while continuing to process other requests. A significant advantage of our approach is that the support

processor can be used to implement more complicated trigger and filtering features over and above those provided by individual cores on the SoC, including cross triggering.

4 Discussion

While the prototype presents a simple design it illustrates ideas that form the core of this paper that are applicable to SoC based design. Using our approach, debugging functionality may be partitioned in the system at different levels. In contrast conventional JTAG based approaches impose a rigid partitioning that is extremely limiting. Our approach offers chip providers the flexibility to expose interfaces at different levels of abstraction, which can be used to conceal proprietary information about the internal design.

The prototype also demonstrates concurrent debugging of multi processor cores using the filesystem approach. The key is for the fileserver and messaging protocol to support multiple outstanding requests along with server handing blocking operations appropriately. The idea can be extended to a multi processor SoC scenario with modest additional chip-level resources devoted to implementing a support processor. In contrast, with JTAG-based approaches the scan chain is a shared resource among the different cores which means concurrently accessing them is non trivial. Heterogeneous debugging also becomes tenable as the filesystem implementation can be used to mask portions of the heterogeneity and provide a more uniform interface.

While a filesystem interface may seem too heavyweight to support time critical operations common in the embedded domain, the prototype demonstrates use of the split filesystem idea in supporting such scenarios. The prototype also illustrates the compositional nature of our approach with the debug interfaces from the various components being combined under one single filesystem namespace to enable system level debugging.

5 Related Work & Extensions

As part of our research, we have explored applying the file system metaphor to managing distributed, loosely coupled embedded devices such as sensor networks[10]. As mentioned earlier, considerable work relating to concurrent debugging in a SoC environment has focused on access of multiple TAP controllers on a single chip in a IEEE 1149.1-compliant manner[5, 4]. Analogous to our idea of introducing a support processor to aid embedded debugging, proposals have been made regarding use of Infrastructure IP[11] within SoC's to facilitate testing and yield improvement. The FUSD[6] framework allows devices to be managed using user space filesystems and has been used by projects[1] in the embedded domain to represent various resources as files.

References

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

[2] The nexus 5001 forum standard for a global embedded processor debug interface, 2004. http://www.nexus5001.org.

[3] Altera. http://www.altera.com.

[4] 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.

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

[6] Fusd: Framework for user-space devices. http://sourceforge.net/projects/fusd.

[7] 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.

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

[9] 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.

[10] 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.

[11] Y. Zorian. Embedding Infrastructure IP for SoC Yield Improvement. In *Proceedings of the Design Automation Conference*, 2002.