Debugging by Visualizing Communication on a Parallel Embedded System

A Report

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Paul Nathan

Version: Tuesday 8th January, 2013 07

Major Professor: Robert Rinker, Ph.D.

Authorization to Submit Report

This report of Paul Nathan, submitted for the degree of Master of Science with a major in Computer Science and titled Debugging by Visualizing Communication on a Parallel Embedded System has been reviewed in final form, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor	______________________ Robert Rinker	Date __________
Committee Members	______________________ Clinton Jeffery	Date __________
	______________________ Gregory Donohoe	Date __________
Department Administrator	______________________ Gregory Donohoe	Date __________
Discipline's College Dean	______________________ Lawrence Stauffer	Date __________

Final Approval and Acceptance by the College of Graduate Studies

______________________ Date __________

Jie Chen

Abstract

A mechanism of debugging multicore/parallel embedded systems, called *targeted trace debugging* is hypothesized in this report. We explore the implementation of this mechanism on the XCore quad-core processor and present the challenges encountered with the implementation. An algorithm, the *Targeted Trace Algorithm*, is developed to analyze the data returned from the targeted trace and place it into a set of graphs describing the execution of the program. We conclude by describing the limitations of our experiments and next steps for targeted trace debugging.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This report presents a mechanism for debugging embedded multicore systems through the visualization of the order of communications. The report terms the mechanism as *targeted trace debugging*, as it provides a limited (targeted) segment of the program for analysis via a trace mechanism. The mechanism of the targeted trace respects the special constraints that embedded systems place on resources, both in time and in memory. This report constructs a flowchart based upon the targeted trace to visualize the communication as an aid to program understanding.

1.1 Problem Statement

In the modern computational landscape, multicore processors enable a parallel computational speedup with significantly less cost in both silicon and wattage than architectural redevelopment or enhancement of the underlying transistor technology [?]. Due to the physical limitations of current heat dissipation technology, multiple cores on a chip are becoming the preferred method of expanding computational capability. This is holding true in the embedded realm as well.

Parallel programming has only been historically available for scientific research in the form of supercomputers and Beowulf clusters. However, due to developments in transistor technologies available today, current computer systems span the gamut from minuscule single-processor singing greeting cards [?] to the large corporate cloud networks with thousands of nodes [?]. Thus, modern chip fabrication technologies has made parallel programming available for most systems, including the domain investigated in this report, embedded systems.

Embedded systems present several differences from non embedded systems. They are designed for a specialized purpose [?]; embedded software is developed on a separate system (the *host*) and loaded onto the embedded memories such as RAM, ROM, or FLASH [?]. The separation between execution environment and development environment inhibits conventional debugging technologies such as viewing one program's memory while using another program, or even having a file system for information logging. This presents an interesting challenge, to obtain sufficient information regarding a computer's state from an entirely different development system and to understand the cause of a particular program behavior on the computer system under analysis.

The widespread adoption of multicore development has brought the problem of parallel programming to average embedded software engineers. This was due to the historically limited availability of more advanced processors [?]. Historically parallel programming has been one of the more difficult areas of programming, and moving to the embedded arena has not diminished its difficulty. Today, multicore is present in embedded systems and practical programming requires understanding the intricacies of parallel programming. One problem the multicore world has made evident is that the software tools for multicore and multiprocessor development are not as sophisticated and usable as their counterparts for non-parallel software, including debugging.

One of the key difficulties in parallel programming involves distributed computing issues; examples include interleaving of messages between cores, latencies of execution, clock skew between chips, communications time lag, dropped packets, and dead communication nodes. Due to these time and communication issues, it has been demonstrated that correctly ordering events is impossible without a clocking mechanism [?]. Prior work in the field of parallel debugging demonstrates the viability of recording information and building analyses after the execution of the program. Consequently, this report examines the causal relationships in parallel programs and devises a mechanism to build a model of the execution path.

A modern full-featured operating system incorporates features such as native operating system threads, memory management, hardware abstraction layers and security models. However, due to limited resources, many devices in the embedded domain do not have the capability to support these facilities. Embedded processors are becoming increasingly prevalent, and therefore, they present an opportunity for study in the hopes that interesting properties may be discovered.

However, this report limits its study to certain characteristics caused by to the wide variety and mechanisms of embedded systems. This report focuses specifically on processors without virtual memory, sophisticated boot procedures, or other capabilities. Such a focus was made to exclude non-essential features that may obscure the essential problem being studied.

1.2 Debugging

Software debugging has a long history, beginning at the earliest mainframes and continuing to the present day. Numerous mechanisms have been created to obtain information regarding the system under examination. Difficulties became apparent when moving to multiprocessor/multithreaded systems, and in the 1980s a variety of approaches were documented. These approaches are described in Chapter 2. This report takes a *post-mortem abstracted* approach, choosing to use a previously executed run's information, and only selecting parts of the run for interest.

The typical industry standard for embedded systems is a hardware-software interface, the JTAG (Joint Test Action Group) interface being the most commonly used. JTAG devices give a high degree of fine-grained visibility into the system. However, this visibility using the hardware carries with it some stringent requirements. First, a hardware interface device, a.k.a dongle, is necessarily required for the software engineer to use the JTAG protocol. Second, the chip that the software runs on must support the JTAG hardware protocol. Third, the board the chip is attached to must support a JTAG breakout pin configuration. Fourth, the dongle itself, a special purpose piece of hardware, can be expensive, out of production, or otherwise unavailable. In contrast to these issues, software can be reprogrammed and scripted, allowing the user flexibility in analysis and probing of the system in the tasks of debugging and other state-inspection tasks. Thus, this report studies software-only debugging systems.

This report develops an algorithm to determine the set of possible execution paths between a distributed system system can take, based upon the communication nodes recorded. This algorithm, called the Targeted Trace Algorithm, is described in Chapter 3.

1.3 Device Selected

Investigating the hardware platforms available, the XMOS XCore device was found. It is a simple inexpensive processor with multiple cores and was selected over its competitors for the following reasons: it was extensively documented, it uses the

GNU open source tool chain allowing examination of the internal tools, and allowed fine-grained control by the programmer.

Several types of test-bed with the XCore can be either used or developed in this report project to experiment with.

1. Manual tracing via logging to a file or other buffer. This approach requires gross source code modification in order to enact the logging. This approach will (1) introduce source code churn, (2) add potential bugs, (3) introduce uncertainty regarding timing of the final product. It will also produce a record of exactly what was desired by the engineer.

2. Software tracing via a debug interrupt. This requires a chip that correctly traps and returns to the place of execution. This will (1) slow down the execution rate of the program, (2) require some amount of storage and (3) require a specialized communication protocol.

3. Simulation tracing via the simulator. This requires (1) a high-fidelity simulator and (2) an accurate reproduction of the input and (if needed) output for the executed program. Further, it does not incorporate the possibility of jitter and other reality-based analog issues. Simulation does provide a basis for analysis without modifying either the source code or the adding of special hardware to the development system.

Manual tracing poses sufficient difficulty that it is not practical for research in debugging. For this particular work, the debug interrupt approach was initially selected. For the reasons more fully described in Section **??**, development was moved to the XCore simulator.

1.4 Summary

This report describes a tool and an algorithm to designed to inspect the behavior of the software under observation using the XCore platform. The scheme for a visualization of communication was selected for the several reasons. First, the amount of data to understand is potentially very large. Second, the communication primitives form a causality of events, giving insight into the execution. Third, communication is a relatively small subset of the total information to be displayed. Taken together, these qualities allow insight into the software under observation. The method for inspecting behavior results in a flowchart-like diagram which visualizes the causal relationships between processors.

In order to fully understand the need for this report, its historical position, and its development, further elucidation is needed. The remainder of this report is organized along the following lines:

1. Background, wherein the problem of debugging is reviewed.

2. Implementation, wherein the implementation environment is described.

3. Experiments and Results, wherein the result of the implementation is described.

4. Conclusion, wherein the results are evaluated and future directions are discussed.

Chapter 2

Background

Debugging's key idea is to allow the user to inspect the state of the program at a given point of execution. The current state of debugging has a long and rich history stemming from the earliest use of computational devices. This report considers the history of debugging and frames the targeted trace mechanism in the historical context. First, the common approaches are considered, definitions are given, and then a brief historical review of debugging is performed in chronological order.

The task of the debugging system is to assist the user in gaining understanding of the software system at a given point in time. Suppose a system was developed that could present the entire state of the system. Such as system might not be called a debugger. However, if harnessed to the purpose of removing bugs, it would become a debugging tool. This carries with it the implication that anything used to denote system state can be leveraged into being a debugging tool.

The essential problems in debugging multicore embedded systems have been previously identified to be observability, repeatability, and control [**?**, **?**]. This report uses these as a structure to examine the debugging approaches taken.

Being able to observe the state of the system under debug (or infer the state) is intuitively required in the debugging task. Embedded systems complicate the observability task, as some communication mechanism must be formed to describe the system state to the user.

The approach of cyclic debugging (or forming a hypothesis, executing, and reviewing results to confirm or deny the hypothesis) is common, which requires repeatability[**?**]. It would be a poor debugger indeed which could not assist with being able to replicate the system.

When the behavior can be observed and repeated, bringing the system under control and varying the state to determine the effect on the behavior will assist the user in determining the undesirable state.

This report will generally define a "feasible", or an "effective" debugger as a system able to determine some information regarding the internal state of a system under investigation. The space of possible errors is very large, and the presentation of debugging adds the nuance of human interfaces. This report explicitly limits itself to obtaining information from the system under investigation without studying the human factors involved.

2.1 Terminology

In order to have a working base of common terms, several definitions are given h3ere. In general, the terminology established by Huselius in his 2002 report is followed here [**?**].

Definition 2.1. *A debugger is a program that is designed for the purpose of finding bugs in a software system.*

Definition 2.2. *Parallel, concurrent, and distributed are all defined to have the same sense in this work, that of multiple processors executing instructions at the same time.*

Definition 2.3. *Embedded systems are systems designed for a single purpose; an identifying feature is that they are not designed to be re-programmable by their users in any usual sense. They are also characteristically limited in resource, both in memory and in execution speed.*

Definition 2.4. *Observability is the ability to see the state of the system either in a total or limited fashion. It will induce a probe effect.*

Definition 2.5. *Repeatability is defined here as the ability to replicate the causality sequence of communications between parallel programs.*

Definition 2.6. *Control is defined to be the ability to adjust the state of the program under debug at the user's will.*

Definition 2.7. *The probe effect is the perturbation of the system inevitably caused by the debugging system injecting code into the system under test [**?**].*

Definition 2.8. *Non-determinism is defined to be the property that, on multiple executions of a system, when given the same inputs, different outputs result.*

Definition 2.9. *Breakpoints are defined to be a mechanism that stops the normal flow of a program's execution and triggers code in a debugger process.*

Definition 2.10. *Trace is defined to be a mechanism that records execution of a program step by step. Step increments and abstractions may vary.*

Definition 2.11. *A single-step is a mechanism that steps the executing program by one (single) step. The step increments and abstractions may vary.*

Definition 2.12. *Memory dumps are copies of a process's memory at a given point of time. If termed a core dump, it implies that the program catastrophically failed and stopped executing; the core dump is the state of memory at that point in time.*

Definition 2.13. *In-execution debugging is defined to be debugging that takes place during the execution of the program and requires interaction with the user.*

Definition 2.14. *After-execution debugging is defined to be debugging that occurs after the program has ended (either normally or abnormally).*

2.2 Historical Review

Research must be placed in the historical context it occurs in, both to draw on previous ideas for inspiration and for contrast. Presented below is a rough sketch of the history of software debugging to the present day,

2.2.1 Path to The Present

The first glimpse of the problem of debugging can be see in the Notes of Lady Lovelace while she corrects them - she described the problem of correcting her notes in 1843 as "troublesome work"[?]. However, it was left for another century to encounter the problem fully developed.

Debugging made its first practical impression in the software development with the EDSAC (Electronic Delay Storage Automatic Calculator) in England by Wilkes, Wheeler, and Gill. Campbell-Kelly writes that EDSAC was the first computer to have non-trivial programs executed upon it and at that point, the EDSAC group discovered debugging at that juncture [?].

Wilkes, Wheeler, and Gill record that single-step, trace, post-mortem memory dumps, and breakpoint were all used on EDSAC. Gill also remarked on an unimplemented idea for register trace via high-speed photography of the CRTs displaying registers [?, ?], noting that it would take enormous amounts of space to trace the registers of the computer.

With the base of studies derived from the EDSAC and the ENIAC (Electronic Numerical Integrator And Computer), the early mainframes evolved into production systems for defense work and major processing. One part of that work included developing a simulator with tracing, breakpoints and memory visibility [?]. IBM's 701 debugger techniques were primarily core dumps, printing and binary differences between the core dumps[?, ?], falling into the post-mortem family of debuggers.

In 1960, the progenitor of the PDP-1 computers was built at MIT, the TX-0. The TX-0 had the `FLIT` debugger, which allowed breakpoints in the Lisp forms[?, ?]. The builders of the TX-0 founded MIT and the `FLIT` was evolved into the PDP DDT debugger.

By the middle of the 1960s, early Lisp variants had developed sophisticated interactive debugging systems, including technologies such as conditional breakpoints and a sophisticated "break, edit, recompile, test, continue" ability. This particular system was reported for the Univac M-460 mainframe [?].

A post-execution approach was also developed in the middle of the 1960s for FORTRAN. A code coverage interpreter implemented with QUICKTRAN on the IBM 7040, allowing a user to have a deep understanding of the software in a given state, without the interactive approach used in the Lisp machines or the assembly style breakpoints[?].

Later in the 1960s, MULTICS was released, and Balzer built the EXDAMS post-execution analysis tool[?]. EXDAMS operated on PL/I, ALGOL, and FORTRAN. EXDAMS introduced the idea of a *history tape* - a trace of relevant data. This separated the execution from the analysis of an execution and allowed multiple analysis tools to be executed on the trace. The EXDAMs approach inserts a constant tracing of the program in the source code. This is in contrast to the Lisp interactive debugger built for the M-460, but logically follows from the compiled language technology instead of the interpreted Lisp approach. Instead of each source statement being evaluated in its turn (thus leading to source code based interactive debugging), logging each statement with the interpreter allows information to be stored and reviewed at leisure after the execution.

Rustin in 1971 [?] edited a collection of papers describing the current state of the art for debugging FORTRAN systems. The two key technologies were simulators to allow inspection and control of the system under review and a command language to perform user-controlled actions when given events were observed. These do not strictly fall into the interactive or post-execution camps but branch out into more subtle developments.

The 1970s was quieter on the original concept front in debugging but saw application development in various areas. In the mid-70s Satterthwaite reviewed the debugging situation and reports that post-mortems, profilers, and command languages (hearkening back to Rustin's collection) were the most common debuggers [**?**]; Tratner in 1979 reported, in agreement, that FORTRAN installations of the time did not permit interactive debugging [**?**]. Memory dumps were used as part of the IBM 360/370 debugging in this period[**?**].

The late 1970s and the early 1980s saw significant advances in VLSI technology, leading to massively parallel architectures, spurring such advances as the Connection Machine, programmed in *Lisp. The parallel architectures demanded debuggers capable of handling the sophisticated situations which arise in non-determinism.

In 1979, Leslie Lamport wrote a seminal paper on distributed systems and ordering time in them[**?**]. The net effect of Lamport's work was to conclusively demonstrate the effective non-determinism of distributed systems. Lamport's paper had profound implications for programs that needed to observe events across distributed systems, such as debuggers. A bare-bones statement of Lamport's work is that due to the time delay involved in communication between different processors, any given event may appear to happen later or earlier than it did. Therefore, only a partial ordering on causality may be applied, and only when a communication from another processor happens can causality be known. Lamport devised several schemes for tracking the partial order, now called Lamport clocks.

This presents a difficulty with the idea of a interactive debugger that presents a wholly true view: the time ordering of the system under analysis becomes disrupted by the user himself. This would be brought out in the 1980s by Leblanc, and McDowell[**?**, **?**]. However, notwithstanding the theoretical difficulties, various parallel debuggers have implemented interactive debugging since that time.

The Connection Machine's *Lisp's debugging facility used the Common Lisp interactive debugger restart mechanism, as it was built on Common Lisp [**?**, **?**]. Programming the Connection Machine required an interface machine that communicated the requests from the user to the parallel machine. This architecture would be repeated routinely in other parallel systems. A similar interface was present in miniature with the Multibug interactive debugging system, where a Unix host was connected with small embedded-type systems [**?**].

Taking a step back in time from the late 1980s and looking at the smaller computers of the late 1970s and early 1980s - the microprocessors and real-time systems, they are a mixed bag for debugging and programming: working at a relatively primitive level; hexadecimal and assembly was still common in industry [?]. Titus presented a non-symbolic interactive debugger for the 8080 computer[?]. The BASIC interpreter was available on some systems, however, and a book of the era for it recommends the non-interactive print statements as a means of debugging.[?].

Early in the 1980s, the transputer was designed and implemented by INMOS, which melded the embedded and parallel worlds. The transputer was a network of small microprocessors that could be joined to form a larger computing device. The Transputer used the parallel language Occam; the existing debugger shipped with the transputer turned out not to be parallel aware and, in the early 1990s, an interactive replay debugger was created for the Transputer[?].

A trace system using compiler hooks in Modula on a simulator was prepared around this time by Hill[?]. This system was unusual for the time in that it was operating in the constraints of real-time systems and the amount of memory available to store information. One key difference in what Hill did is that instead of tracing statement executions (e.g, as Balzer did in EXDAMS), it traced blocks of code, e.g., BEGIN/END pairs.

Garcia-Molina used a similar architecture as the connection machine distribution system with one master node and controlling other nodes. His works assumed that communications will be monitored. His system used a trace in a circular buffer to allow recording to proceed. In his work, it is partially traced and partially interactive; a stop-the-world command can be issued[?].

Some interest in real-time distributed systems was present at that time; Tokuda *et al.* developed a tracing probe for the ARTS operating system[?]. This debugger used an always-installed monitor interface which dispatched information regarding the scheduling decisions. The information was graphed for the user's perusal.

LeBlanc introduced the idea of Instant Replay, which traces the execution of a program and obtains sufficient information to perform cyclic debugging simulations after the execution and checkpointing external information[?].

Miller developed a behavior-abstraction type of debugger, similar to [?] around a flow of the program based on static analysis[?]. Logging of small events was performed and interrelated with the static flow graph.

McDowell and Helmbold in 1989 connected the dots from Lamport to distributed debugging, linking a lack of global synchronized clock to the inability to correctly determine order.

Cheung *et al.* in 1990 summarized debuggers of the time as being one of database-, behavioral abstraction-, or artificial intelligence-driven. According to Cheung, the fundamental issue was managing the amount of information in order to understand the behavior of the system.

A common implementation of parallel operation in the 1990s was the (MPI) Message Passing Interface standard. Debugging with MPI is touched on here to highlight popular tools in the 1990s in the field of parallel debugging. However, as the 90s moved forward, the conception of concurrent debugging maintained the two typical approaches of interactive and post-execution. Debuggers from academia appeared to slant towards post-execution debugger mechanisms; replay or analysis[**?**, **?**, **?**].

On the industrial side, interactive debuggers seemed to be more popular, e.g., the *Lisp debugger. Sun Microsystems developed the Prism debugger based on work in the Connection Machine; it used the general architecture of the Connection Machine and the Multibug; however, now instead of a host machine feeding other machines, it was a host process feeding other processes. Prism debugged FORTRAN- and C-based systems. One of its key attributes and approaches was attempting to visualize the system and its communications[**?**].

Similarly, in 2001, `net-dbx`, a Java-based interactive debugger for MPI was announced [**?**]. Using `net-dbx`, the user attaches to compiled programs and can interactively manipulate them with breakpoint, etc.

Hyder developed a theory of concurrent debugging in 1995 in an effort to unify the various debugging approaches and techniques[**?**]. His dissertation covers both a implementation and a theoretical approach. His practical framework requires that the debugger record events and their orderings using a Lamport clock in order to allow a correct reconstruction of execution. Hyder's theoretical contribution was to provide a unified notation and conceptual framework for debugging.

In 2002 and 2003, two separate reviews of extant states of the art were released, by Huselius and Metzger. Huselius reported on the state of the art in general and Metzger focused on industrial tools. Both authors reported that debugging schemes were either interactive mechanisms oriented around breakpoints, logging schemes, or post-mortem dumps.

2.2.2 The State of The Art

Parallel debugging today is often oriented towards massive networks of computers: "cloud-scale" computing, bringing with it new challenges[**?**]. One of the mechanisms used to handle the large amount of debugging data is the classic trace approach, now dispatching trace information into a database system[?, ?].

In the current embedded parallel debugging environment, several trends are asserting themselves. One relatively novel approach is to leverage formal methods to build a model checking system: such a system requires the user to create a set of static constraints and expected inputs and outputs. The model checking system uses formal methods to check the expected model against the actual run[?, ?, ?, ?]. A variant on this technique is to use code generation from a domain model[?]

Another novel approach is to insert scheduling re-sequencers[?], and relax the storing of events [?, ?] for replay debugging. The goal is to show timing bugs that are caused by certain timing orderings. Therefore, the scheduling re-sequencer approach manipulates the system to search through the different scheduling ordering possibilities to reveal the bug for perusal by the user.

One recent novel idea is the concept of transferring the entire debugging model into a virtual file system, similar to the Plan 9 design[?]. The implementation was designed for Linux, requires installing GDB stubs, and was designed around system-on-chip devices.

The trace and replay debugging approach is still an active research area in the field of debugging[?]. Maeng *et al.* devised a limited scheme for real-time replay debugging based on deterministic execution[?]. Thane, Sundmark, and Huselius produced a number of works on replay debugging using an operating system[?, ?, ?, ?]. Moving into a more resource-hungry approach, virtual machines and simulators can be leveraged to analyze software in various fashion[?, ?].

A straightforward for after-execution/in-execution debugging is logging. Logging allows custom messages to be placed in a log, which the developer subsequently reviews. The approach can be classified into two styles: one style is to hand-tune the logging messages, manually adding and removing them as required (e.g., the common printf approach), or instead, using a logging framework. A logging framework can can write to a file, a database management system, or some other repository of information for later analysis and auditing. The log4j system [?] is an example of the current logging technology.

Unless a preprocessing system is used, logging mechanisms are left in the code image which is delivered to the consumer.

In-execution debugging systems primarily consist of the interactive debuggers such as gdb and Microsoft Visual Studio. Dynamic languages typically have debugging facilities as well; Perl and Common Lisp debuggers demonstrate examples of this

[?, ?].

gdb includes a mechanism to minimize the probe effect with their tracepoint system[?]. The GDB tracepoint system involves setting “tracepoints” at places in the code. Each tracepoint contains information on what information to sample, and when execution traverses over the tracepoint, then the sample is taken. The user is able to observe the tracing after the execution.

Common industrial practice today is to use hardware-based solution such as in-circuit emulators or JTAG. This report focuses on software based solutions.

2.3 Approaches

Historically, debuggers have broken into two main areas; interactive debuggers and post-execution debuggers. While several projects have worked towards unification of the two approaches or have more unusual approaches, most debuggers still fall into one of two approaches in 2011.

2.3.1 Interactive

Typically, interactive debugging involves freezing the state of the executing program and allowing the user to introspect into the ’live’ state of the program. It is the earlier of the two debugging approaches, referenced by Gill in his classic paper on finding faults in EDSAC programming[?]. Careful analysis and extra work is needed to ensure the state of the program is not corrupted by the debugging process. The program must be configured for viewing and modification, i.e., being run in debug mode by the kernel. The interactive approach is usually founded in the use of breakpoints and inspecting variable values, e.g., the watch list (a variable inspection mechanism) in Microsoft’s Visual Studio. The interactive approach has theoretical difficulties when implemented for distributed systems, however, debugging systems in the 1980s and the 1990s frequently used interactive debugging as a means to analyze state[?].

2.3.2 Post-Execution

Post execution is initially academically referenced in the late 1960s by Balzer in his EXDAMS system[?]. Balzer separated out the execution of the program from the reviewing the actions of the program. Decoupling the execution from the analysis allows a variety of analyses to be performed on the traces. Initially, post-execution debugging was focused on post-crash examinations of the state of memory when the program crashed. This memory dump can be reviewed for a snapshot at the particular state of failure. However, as research in the field developed, logging techniques and other similar mechanisms for observing executing program states developed. This was particularly spurred in parallel computing by the theory developed in Lamport's classic paper[?], describing the theoretical limit on being able to trust ordering of events in distributed systems.

2.3.3 Event Based

A less common approach similar to the post-execution approach was the event-based mechanism of having a dynamic set of debugger actions that could occur when the state of the system reached a certain point. The event approach is designed towards abstracting out particular instructions and allowing the user o act at a higher level of understanding in the program abstraction hierarchy. This goal was developed into domain-specific languages in different works[?, ?]. Another approach is to build "monitors" to observe events[?, ?, ?].

2.3.4 Other Research Areas

Several other research areas have been explored; typically using using static analysis as their founding point. The model checking approach (derived from the formal methods and type annotation body of work) has produced results in recent years[?, ?]. Another area is the "program slicing" line of work, started in the early 1980s and continuing into the present[?, ?, ?, ?]. Program slicing takes as its founding principle statically forming a model of what variables may be modified by what effects, and forming a "slice" or view upon these variables. This report is related to the idea of observing a subset of a program, but takes an entirely different approach rooted in manual inspection.

2.4 Attacking The Problem

It is plain to see by the amount of continuing work that debugging is not a solved problem, and continues to have study on it to the present day. The previously listed approaches each have trade-offs, sketched out below.

2.4.1 Problem Environment

This report's goal is to provide a debug tool for a embedded multicore system too small to have a operating system such as ThreadX or Linux. This particular area is interesting for several reasons; without the action of the operating system, the computer activity is straightforward to predict and reason about. Operating systems also provide a great variety of tools that simplify the debugging process. Multicore systems have become prevalent and parallelism is an unsolved problem in debugging, thus providing an interesting challenge. Embedded systems present an additional difficulty in that the computer under debug is entirely outside of the development software. These restrictions combine to present an interesting and unsolved problem. The debuggers discussed below are viewed in light of this system for tradeoffs.

2.4.2 In-Execution

In-execution debugging is typically done with breakpoints or single step, both in parallel or single-core debuggers. This requires interacting with the user. Interaction with the user creates a large probe effect on timing, which can affect correctness.

In-execution debugging also requires a communication resource between the system under observation and the user, further disrupting the system.

Finally, in-execution systems will not be truthful in their ability to observe time unless a synchronization system such as Lamport clocks is placed upon the communication system. In the general case, this requires affecting the communication packets, causing code churn in the program under test and adding a layer to the communication stack. Control semantics are affected by this synchronization problem as well.

2.4.3 After-Execution

After-execution systems have, as a general form, the decoupling of the software under observation and the analysis software. Some block of data is passed between the software under observation and the analysis system. This approach has been followed up in the literature with productive results as observed above. Replay systems usually receive data and attempt to "replay" the execution for the user. This requires a large amount of data. Stack trace or memory dumps on crash will provide only a snapshot of the system, and then, only when the program has catastrophically failed.

However, in an embedded system, the amount of data passed into the analysis system must be managed with care. If the data is too large or sampled too often it will produce a probe effect sufficient to disrupt the system. The nature of embedded systems preclude virtual machine checkpointing or large semantic infrastructure such as event-based systems.

Therefore, this report takes an approach of targeting only a small part of the system under investigation and analyzing it. This is similar to a replay analysis, but unlike replay systems, the amount of memory needed can be controlled by the user.

2.5 Hypothesis

The hypothesis of this report is that a *targeted trace mechanism* is a feasible approach for a software-only embedded multicore system.

In this work, a targeted trace mechanism is defined as an execution tracing scheme that only traces and records a bounded portion of the executing code, then renders in a viewer the execution trace in some fashion.

The viewer mechanism provides the needed window of observability into the system that debugging requires. Repeatability and control have been given up, but the viewer has the capacity to examine the system at leisure.

Using the targeted trace, the debugger does not consume the amount of memory a typical trace system requires. Nor does target trace require the extensive resources of full checkpointing.

Unlike a logging system, a trace mechanism has the capacity to provide a sufficient semantic description of the state of the system to cross-link the execution to the original source code. Since the trace mechanism is executed at machine speed, the delay induced by the debug probe is not nearly as significant as an interactive system, which must operate at human speeds.

However, while trace systems do have significant advantages, they do have certain disadvantages. They do incur a recording time. Therefore, in a hard real-time system, the trace time of the interrupt must be smaller than the free time in a given timeslot. Further, all trace mechanisms can use up significant amount of space. The targeted replay approach may not be sufficient to analyze a given bug.

Even considering the disadvantages, the targeted trace mechanism is a viable approach to debugging embedded systems which cannot be debugged using hardware methods. It does not suffer the memory problems of the checkpoint systems, the the abstraction problems of the formal models, the frugal information of the stack traces, the gross manual source code manipulation of the logging systems, or the time-disruption of the interactive debuggers. Nor does it incur the memory problems extant in a full replay system.

The hypothesis will be tested by constructing a sequence of XC programs for the XCore and determining the multiprocessing execution paths the particular run took.

Chapter 3

Implementation

This chapter presents the experimental environment, the architecture of the system used to test the hypothesis, and the implementation of an algorithm designed to implement the hypothesis.

3.1 XCore Architecture

The remote processor chosen for the project is the XCore XS1-G4[?, ?] produced by the XMOS company. The XCore is a quad-core system with separate memory and address spaces for each core. It provides a GNU-based environment for programming, with its processor-specific language having tools such as `gcc`, `gdb` and `objdump`.[?, ?].

The processor design that preceded the XS1 was the Transputer, produced by INMOS and developed by David May [?] . The Transputer architecture relied heavily on Hoare's Communicating Sequential Processes (CSP) approach[?].

The Transputer was an embedded system multiprocessor produced in the 1980s and early 1990s. The design of the Transputer was to have a network of small processors, linked together to accomplish larger tasks. It was programmed with the parallel language occam, which has evolved into the current occam-pi project [?].

Tony Hoare, inventor of the Communicating Sequential Process (CSP) model, advised the Inmos team on the development of the parallel language occam for the Transputer[?] and occam was initially based on the Communicating Sequential Process theoretical framework [?]. occam programs are modeled as a collection of processes, communicating over "channels"; each occam channel is a point-to-point connection. Programs are composed of statements with each "statement" being considered to be a process on its own. A set of sequence operations are built in to define flows of information, as well as a PAR operation to denote parallel logic. XMOS developed a parallel language XC which was derived from occam and the Inmos's team's approach.

The XCore was developed by David May and has a CSP-styled architecture. Each core has a number of threads (in the processor used in this study, eight threads were implemented); each core has its own private memory. Communications are accomplished by an inter-core message passing design between the cores on the chip, analogous to the CSP events[?] (see Figure ?? for the block diagram). The design for the XCore contains the ability to reference off-chip processing elements, called *nodes* [?]. The XCore further has multiple threads within a given core, and the processor-specific language, XC, enforces the constraint that communications are limited to the message passing approach. XC is briefly referenced in this section, and the next section will discuss it in further detail.

Programming on the XCore is accomplished in three layers of abstraction using the instruction set architecture and programming model; each layer can be programmed in software; higher layers of abstraction encapsulate blocks of lower layer functionality. This is graphically depicted by Figure ??.

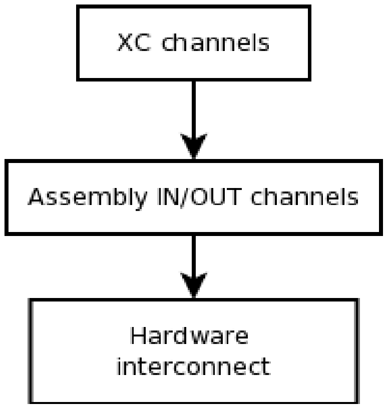

Figure 3.1: Layers of Communication

The lowest level of abstraction is the interconnect on the chip. This is achieved by the presence of a switch on the silicon. Communication between threads of execution is accomplished by sending tokens between threads on the switch interconnect via the XLinks communication channels. The data transferred between the threads is separated out into control tokens and data.

The next level of abstraction is the assembly that manipulates the links. The assembly code can execute the instructions `GET`, `FREE`, `TEST`, `IN`, and `OUT` on the communication mechanisms (called channels at this level of abstraction). `TEST`s, `IN`, and `OUT`s have "CT" or control token, equivalents. Control tokens are used as metadata to manage the channels. Threads will block on `IN`s to ensure that enough data is available.

The highest level of abstraction offered by the XCore tools is the XC language, a custom parallel language for the XCore. Again, here the message passing approach is employed, with the communication mechanism called channels. The XC language offers a simple notation for sending and receiving as shown in Listing **??**:

Listing 3.1: Sending and Receiving in XC

```
chan channel1;
int rx;
int tx;

// ...

channel1 <: tx;

// ...

rx <: channel1;
```

Internally, these correspond to an `OUT` and an `IN`. `<:` denotes a send, and `:>` denotes a receive.

3.1.1 Architecture

The XS1-G4 is a CISC processor with variable-length instructions. It has twelve general-purpose registers, four base registers for stack/data access, and nine registers for system purposes (exception, debug, kernel). All registers are 32-bit wide. The processor architecture is load-store approach. It incorporates instructions such as a `CRC`, a multiply-add, and a set of instructions to manage exceptions.

A significant component of the architecture is the access of the running instruction stream to the internal "COMM Switch" system, as diagrammed in Figure **??**. This is presented as an abstraction separate from the central processor; it takes inputs to control different internal behaviors, for example, setting the JTAG configurations, allowing access of thread registers in debug mode, and other internal control mechanisms. The interface for controlling the comm switch is to read or write registers in the from the processor using the `SETPS` and `GETPS` instructions. These instructions use two registers - one for the register number in the processor switch and one for the value to read in or write out.

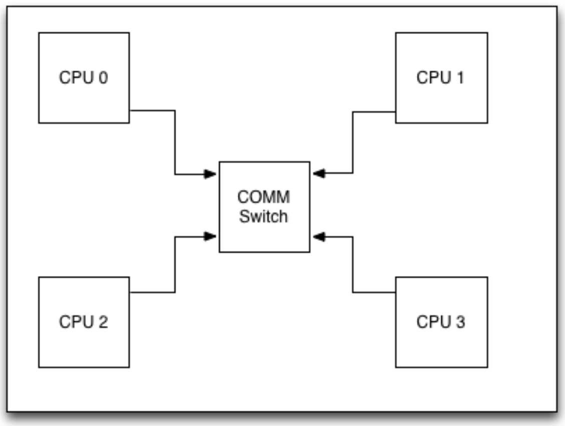

Figure 3.2: XCore Architecture

3.1.2 Concurrency

The XS1 has eight hardware threads on each core. Threads are scheduled with a round-robin hardware scheduler; threads in a block state are skipped until they are active again.

One of the design objectives of the the XS1 is deterministic timing. This implies not having a cache. Performance is deterministic, according to these bounds given by the XC handbook[?]:

$$timeslice = \begin{cases} |t| < 5 & : 1/4 * |cycles| \\ |t| >= 5 & : 1/|t| * |cycles| \end{cases}$$

Note that after 4 threads are allocated upon a core, performance degrades. The above equation is graphed on Figure ??.

The XC threading model specifies that only a single thread is able to write a given variable. Other threads can read that variable, but are unable to write. Multiple writers are prohibited by the compiler. Therefore, the architecture of XC ensures that memory write conflicts are impossible by design, even in a multi-threaded configuration. This definition is extended to the I/O port definitions and use. Only one thread can write a given port; all other threads on that core are limited to reading the port.

This semantic enforcing of only a single thread having write access in XC handles

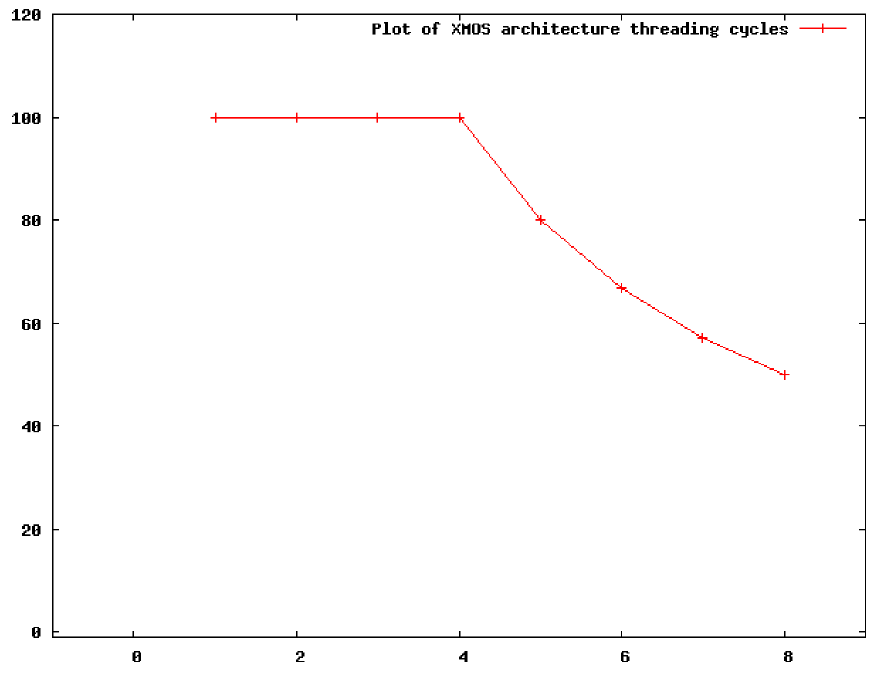

Figure 3.3: Cycles allocated per thread

the race conditions that can arise when multiple threads of execution occur and attempt to write a memory location.

One of the key ideas of the XS1 architecture is event driven programming instead of loop-driven programming. Part of the Instruction Set Architecture (ISA) includes programmable events that allow a “thread wait until event” approach to programming. XC supports this idea with a SELECT statement, which selects the next available event in a given set of possible incoming events. In addition to this design, classic interrupts are supported as well as a hardware based exception model.

3.2 Programming the XCore

The XCore used is the XC1-A development board, produced by the XMOS company. It is programmed with a Universal Serial Bus (USB) connector, using software development tools provided by XMOS. The tools are based on the GNU tool chain and run on Windows, Macintosh, and Linux systems that support a USB connection. Also supplied is an IDE based off the Eclipse IDE framework. The XCore IDE includes a gdb-based debugger and a simulator.

XMOS developed and released a parallel language XC, a modified C-family programming language using occam constructs. Key XC architectural decisions that share the CSP lineage are shared-nothing memory and synchronized channels. In the discussion following, the difference between XC and C is reviewed.

Keywords

XC adds a number of new keywords to C to enable parallel programming. They are summarized below.

chan defines a channel; a queue through which information can flow.

chanend defines the end of a channel; used when passing a channel to a function.

core defines a type of 'core' which takes ports and parallel statements.

master defines the sender in a transaction-style communication.

on defines which core to execute the parallel computation upon.

par defines a block of parallel computations at the top-most level of the XC program.

select defines a polling mechanism much like Unix `select(2)`. Select takes a set of condition blocks such as one containing `when` and takes the first available condition that evaluates to true.

slave defines the receiver of data in a transaction model of data transfer.

streaming defines a channel that does not contain overhead of control tokens; e.g., a channel receiving a long list of data coming.

transaction defines a communication such that the synchronization between each transfer is turned off.

when defines a blocking wait until a given condition becomes true.

Static Rules

Pointers have been removed; a reference type has been added, with the restriction that a reference can be valid or nullable. A reference has a 1-1 mapping between a reference and an object. Arrays are implicitly passed by reference.

3.3 Hardware/Software Environment

The environment used to create the debugging tool is described below, along with sufficient information to recreate the system.

3.3.1 Embedded Environment

Hardware Environment

The development kit is a pre-assembled board with 12 LEDs, 4 buttons, a speaker, a power supply, the XC-1A CPU, and a USB communications cable. The CPU has 65535 bytes of RAM per core, and executes at 400 MHz.

In order to load programs on the embedded system, the `xrun` program must be executed on the development computer. `xrun` can function as both a JTAG connector and the loader. `xrun` installs the program in memory and jumps to the binary executable point.

Software Environment

The software environment on the embedded system the programmer is presented with is the XC software abstraction, along with a set of libraries; `stdlib.h` and several XCore-specific libraries.

The programmer can access the processor switch via the standard GNU C compiler assembly function `asm`, combined with the `GETPS` and `SETPS` instructions.

Simulator

XMOS also offers a simulator, `xsim`. The simulator can utilize several models of the XMOS hardware in varying fidelity, as well as simulating inputs and outputs. For the research in this report, the default (slower) option of high fidelity was used.

The simulator offers the ability to trace the execution of the hardware, with a detailed record of the state of the processor core registers.

The simulator does not record the state of the internal processor switch in its output.

3.3.2 Desktop Software

The desktop software is composed of three different systems: the XMOS software, the software tools used to build this report's implementation, and the software implemented for this report.

The XMOS software programs this report uses are `xcc` (a XC/C compiler derived from `gcc`), `xobjdump` (a disassembler), and `xsim`, a simulator.

The software chosen for a development language to build the debugger in is Common Lisp for its standardization and its cross-platform open-source implementations.

The Common Lisp implementation chosen was Steel Bank Common Lisp (SBCL) [?], and the report software was developed on SBCL versions 1.0.51-55. In addition to the compiler itself, several libraries were used in producing the report software, enumerated below.

1. CL-PPCRE is "Portable Perl-compatible regular expressions for Common Lisp"; it serves to allow us to parse simulator outputs[?].
2. ALEXANDRIA is an assortment of common utility functions not included in the Common Lisp standard [?].
3. CL-DOT is a package for converting a Common Lisp graph structure to a PNG or other format supported by the DOT utility in the Graphviz suite[?, ?].
4. BATTERIES is a package that has utility functions, written by the author of this report[?].

CL-PPCRE, ALEXANDRIA, and CL-DOT were installed using the Quicklisp[?] library manager, which keeps them up to date with the latest release.

Software Environment

The development system for the report project was primarily a Macintosh OSX 10.6 ("Snow Leopard") laptop with 4 GB of RAM and a four-core Intel processor.

Due to the portability of the Common Lisp systems, the report project was also able to be executed on an Ubuntu Linux 10.10 installation, using SBCL 1.0.52.

3.4 Theory of the Report Project

Generally, the goal in debugging is to understand part of the state of the software under analysis. Then, through the process of cyclical debugging, the software can be modified to perform correctly.

As described in chapter 2, the key challenges in debugging relate to observability, repeatability, and control. This report focuses on the challenge of observing the system, leaving aside repeatability and control as not in scope.

Slightly restated, the hypothesis from chapter 2 is that *a targeted trace can provide an effective view into the state of the software.* Recall that "effective debugging" is defined as the capacity to determine some information regarding the internal state of a system under investigation. The consequence of a targeted trace is that what really happened is recorded, for a given segment of the software under consideration.

In order to this test this hypothesis, this report observes the communication points and determines the set of possible interleaved executions of a given trace. The mechanism for determining interleaving uses the causality relationships between messages. The algorithm developed herein is called the Targeted Trace Algorithm, or *TTA*.

Algorithm Synopsis The algorithm, at a high level, is as follows: it collects the record of the execution from each thread of execution, filters out non-communicating records, and forms links of *possible* paths of execution based upon the ordering of communication. Links are formed starting from the end point, working back towards the start point. Following the linking, an all-paths routine is executed over the graph from the starting point to the ending point; this is called the *possible* graph - it denotes the collected sets of possibilities, including possibilities that will be found impossible. Then, a set of rules are applied to generate the set of actually possible execution orderings, the *feasible* paths. These rules are based on causality requirements; TTA leverages the constraints of synchronous communication in order to form causality requirements. Each feasible path represents an execution which could really have occurred. This is a two-phase algorithm: phase one connects the possible linkings together into a possible graph, and phase two extracts the set of execution orderings that are feasible. The implementation of the TTA algorithm presented in this report renders both the possible graph and the set of feasible path graphs as PNG images.

3.4.1 Causality And Linearization

A targeted record is defined to be a subset of the execution sequence, beginning at some start point of execution and finishing at some end point of execution. This targeted record is called a *trace*. Therefore, in order to generate a trace of the system, the ability to (i) insert a START and STOP mechanism into the execution and (ii) the ability to trace the system between START and STOP must be present.

START/STOP The START/STOP mechanisms represent the beginning and ending of the tracing. In an arbitrary program, there is no mechanism to ensure that a program flow of control will wind its way from one point to another. Therefore, the developer must inject the constraints themself. Informally stated, the following restrictions must apply on the START/STOP pair:

1. The STOP must be reachable by all paths from the START.
2. When multiple cores execute different regions of code, the STARTs and STOPs must logically correspond between cores; that is, all STARTs/STOPs must represent the beginning/ending of tracing in the flow of the parallel program.

The reasoning is as follows: suppose the STOP was not reachable by all paths from the START. Then, in the paths where it was not reachable, the information gathering would not cease and would require too much memory to hold the information for the region under examination.

Again, suppose the START/STOPS were distributed over different blocks of code. In this case, if they correspond to temporally different sections of the software under study, then the analysis of communication will be flawed, since the analysis will be run over different points in time with different communicating INs and OUTs.

While this study's implementation of TTA frames START/STOP as a pair, it is reasonable that this mechanism could be generalized to have multiple STARTs and STOPs, so long as the restrictions above apply. In the abstract analyses in this report, START/STOP will be referred to as top ($\top$), and bottom ($\perp$), respectively.

Rules of Feasible Traces In order to form what this report terms a *candidate trace*, that is, a trace under consideration to be a trace that *really happened*, certain rules must be obeyed. Some of these rules are intuitively obvious, others stem from the nature of the XCore communication.

In the XCORE architecture, for a given core C_i, an `OUT` message to another core C_j over a channel H_i will be paired with a `IN` message. It is assumed that in a core C_i, all instructions on it are executed sequentially. With an `OUT,IN` pair, the `OUT` always initiates before the `IN`; until the `IN` finishes and signals that it has received communication, the `OUT` does not proceed and the thread is blocked. This is a synchronous communication with the `OUT`s blocking for reception by `INs`. Conceptually, this is abstracted to say that an `OUT` is always followed by an `IN`. Further, `IN`s will always be followed by an `OUT` in regular communication (not the beginning or ending of a communication sequence).

To demonstrate this, suppose a candidate trace looked like this: OUT_0, IN_0, IN_1. The first OUT_0 has an IN_0. Then, the IN_1 must only begin its processing after data has come to it; but there is no OUT for IN_1. Thus, an `OUT` must take place after an `IN` *unless* the `IN` is the terminating `IN`.

The following rules can be constructed:

1. `IN`s are always preceded by an `OUT`.
2. An `IN` blocks communication on both execution threads until its data is received.

Conceptual Example In this example a trace for a two-core device is reconstructed. Rules are applied *on the fly*, to simplify the example.

In a execution trace such as the following:

$$C_0 = [OUT_0, OUT_1, OUT_2]$$
$$C_1 = [IN_0, IN_1, IN_2]$$

We know by (1) that each IN_i corresponds to each OUT_i, since each core executes in order on its own line of action. We know by (2) that the subscripts on C_1 correspond to the correct ordering of execution on C_1. Therefore, we can reconstruct the ordering of the trace. The idea is sketched below in a step-through. On the right is given the set of possible states with the most recent event on the right.

1. OUT_2 precedes IN_2 by (2) $\rightarrow \{[OUT_2, IN_2]\}$
2. OUT_1 precedes OUT_2 by (1) $\rightarrow \{[OUT_1, OUT_2, IN_2]\}$
3. OUT_1 precedes IN_1 by (2) $\rightarrow$
   $\{[OUT_1, IN_1, OUT_2, IN_2], [OUT_1, OUT_2, IN_1, IN_2]\}$
4. OUT_1 must be followed by $IN_1 \rightarrow \{[OUT_1, IN_1, OUT_2, IN_2]\}$
5. OUT_0 precedes OUT_1 by (1) $\rightarrow \{[OUT_1, IN_1, OUT_2, IN_2]\}$
6. IN_0 precedes OUT_1 by (1) $\rightarrow$
   $\{[OUT_0, IN_0, OUT_1, IN_1, OUT_2, IN_2], [IN_0, OUT_0, OUT_1, IN_1, OUT_2, IN_2]\}$
7. OUT_0 precedes IN_0 by (1) $\rightarrow \{[OUT_0, IN_0, OUT_1, IN_1, OUT_2, IN_2]\}$

It is obvious that for a set of nodes to represent the communication that took place, they must be the *entire* set of communication nodes. This allows this report to observe a third rule for a feasible trace:

3. All nodes must be represented in a trace.

Consider the case when there are four threads of execution. Threads 0 and 1 are communicating and threads 2 and 3 are communicating. There are no other communication links.

In this configuration, whether the entirety of communication between threads 0 and 1 occurs before, after, or interleaved with the communication beetween threads 2 and 3 is not able to be determined by the causality rules laid down above. Because of this, the correct result of the causality analysis will be a set of feasible communications; each element in the set represents a *possible* communication record that is unable to be ruled out. However, because these results do not depend on each other (by communication), all are feasible for the purpose of debugging.

Building the communication graph One of the building blocks for the TTA algorithm is to link `OUT`s with possible `IN`s for graph analysis. As an informal walk-through for what is required, consider the following constructed case; each instruction is subscripted by its communication with its counterpart:

$$\begin{aligned} C_0 &= [\top, OUT_0, OUT_1, OUT_2, IN_4 \bot] \\ C_1 &= [\top, IN_0, IN_1, IN_2, IN_3, OUT_4 \bot] \\ C_2 &= [\top, OUT_3, \bot] \end{aligned}$$

A visual depiction of the example is given in Figure **??**, with manually drawn dotted lines denoting the actual path of execution. The task of the TTA algorithm will be to automatically realize the dotted lines.

Suppose that we begin to construct a set of candidate communication linearizations by walking backwards from $\bot$.

Note that C_0 ends with an `IN`, the only core to do so. Therefore, IN_4 on C_0 was the last element in the sequence. However, both C_1 and C_2 have a trailing `OUT`, and C_0 has an `OUT` immediately prior to the final `IN`. Unless the software environment adds metadata serving as a communication index (e.g., a Lamport clock), it is impossible to disambiguate at this point between C_0, C_1 and C_2 - any could have sent the message - and therefore, the trace graph must now include all of C_i as possibilities prior to pruning infeasible possibilities. This reasoning must be applied at each instruction - what instructions could have led to it? - and appropriate links formed. To continue the example, consider the following walk through of a link-up process, giving reasoning of each step.

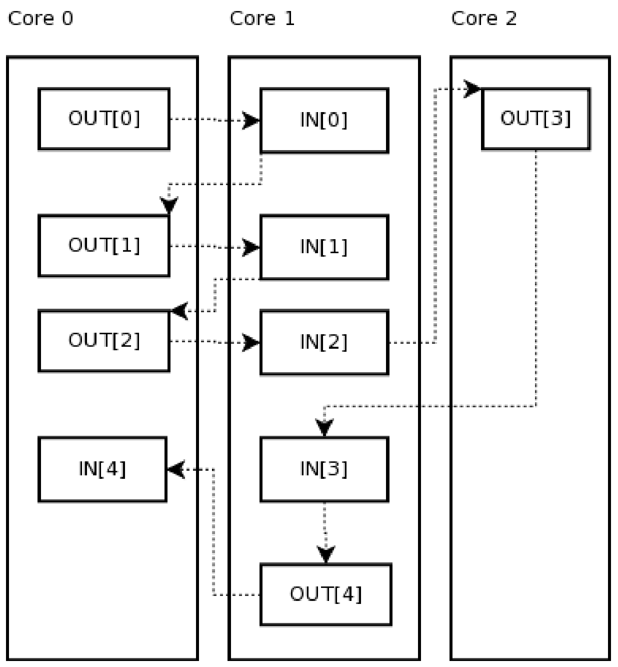

Figure 3.4: Nodes in Link Example

1. $[[IN_4, OUT_2], [IN_4, OUT_4], [IN_4, OUT_3]]$, as already described.

2. $[[IN_4, OUT_2, IN_3], [IN_4, OUT_4, IN_3], [IN_4, OUT_3, IN_3]]$. Since `OUT` must be followed by `IN`, and the next available `IN` is IN_3, use IN_3. While IN_2 or earlier *might* be selected, that would not respect the order of execution on C_1.

3. $[[IN_4, OUT_2, IN_3, OUT_3], [IN_4, OUT_4, IN_3, OUT_2],$
$[IN_4, OUT_4, IN_3, OUT_3], [IN_4, OUT_3, IN_3,]$. IN_3 must be preceded by an `OUT`; the only available `OUT`s are OUT_2 and OUT_3.

The above demonstrates a sample linking operation.
Consider the following constructed case:

$$C_0 = [\top, OUT_0, OUT_1, OUT_2 \bot]$$
$$C_1 = [\top, IN_0, IN_1, IN_2 \bot]$$

In the initial naive link-forming setup of the possibility graph, IN_2 will read OUT_0, OUT_1 and OUT_2 as being potential precedence candidates, as well as having IN_1 as a definite preceding instruction (since it occured prior in the execution thread). OUT_1

will also have IN_0, IN_1 and IN_2 as a potential for sending out. This will cause a cycle to form in the graph. After these links have been formed into potential paths with an all-paths algorithm, feasibility tests (tests 1, 2, and 3 above) are run against each path from $\top$ to $\perp$ to determine if a given path is legitimate. In the experimental work in chapter 4, all cycles have been eliminated. This is due to all-paths finding all *linear* paths from the START node to the END node.

In order to generate feasible sets of communications, this report uses the following algorithm:

Algorithm 1 General trace algorithm

```
  Given a set of traces $T_i$ with INs and OUTs for a number of cores $n$. Each core $C_i$
  is an independantly numbered trace.
  Consider the $T_i$ as a set of disconnected links in a graph.
  Add a $\top$ node for the START, and connect it to the first element for all $C_i$; add a
  $\perp$ node for the STOP, and connect it to the the final element for all $C_i$.
  Foreach IN; For the given IN, find all possible OUTs for that IN.
  Determine all paths between $\top$ and $\perp$.
  Remove all paths that are not feasible.
```

As a note, in a complete graph G, there are $n!$ paths possible between $\top$ and $\perp$. As this is computationally impractical on larger n, the algorithm will have to be optimized. One possible path to improvements would be rolling the infeasibility tests into the all-paths algorithm and pruning paths that become infeasible, thus not storing or forming further links on infeasible paths.

XCore Hardware Trace

In order to trace the instruction stream, it must be possible to (1) turn tracing on and off, (2), record all pertinent information, and (3) transmit it to the host system. Obtaining an execution trace with software will be limited by the test-bed environment, whatever it will be.

The XCore chip does not have a single-step flag that can be set in the CPU status word. Therefore, as a mechanism of tracing, the debug interrupt was selected as a way to perform a trace. The high-level algorithm is given in Algorithm **??**.

The XCore debug interrupt performs the routine in Algorithm **??** (pseudo-coded) when the interrupt is executed. PC denotes the Program Counter; SP denotes the Stack Pointer.

After the SP has been pushed, the user inserts whatever code he so desires.

Algorithm 2 General hardware trace algorithm

```
Define a DBG-trace routine, which records the information
Define a START-dbg routine, which sets up the tracing environment
Define a STOP-dbg routine, which tears down the tracing environment
Define a DISPATCH-dbg routine, which communicates the trace
Set debug interrupt vector to our DBG-trace routine.
Execute START-dbg on all cores of interest
Perform computations under inspection
Execute STOP-dbg on all cores of interest
Execute DISPATCH-dbg
```

Algorithm 3 Beginning debug interrupt

```
oldPC $\leftarrow$ PC
oldSP $\leftarrow$ SP
PC $\leftarrow$ debug-interrupt-address
SP $\leftarrow$ newSP
```

Then, on execution of the return-from-debug-interrupt instruction `DRET`, the routine (pseudo-coded) in Algorithm **??** is executed

Algorithm 4 Returning from debug interrupt

```
PC $\leftarrow$ oldPC
SP $\leftarrow$ oldSP
```

The key difficulty is the fact that on the debug interrupt being taken, the correct next instruction address is not saved, but it remains set to the PC when the interrupt was fired. Since the next instruction address is variable due to (1) the branch instructions and (2) the variable length instruction set, computing the next address to be executed is not wholly straightforward.

Therefore, in order to correctly increment the PC, the following algorithm in Algorithm **??** must be implemented.

Implementing this algorithm raises two issues; first, the algorithm to parse the instruction takes significant amounts of instruction memory space, approaching the total size loadable into the memory space, and second, it does not provide insights into the hypothesis under question, rather, it provides insights into the *specific* processor, rather than a more general situation. Since it does not contribute to verifying the hypothesis, the approach is set aside.

By the process of elimination, this leaves the simulator for the experiments.

Algorithm 5 Correct PC Incrementing

```
if PC is a jump instruction then:
    Determine the jump location.
    PC $\leftarrow$ to the jump location
else:
    Determine the length of the PC's instruction
    PC $\leftarrow$ current PC + length
end if
```

3.4.2 Solution Design - Simulator

Since the simulator executes the trace and records the result, the simulator trivially solves the problem of recording the trace and communicating the trace to the analysis system. The next question to be be solved is how to present the information.

An abstract flowchart is used in the rendering from the TTA to present the information regarding causality. A sketch of the reasoning is presented here. Multiple mechanisms for displaying program behavior exist. They can largely be categorized into two different groups: exact representations of the behavior or abstracted representations of the behavior. The exact representations are precise, but they have a significant potential problem in the amount of data the user is not interested in. The converse is true with an abstracted representation. They are imprecise, but the data is selected to present only the useful information to the user, frequently losing detail that is useful. Potentially, depending on the system and the available information, higher-level information concepts can be drawn out of the trace and presented, leading to more useful relationships being visible to the user.

This study takes the abstracted approach, and is able to formulate from the trace the interleaving of the messages. The presentation of the information is given as a graphic display of the messages exchanged between the executing cores.

3.4.3 Targeted Trace Reporting Algorithm

This describes how to obtain the needed information from the embedded system.

1. Install a monitoring system on the embedded system. This can be mimicked with a simulator

2. Insert an operation for START and STOP in the source code for the system. From the point where START is tripped to the point where STOP is encounted,

each communication IN or OUT is monitored and recorded; this information forms the trace. A simulator can be used to store this information.

3. After the STOP is encountered, the trace information is stored outside of the embedded system.

3.4.4 Targeted Trace Analysis Algorithm

Below, Lisp pseudocode is listed which provides a concrete view of the key idea given in section **??**.

Listing 3.2: Targeted Trade Analysis Algorithm

```

(defun all-paths (start-node end-node)
  "Returns all paths in a graph beginning at ‘start-node‘
   and ending at ‘end-node‘."
   ;; standard algorithm, not pseudocoded.
   )

(defun read-graph-from-trace (trace objdump)
  "Returns a graph from a program execution, combined with a
   disassembly reading of the program. This reads from the
    Targeted Trace Reporting Algorithm output"
   ;; not pseudocoded, see project source for implementation
   ;; details
   )

(defun find-all-paths-for-trace  (trace objdump)
  "Finds all paths in the trace"
  (multiple-value-bind (start end)
      (read-graph-from-trace trace objdump)
    (all-paths start end)))

(defun prune-too-short-paths (paths trace objdump)
  "Finds the number of nodes in the execution trace, then returns
   a list of all paths that have only that number of nodes"
  (let ((actual-length (determine-number-of-nodes (trace objdump))))
    (remove-if #'(lambda (path)
                     (/= (length path) actual-length))
               paths)))

(defun prune-bad-starts (paths)
  "Removes all paths that attempt to start with an IN; those violate
   a causality rule"
  (remove-if-not #'(lambda (path)
                         (is-outgoing (first path)))
                 paths))

(defun pair-up-nodes (list)
  "Pairs up all list[i] with the corresponding list[i+1] and
   returns a list of pairs.
   e.g., (1 2 3 4) => ((1 2) (2 3) (3 4)"
    ;; helper code, unimplemented in pseudocode
  )

(defun prune-incorrect-sequencing (paths)
  "Removes all paths that do not have ins followed by outs
   followed by ins"
   (remove-if-not #'(lambda (path)
                      (let ((pair-up-nodes path))
                         (loop foreach pair in path
                          set-and
                           (or (and (is-incoming (first pair))
                                    (is-outgoing (second pair)))
                               (and (is-outgoing (first pair))
                                    (is-incoming (second pair)))

                  paths)))

(defun find-feasible-paths-for-trace (trace objdump)
  "Returns paths that really could have existed"
  (prune-incorrect-sequencing
    (prune-bad-starts
      (prune-too-short-paths
        (find-all-paths-for-trace trace objdump)
        trace
        objdump))))
```

The heart of the algorithm is to determine which set of paths actually could have happened is `find-feasible-paths-for-trace`.

The implementation for this report uses these paths to generate a graphical layout of how the program executed. It is theoretically possible to have multiple paths of feasible execution; this means that multiple images will be generated.

In the implementation this report also emits a rendering of the nodes from `read-graph-from-trace`: this presents a view of the graph before linearization and pruning out of infeasible paths.

3.5 Implementing the system

Two fundamental areas have to be implemented for an embedded debugging system: the mechanism on the embedded system itself, and the development computer. When a simulator is used, the simulator is considered the embedded system.

3.5.1 Overview

In the research implementation of the targeted trace algorithm, START and STOP are implemented as labels in the assembly code. These labels are emitted with the given instruction memory address when the object code is dumped out via the tool `xobjdump`. The object code is analyzed to determine what part of the slice to present to the analysis and rendering system.

If the algorithm is implemented on hardware instead of a simulator, another approach is to have START begin single-stepping and STOP to pause single-stepping, then communicate the gathered information for the duration of the single-stepping to the host. This is presented in Algorithm **??**.

The primary task for the report implementation is to read the trace information and derive meaningful content for the user to be able to make correct inferences regarding the state of the executing software. The analysis task is to overlay the previously mentioned theory to the existing system.

Figure **??** gives a flowchart of how the system passes data around. XC code is compiled by `xcc`; the executable is decompiled with `xobjdump` and the results fed into the TTA system. The executable is also executed in the simulator (`xsim`) and the simulation results fed into the TTA system, which renders feasible graphs and teh possibler graph to png files.

3.5.2 Internal Software Design

The report software is structured in a layered software design approach. Each layer communicates with the ones above and below it and only those. The top level function manages both the input and the output.

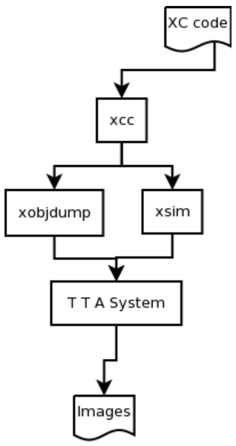

Figure 3.5: Flow of Data in the System

Figure **??** provides a call-tree graph of the software design, giving an approximate view of the software organization.

Input interface The input interface is used to read the key information in from the trace and the disassembly of the object code.

At the IO interface in the `render-trace` function, the trace file and the `xobjdump` file are read in with the `load-sim-file` and `load-xobjdump` blocks. Following the initial read of the simulator file, the trace file is parsed into semantically meaningful objects, describing the events of the trace. The xobjdump file is read in and the START/STOP labels are parsed out.

The `xobjdump` and simulator information is sent to a first pass of filtering in `filter-untraced-location`. Here the `xobjdump` information giving boundaries for the trace to occur in is used. Instructions occurring outside of these boundaries are disregarded.

The result is a set of lists of instructions; a concurrent trace graph. The input interface will, in general, be the place where new architectures would be fed into the report implementation.

Analysis In the analysis phase, a semantic analysis is performed and the key causality algorithms are applied against the trace graph.

Semantic rules are applied to the trace: instructions that are not `IN` or `OUT` are stripped out of the trace, leaving only communication instructions. These are sorted into the cores they occurred on and which channel the communications took place. Each communication channel is paired up and then consolidated in

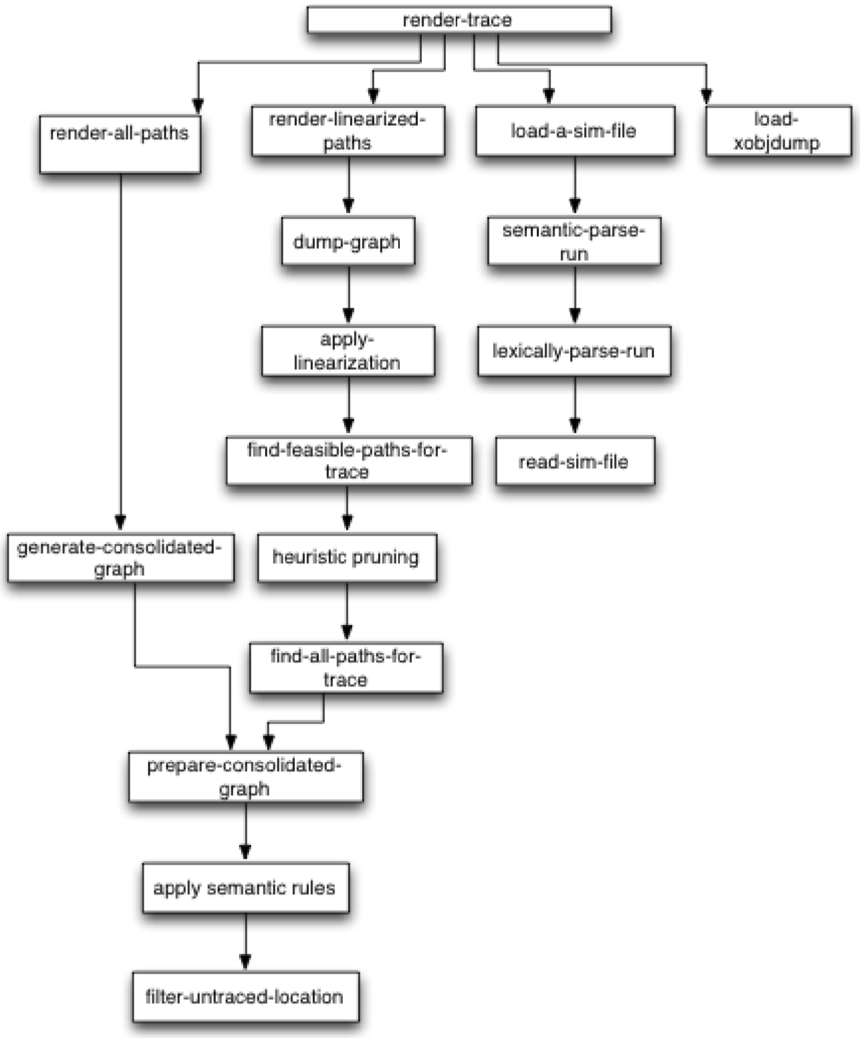

Figure 3.6: Software Call Flow

`prepare-consolidated-graph`.

At this point, a graph has been fully set up with nodes being `IN` or `OUT` communication points, and arcs being the combination of all possible control paths *and* data paths. The combined representation is used as a tool to indicate communication, since communication in this system induces causality (control flow), and also data, since communication is done with data.

Referring to section **??**, this graph, since it represents *possibilities*, has the potential to contain cycles. While the actual trace of execution is, of course, acyclic, and the set of feasible outputs must be acyclic, intermediate possibilities have proven in our experiments to be cyclic in possibility. Each possible path is connected to a

START and STOP node in the graph, respectively denoting the the start of the trace and the end of the trace. All computations must begin at the START of the trace; all traced computations are defined to be ended at what the STOP node.

After all possible paths are generated, all possible directed graphs are formed that begin at the START and end at the STOP node.

At this point, the software produces a graph of the possibilities, without any heuristic or rules pruning for feasibility. This is intended to demonstrate the entire possibility space directly. The `render-all-paths` branch of the `render-trace` call tree shows this graph.

On the `render-linearized-paths` branch, the set of possible (a.k.a feasible) actual execution paths are sought. Feasibility in the context of the execution traces is informally defined as "Could actually have occurred" This set of node paths is pruned for feasibility in `find-feasible-paths-for-trace`, according to the rules presented earlier in this chapter:

1. INs are followed by OUTs
2. OUTS are followed by INs.
3. The number of nodes in the computation must be represented in the graph.

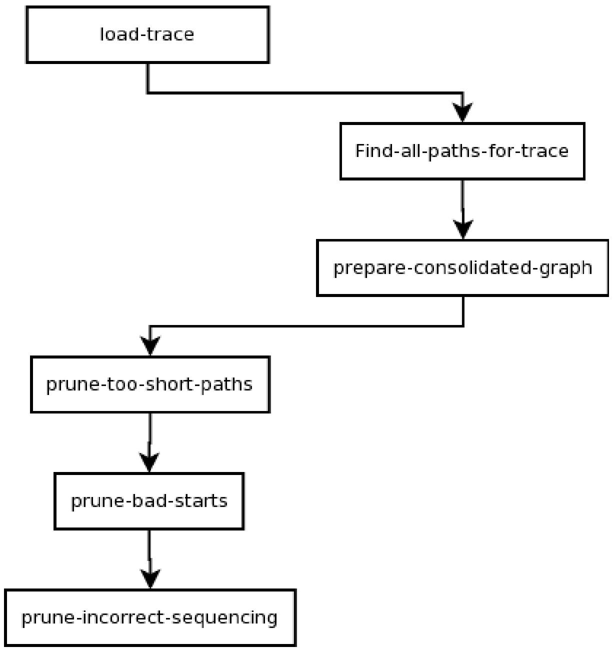

Figure 3.7: Analysis Algorithm Flow

Output interface After the analysis is finalized, the data structures are manipulated to produce the set of feasible graphs in a form that is renderable to the graphical format for `dump-graph` and the CL-DOT package. CL-DOT is used by passing in an object graph, which is then annotated by the report code and a PNG file of the graph is built. Annotations provide information regarding each communication.

3.6 Interface

Suppose the reader has an XC file `example.xc` is under examination.

Then, in order to use the existing implementation, the user must insert a START/STOP macro (see Chapter 4 for examples, as well as the source code for this report's project). Then, the following sequence of commands should be issued (see Fig. **??** for a graphical depiction):

```
xcc example.xc XC-1A.xn -o example.xc

xobjdump example.xe -S -D > example.objdump

xsim example.xe --trace-to example.trace
```

Currently, the expectation is that all the files will have the same prefix but different suffixes.

The software is implemented in Common Lisp; in order to execute the software, the file "sim-reader.lisp" is loaded into a Lisp system; when the file is loaded, the command:

```
(render-trace example example.png)
```

will emit the png files `example-all.png` and `example-`n`.png` (the `-`n pngs are possible feasible paths) into the current working directory.

These are the visualizations for the purposes of debugging; they will be reviewed in Chapter 4.

Chapter 4

Experiments and Results

These experiments will attempt to demonstrate the viability of post-execution analysis in a multicore embedded environment. Since one of the key issues in distributed computations is the ordering of events, communication events have been selected for analysis, specifically, IN and OUT events.

In chapter one, three key challenges were identified: observability, repeatability, and control. In the experiments, observability is attained (i) by the START/STOP probes as described in the prior chapter, and (ii) by the use of the `xsim` simulator for the XCore. Repeatability and control are essentially out of the scope for this report and are not treated here; the goal in this work is to improve observability.

Each experiment is reviewed and related to the background and the environment, noting where limitations were found and what the software system did or would have had to do to overcome them.

4.1 Experiments

The general form of the experiments is as follows. XC source core is compiled and simulated. The output is gathered into the research software and the analysis described in Algorithm **??** is performed. The software emits at last twooutputs in PNG files; the graphics is in a flowchart format. One output is the "all paths" output, which contains both the data and control flow dependencies. The other output is the analyzed paths which are the set of possible event orderings found that could possibly have happened; one picture per ordering is emitted.

4.1.1 Tracing a single communication

The most elementary message possible is a single message that is received on another core.

It was implemented in the following program.

Listing 4.1: One Transfer

```
/*
  Paul Nathan 2011:  one-transfer.xc
  Produced as part of Master's Thesis work at the University of Idaho.

  This code is intended to transfer 1 word from 1 core to the next
  one, then stop.
 */
#include <platform.h>
#include <stdlib.h>
#include <stdio.h>

#define STARTTRACE(core) asm("starttrace" core ":");
#define ENDTRACE(core) asm("endtrace" core ":");

int main(void) {
    chan queue1;
    par {
        on stdcore[0]: {
                int outpacket = 0xf0;

                STARTTRACE("0");
                queue1 <: outpacket;
                ENDTRACE("0");
            }
        on stdcore[1]: {
                int inpacket;

                STARTTRACE("1");
                queue1 :> inpacket;
                ENDTRACE("1");
            }
    }
    return 0;
}
```

Observe the use of the STARTTRACE and ENDTRACE macros, which imple-

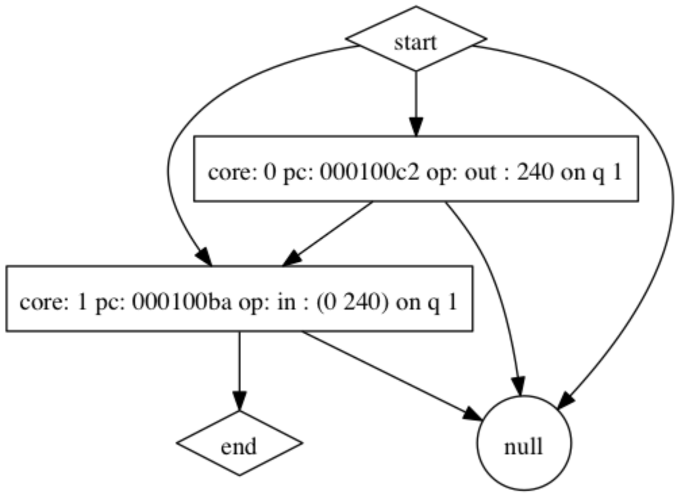
start
core: 0 pc: 000100c2 op: out : 240 on q 1
core: 1 pc: 000100ba op: in : (0 240) on q 1
end
null

Figure 4.1: All possible paths for the transfer

ment the START/END probes from Chapter 3. They are used to provide information via the `xobjdump` command to the system to determine where to begin tracing.

In this experiment and the ones that follow, the program under investigation is compiled, simulated, `xobjdump`'d and the results are analyzed. The primary output of the analysis are graphical visualizations of the communications.

All possible paths in the directed graph from the START to the END are calculated, both in the control and data path spaces. This is graphed in Figure **??**.

Figure **??** displays several critical pieces of information. A virtual start node is created, forming a 'start state' for all cores. The execution graph begins at start. Similarly, the stop node serves the same purpose. The Null node denotes a non-use of a core; in this graph, the other two cores are denoted to be in null.

The rectangles denote operations. Each operation is formatted similarly:

```
core <num> pc <PC address> op:  <operation> <data> on q <channel num>
```

Notice the data is different between IN and OUT. OUT's data denotes the data that is being sent out; IN's data denotes the data received, and the register number that received it.

After determining all paths between the START and the STOP and pruning according to the rules laid out in chapter 3, Figure **??** represents the inferred path of the communication.

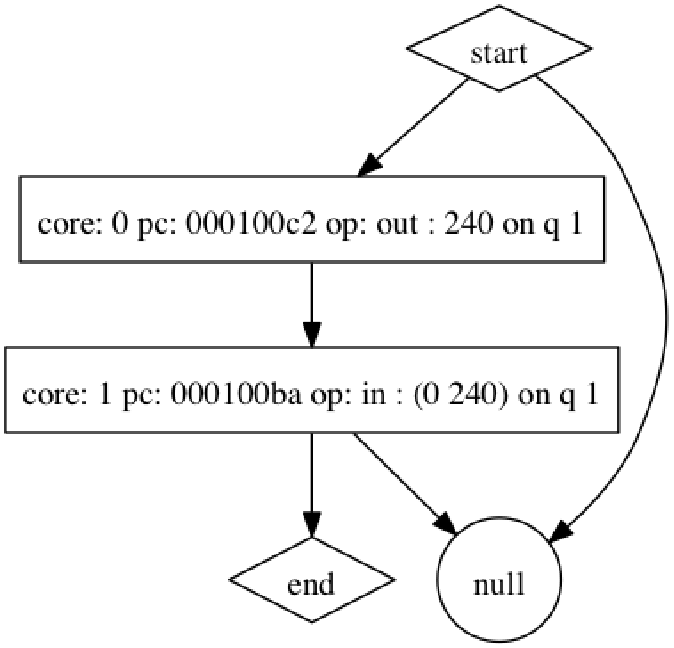
start
core: 0 pc: 000100c2 op: out : 240 on q 1
core: 1 pc: 000100ba op: in : (0 240) on q 1
end
null

Figure 4.2: The inferred path of the transfer

4.1.2 An exchange

The next step of difficulty is to send and receive from one core to another. This demonstrates the ability to observe both the sending and receiving on both cores, a critical part of a correct observation. This is implemented in Listing ??.

Listing 4.2: One Exchange

```
/*
  Paul Nathan 2011: one-exchange.xc
  Produced as part of Master's Thesis work at the University of Idaho.

  This code is intended to transfer 1 word from 1 core to the next
  one, then back again.
 */
#include <platform.h>
#include <stdlib.h>
#include <stdio.h>

#define STARTTRACE(core) asm("starttrace" core ":");
#define ENDTRACE(core) asm("endtrace" core ":");

int main(void) {
    chan queue1;
    par {
        on stdcore[0]: {
                int outpacket = 0xf0;

                STARTTRACE("0");
                queue1 <: outpacket;
                queue1 :> outpacket;
                ENDTRACE("0");
            }
        on stdcore[1]: {
                int inpacket;

                STARTTRACE("1");
                queue1 :> inpacket;
                queue1 <: inpacket + 1;
                ENDTRACE("1");
            }
    }
    return 0;
}
```

Analysis of Listing **??** proceeds to the generation of Figure **??** to visualize the all-paths flow.

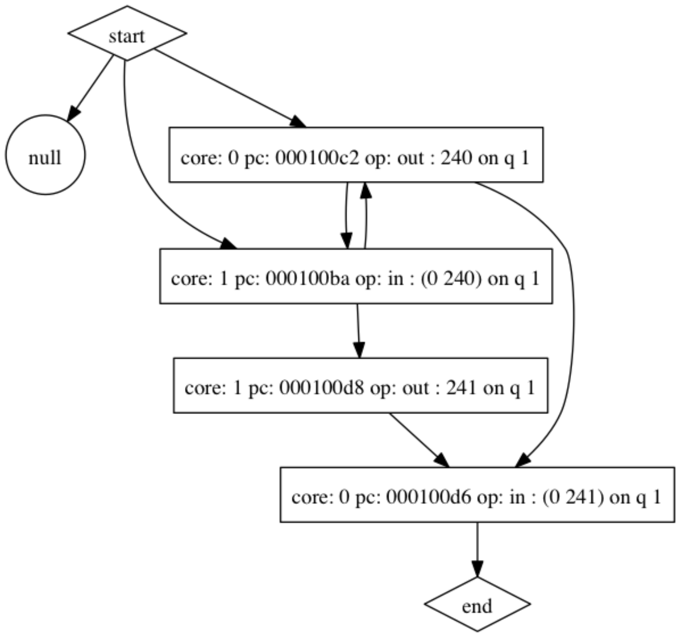
start
null
core: 0 pc: 000100c2 op: out : 240 on q 1
core: 1 pc: 000100ba op: in : (0 240) on q 1
core: 1 pc: 000100d8 op: out : 241 on q 1
core: 0 pc: 000100d6 op: in : (0 241) on q 1
end

Figure 4.3: All paths of an exchange

Applying the rules leads to Figure ??. Notice that OUTs and INs have different transition capabilities, conforming to the causality rules from chapter 3.

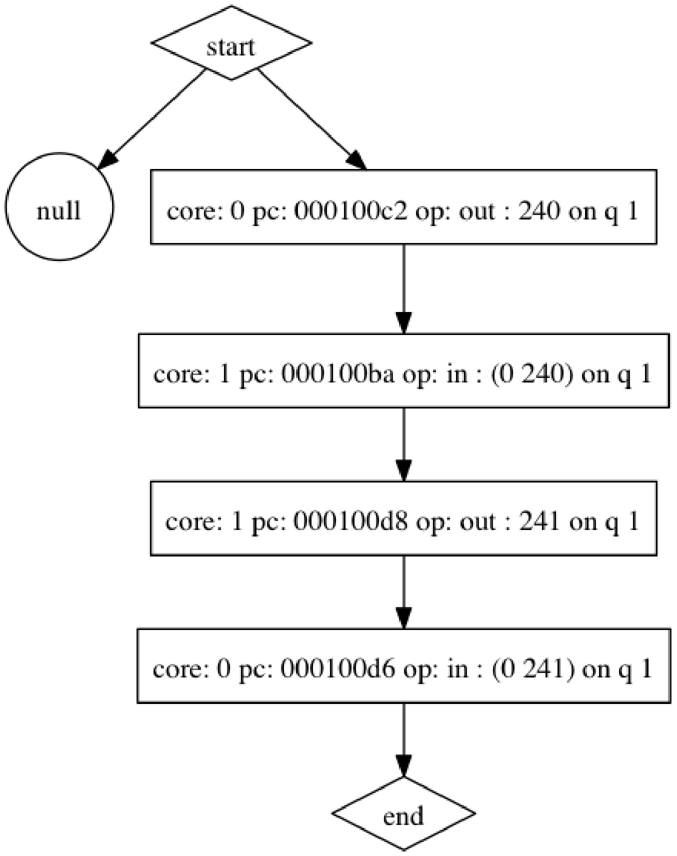
start
null
core: 0 pc: 000100c2 op: out : 240 on q 1
core: 1 pc: 000100ba op: in : (0 240) on q 1
core: 1 pc: 000100d8 op: out : 241 on q 1
core: 0 pc: 000100d6 op: in : (0 241) on q 1
end

Figure 4.4: Path of an exchange

This begins to demonstrate the software's capacity to visualize the communication between cores.

4.1.3 Three transfers

Moving forward in complexity, a classic producer-consumer system is selected; Listing ?? models a producer-consumer system that dispatches three messages from the producer to the consumer. Notice the values selected to be dispatched: `0x0f`, `0x10`, `0x11`.

Listing 4.3: Mini Producer-consumer

```
/*
  Paul Nathan 2011. three-transfer.xc
  Produced as part of Master's Thesis work at the University of Idaho.

  This code is intended to transfer 1 word from 1 core to the next
  one, 3 times
 */
#include <platform.h>
#include <stdlib.h>
#include <stdio.h>

#define STARTTRACE(core) asm("starttrace" core ":");
#define ENDTRACE(core) asm("endtrace" core ":");

int main(void) {
    chan queue1;
    par {
        on stdcore[0]: {
                int outpacket = 0x0f;

                STARTTRACE("0");
                queue1 <: outpacket;
                queue1 <: outpacket + 1;
                queue1 <: outpacket + 2;
                ENDTRACE("0");
            }
        on stdcore[1]: {
                int inpacket;

                STARTTRACE("1");
                queue1 :> inpacket;
                queue1 :> inpacket;
                queue1 :> inpacket;
                ENDTRACE("1");
            }
    }
    return 0;
}
```

Observe Figure **??** renders the transferred values in decimal, not hexadecimal. Figure **??** shows the rapidly increasing complexity of the possible solutions to the problem of determining actual execution. Figure **??** visualizes the solution found by the rules pruning the graph of possible routes.

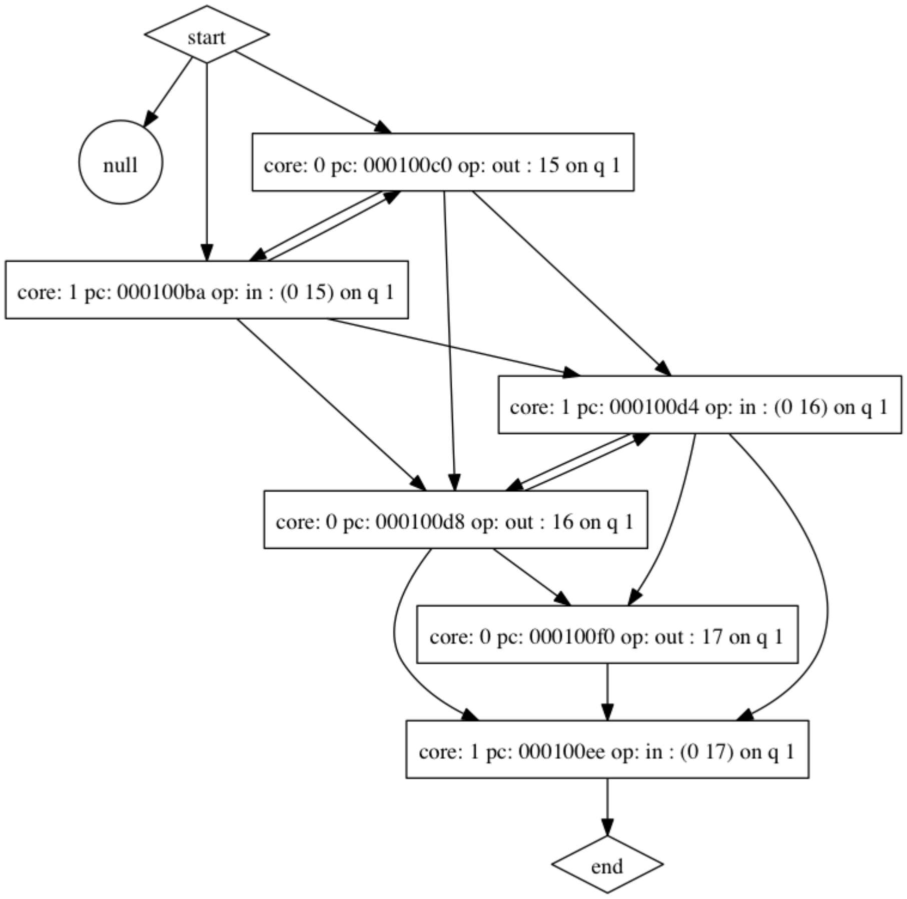
start
null
core: 0 pc: 000100c0 op: out : 15 on q 1
core: 1 pc: 000100ba op: in : (0 15) on q 1
core: 1 pc: 000100d4 op: in : (0 16) on q 1
core: 0 pc: 000100d8 op: out : 16 on q 1
core: 0 pc: 000100f0 op: out : 17 on q 1
core: 1 pc: 000100ee op: in : (0 17) on q 1
end

Figure 4.5: All path of the three transfers

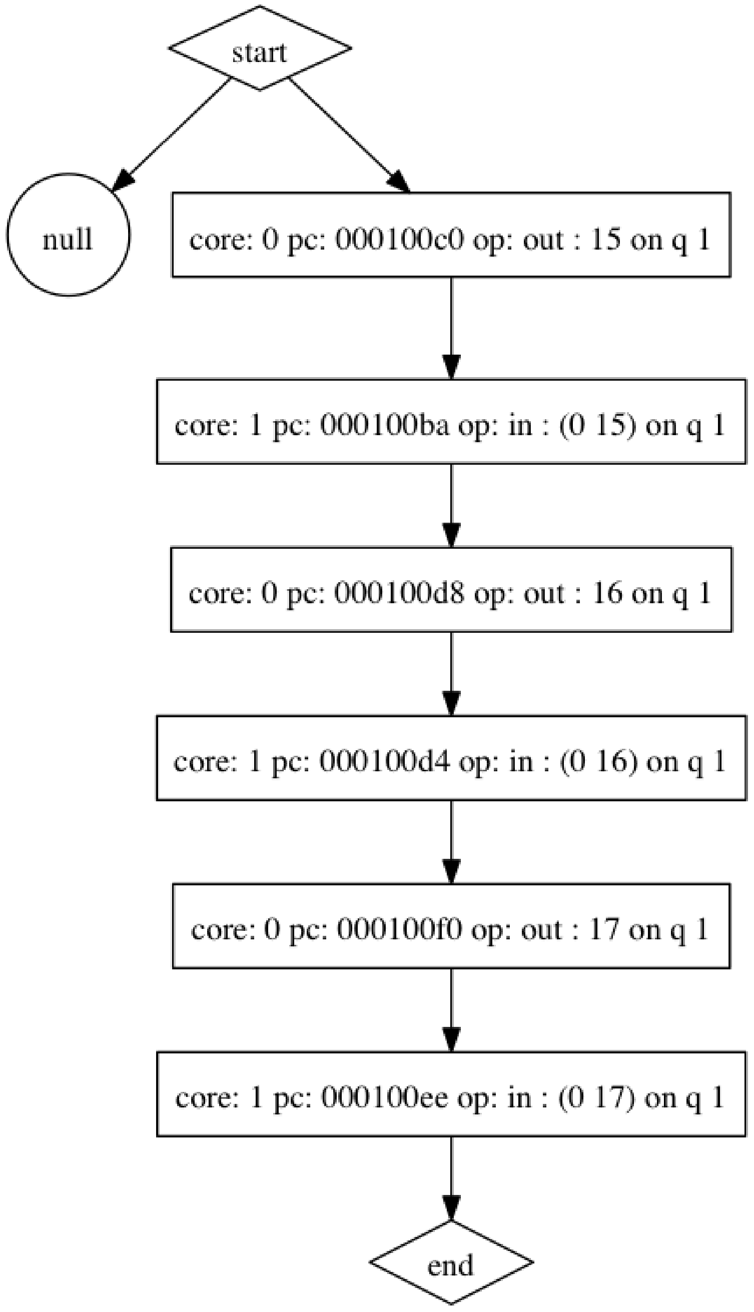
start
null
core: 0 pc: 000100c0 op: out : 15 on q 1
core: 1 pc: 000100ba op: in : (0 15) on q 1
core: 0 pc: 000100d8 op: out : 16 on q 1
core: 1 pc: 000100d4 op: in : (0 16) on q 1
core: 0 pc: 000100f0 op: out : 17 on q 1
core: 1 pc: 000100ee op: in : (0 17) on q 1
end

Figure 4.6: Path of the three transfers

4.1.4 Transaction

XC and the XCore architecture have a mode of communication called a transaction, where synchronization happens at the beginning of communication, but not in the middle. Here a transaction-driven model of the producer-consumer approach is presented. The denotation for the transaction are the `master` and `slave` keywords, indicating the sender and the receiver.

Listing 4.4: Transaction producer-consumer

```
/*
  Paul Nathan 2011: transaction.xc
  Produced as part of Master's Thesis work at the University of Idaho.

  This code is intended to run a transaction.
*/
#include <platform.h>
#include <stdlib.h>
#include <stdio.h>

#define STARTTRACE(core) asm("starttrace" core ":");
#define ENDTRACE(core) asm("endtrace" core ":");

int main(void) {
    chan queue1;
    par {
        on stdcore[0]: master {
                int outpacket = 0x0f;

                STARTTRACE("0");
                queue1 <: outpacket;
                queue1 <: outpacket + 1;
                queue1 <: outpacket + 2;
                ENDTRACE("0");
            }
        on stdcore[1]: slave {
                int inpacket;

                STARTTRACE("1");
                queue1 :> inpacket;
                queue1 :> inpacket;
                queue1 :> inpacket;
                ENDTRACE("1");
            }
    }
    return 0;
}
```

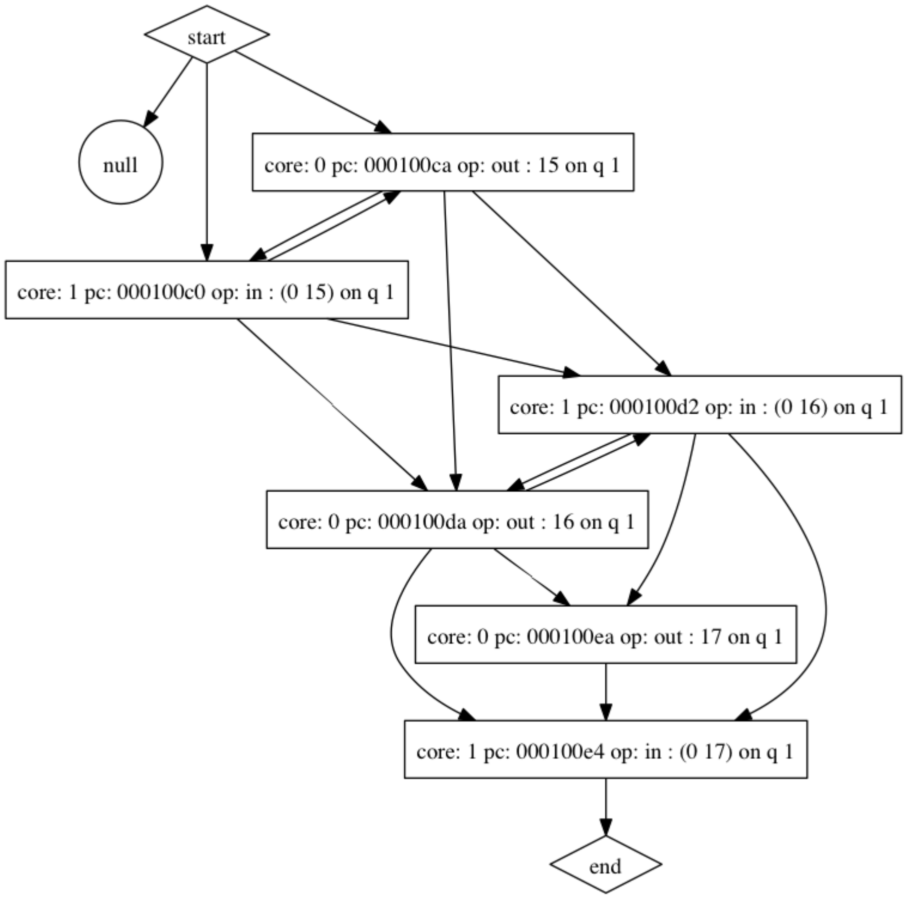
start
null
core: 0 pc: 000100ca op: out : 15 on q 1
core: 1 pc: 000100c0 op: in : (0 15) on q 1
core: 1 pc: 000100d2 op: in : (0 16) on q 1
core: 0 pc: 000100da op: out : 16 on q 1
core: 0 pc: 000100ea op: out : 17 on q 1
core: 1 pc: 000100e4 op: in : (0 17) on q 1
end

Figure 4.7: All-paths on the transaction

Observe that the generation of Figure **??** is identical to the non-transaction graph in Figure **??**. This is a correct correspondence: in both programs, the same data is expected to be transferred, in the same order.

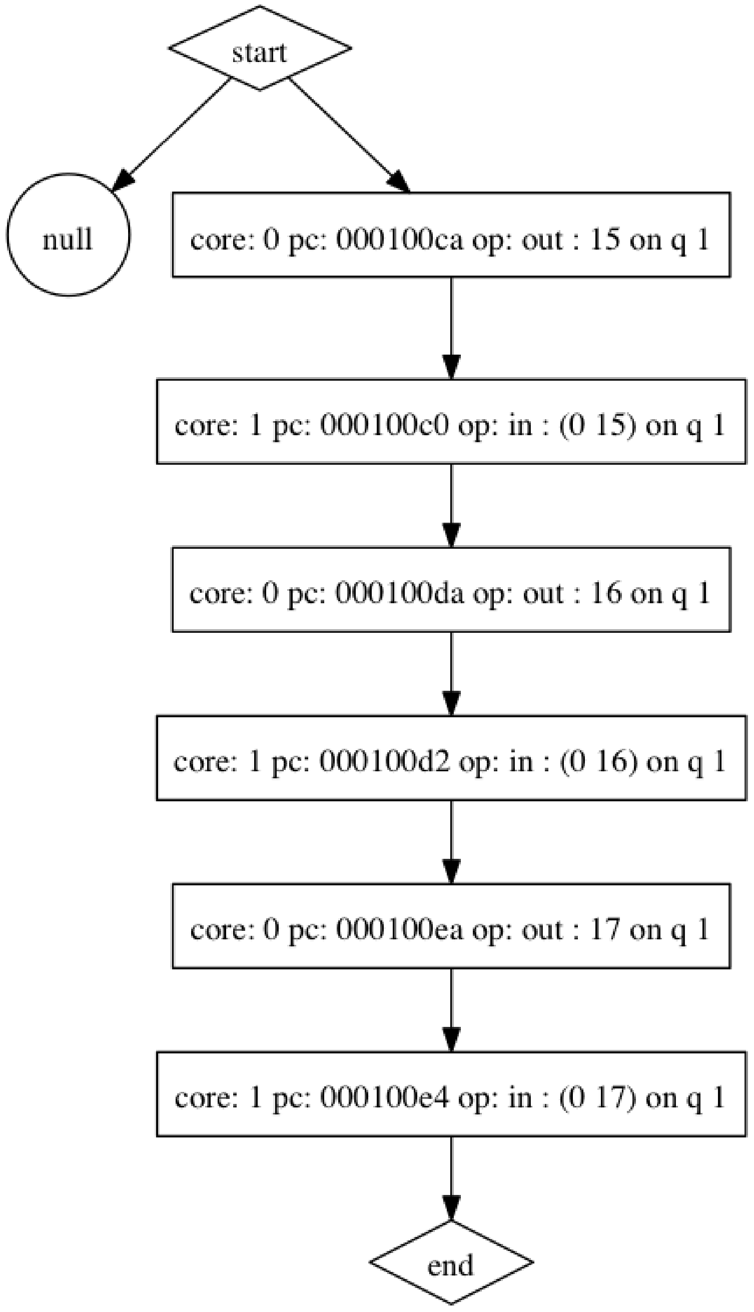
start
null
core: 0 pc: 000100ca op: out : 15 on q 1
core: 1 pc: 000100c0 op: in : (0 15) on q 1
core: 0 pc: 000100da op: out : 16 on q 1
core: 1 pc: 000100d2 op: in : (0 16) on q 1
core: 0 pc: 000100ea op: out : 17 on q 1
core: 1 pc: 000100e4 op: in : (0 17) on q 1
end

Figure 4.8: Path of the transaction

4.1.5 Pathological case

As a final test, a worst case scenario was constructed. To form this scenario, consider the following table of two-gram possibilities, drawn from the alphabet IN, OUT.

Observe that Table **??** forms a symmetrical matrix. The assumption of symmetri-

sender	receiver
IN IN	OUT OUT
IN OUT	OUT IN
OUT IN	IN OUT
OUT OUT	IN IN

Table 4.1: Two-gram Possibilities of Communication

sender	receiver
OUT	IN
IN	OUT

Table 4.2: Back and Forth Communication

cal operation on both cores is taken, and then, in order to test all possibilities, either the top two or the bottom two possibilities can be selected.

While more complex programs could be devised, other combinations of communication can be decomposed into one of the two following patterns.

Table **??** describes the simple exchange sequence. Table **??** describes the simple producer/consumer sequence already observed in Listing **??**.

The possibilities represented in Table **??** and Table **??** are combined in Listing **??**.

Listing **??**, `multi-transfer.xc`, is considerably more complicated than the previously demonstrated snippets. This is observable with the all paths graph in Figure **??**.

sender	receiver
OUT	IN
OUT	IN

Table 4.3: Multiple Out Communication

Listing 4.5: Multi Transfer

```
/*
  Paul Nathan 2011: multi-transfer.xc
  Produced as part of Master's Thesis work at the University of Idaho.

  This code is intended to run some exchanges in a fairly tricky set
  of patterns.
*/
#include <platform.h>
#include <stdlib.h>
#include <stdio.h>

#define STARTTRACE(core) asm("starttrace" core ":");
#define ENDTRACE(core) asm("endtrace" core ":");

int main(void) {
    chan queue;
    par {
        on stdcore[0]: {
                int packet = 0xf;
                STARTTRACE("0");
                queue <: packet;                // out
                queue :> packet;                // in
                queue <: packet + 1;            // out
                queue <: packet + 2;            // out
                queue :> packet;                // in
                ENDTRACE("0");
            }
        on stdcore[1]: {
                int packet;

                STARTTRACE("1");
                queue :> packet;                // in
                queue <: packet + 1;            // out
                queue :> packet;                // in
                queue :> packet;                // in, 2
                queue <: packet + 8;            // out
                ENDTRACE("1");
            }
    }
    return 0;
}
```

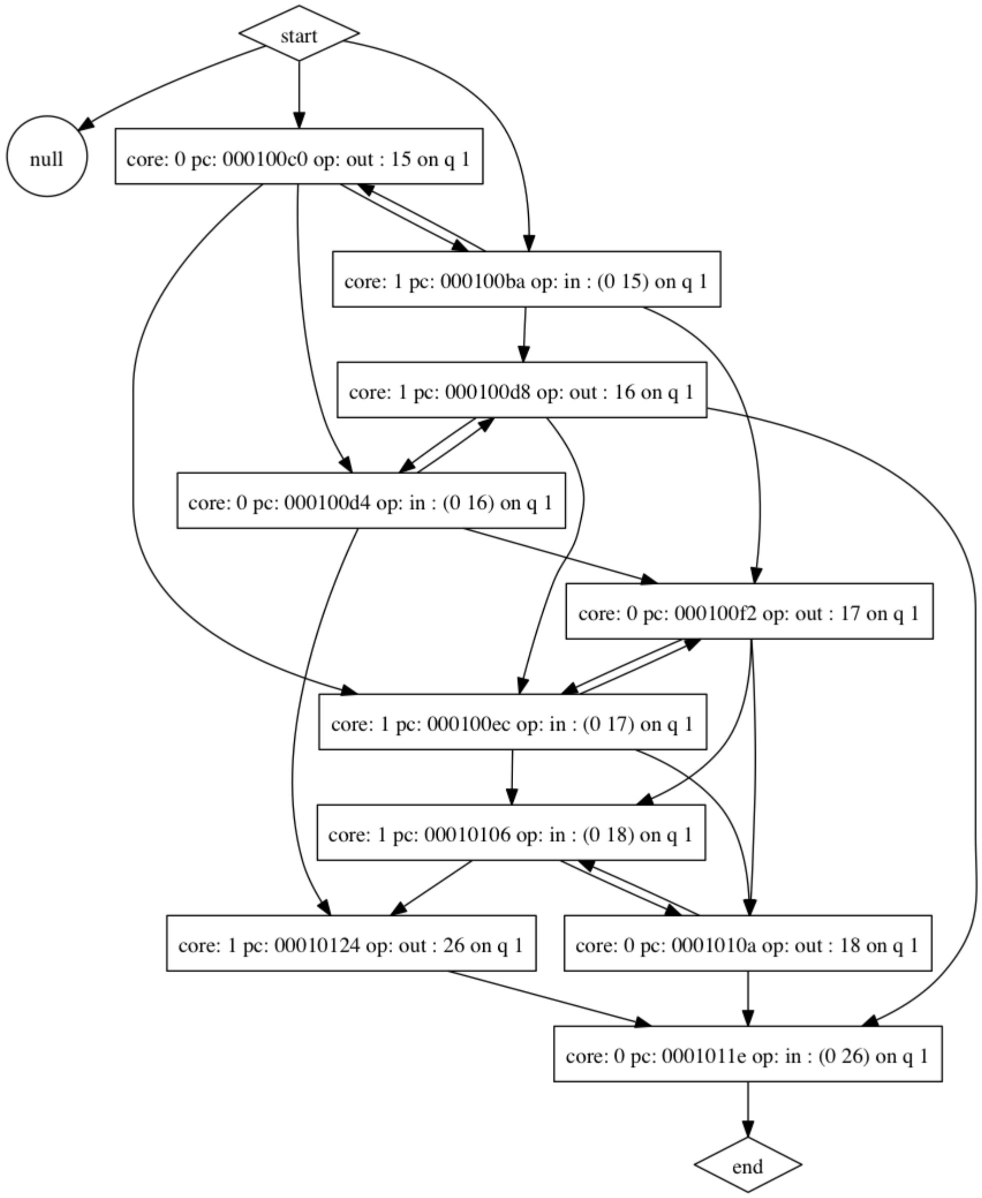
start
null
core: 0 pc: 000100c0 op: out : 15 on q 1
core: 1 pc: 000100ba op: in : (0 15) on q 1
core: 1 pc: 000100d8 op: out : 16 on q 1
core: 0 pc: 000100d4 op: in : (0 16) on q 1
core: 0 pc: 000100f2 op: out : 17 on q 1
core: 1 pc: 000100ec op: in : (0 17) on q 1
core: 1 pc: 00010106 op: in : (0 18) on q 1
core: 1 pc: 00010124 op: out : 26 on q 1
core: 0 pc: 0001010a op: out : 18 on q 1
core: 0 pc: 0001011e op: in : (0 26) on q 1
end

Figure 4.9: All-paths on the multi-transfer

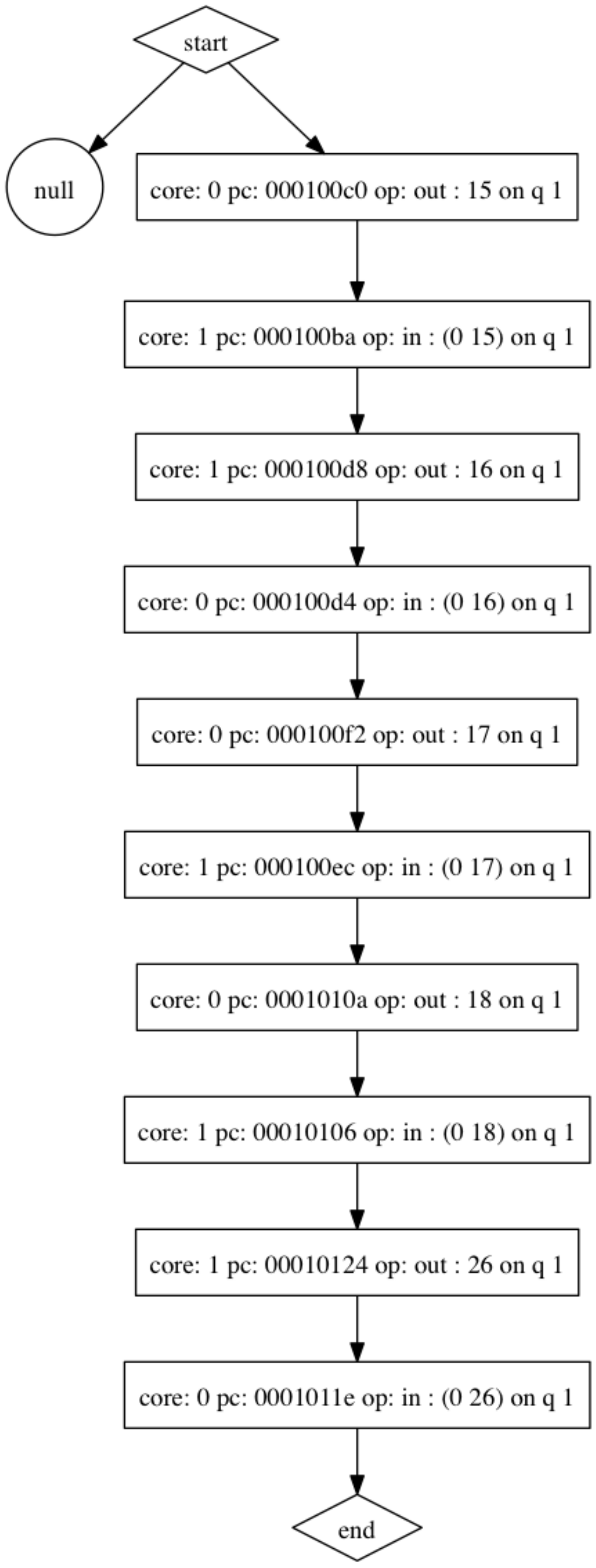
start
null
core: 0 pc: 000100c0 op: out : 15 on q 1
core: 1 pc: 000100ba op: in : (0 15) on q 1
core: 1 pc: 000100d8 op: out : 16 on q 1
core: 0 pc: 000100d4 op: in : (0 16) on q 1
core: 0 pc: 000100f2 op: out : 17 on q 1
core: 1 pc: 000100ec op: in : (0 17) on q 1
core: 0 pc: 0001010a op: out : 18 on q 1
core: 1 pc: 00010106 op: in : (0 18) on q 1
core: 1 pc: 00010124 op: out : 26 on q 1
core: 0 pc: 0001011e op: in : (0 26) on q 1
end

Figure 4.10: Path of the multi-transfer

4.2 Discussion

Chapter 3 reviews the software that produced the above graphs. When the semantic rules in Chapter 3 are applied to the graph, the experiments returned only a single graph considered "feasible" in the execution. This is probably due to the limitations in the current hardware scenario.

4.2.1 Limitations

The key inherent limitation in the experiment was the inability to accurately separate out multiple channels in the trace. This was caused by `xcc`'s (the XC compiler) generation of the assembly; the channels are defined before the START can be inserted. It was deemed to be out of the scope of this research to modify the compiler to instrument the generated code.

The simulator environment was chosen due to the limitation of the processor itself. As discussed in Section **??**, it was exceptionally difficult to perform tracing accurately and efficiently using the built-in single step mechanisms.

Another inherent limitation on the usability front is the amount of information that can be visualized. While that can be mitigated by shrinking the targeting window, that strategy is not practical if the bug being sought occurs over a wide span of execution (thus leading to a large amount of information visualized).

An artificial limitation is the restriction of the system from using the `timer` IO command. The `timer` generates IN instructions against a timer port on the chip. In order to trap this, the `GETR` assembly instruction will need to be parsed, and the resource ID returned will have to be tracked until the resource is released. As this is not a key concept in this report, `timer` tracing is not implemented.

4.2.2 Results

The key question for these experiments is, *did they prove the hypothesis*?

Hypothesis A *targeted replay mechanism* is an effective approach for a software-only debugging system.

In this report, an effective approach is defined to be able to determine information regarding the internal state of the system.

These experiments have demonstrated the following internal states of the programs under experiment:

1. Ordering of communication
2. Data in the communication

Further, the experiments demonstrated by construction that the *targeted trace* of the execution was able to abstract out information not needed for the inspection of the state.

The approach of analyzing the trace and rendering the analysis as a flowchart proved to indeed be an approach to view the communications that provided state. Interestingly, under the given limitations, only one possible event ordering was possible in the experiments, even for the complex Listing **??**.

Chapter 5

Conclusions

A targeted trace scheme has been demonstrated, within limitations, to have viability in a simulation of an embedded system environment. In order for targeted trace schemes to be exploited fully, the total environment *must* support the ability to unambiguously trace execution from the beginning of the probe period to the end of the probe period. However, in the case studied in this research, the hardware environmental support was found to be lacking for the purposes of targeted trace, causing a simulator to be used instead. Visualization is a known technique for gaining understanding of how computer systems interact and this work demonstrates an application of visualization of communication between system components. Given sufficient relaxing of the limitations, targeted trace analysis offers a both a theoretically sound and pragmatically workable mechanism to build debugging systems for the understanding of software state.

In general, this research demonstrated a highly limited application of targeted trace, but did not provide a complete demonstration of the hypothesis. It is hoped that the Targeted Trace Algorithm can be applied in other research efforts.

Future directions of this work can be immediately extended in two different directions. The first and most obvious direction is to generalize the software onto different architectures, including extending the kind of events traced and the abstraction of the events traced. Another approach, is to examine the concept of the targeted trace in depth. With the increasing size of modern computer software, it becomes more and more difficult to trace out exactly what the particular issues are without intense personal knowledge of the system under development. Being able to insert software probes into the system and gain an understanding of the execution without having to have a comprehensive awareness of the entire system is expected to lead to a fruitful result. An immediate extension of Algortithm **??** given in Chapter 3 is to improve performance by applying online pruning to only process paths that need to be processed.

Targeted trace debugging has not been previously explored indepth in the literature, and the author of this thesis hopes that it will prove to be useful in future research and endeavors.