Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Virtual Machine Based Debugging for occam-π

Carl G. RITSON and Jonathan SIMPSON

Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, England.

{c.g.ritson, j.simpson}@kent.ac.uk

Abstract. While we strive to create robust language constructs and design patterns which prevent the introduction of faults during software development, an inevitable element of human error still remains. We must therefore endeavor to ease and accelerate the process of diagnosing and fixing software faults, commonly known as *debugging*. Current support for debugging occam-π programs is fairly limited. At best the developer is presented with a reference to the last known code line executed before their program abnormally terminated. This assumes the program does in fact terminate, and does not instead live-lock. In cases where this support is not sufficient, developers must instrument their own tracing support, "printf style". An exercise which typically enlightens one as to the true meaning of concurrency... In this paper we explore previous work in the field of debugging occam programs and introduce a new method for run-time monitoring of occam-π applications, based on the Transterpreter virtual machine interpreter. By adding a set of extensions to the Transterpreter, we give occam-π processes the ability to interact with their execution environment. Use of a virtual machine allows us to expose program execution state which would otherwise require non-portable or specialised hardware support. Using a model which bears similarities to that applied when debugging embedded systems with a JTAG connection, we describe debugging occam-π by mediating the execution of one execution process from another.

Keywords. concurrency, CSP, debugging, occam-pi, virtual machine.

Introduction

While a program may be correct by design, as long as there remains a gap between that design and its implementation then an opportunity exists for errors to be introduced. In the ideal world our implementations would be automatically verified against higher-level specifications of our designs, but despite much work in the area, such checking is not yet supported for all cases in occam-π. Hence, support for finding of and fixing errors is essential to the rapid development of large-scale and complex applications. Debugging is the process of locating, analyzing, and correcting suspected errors [1].

The Kent Retargetable occam-π Compiler (KRoC) supports basic post-mortem debugging [2], which gives the developer access to the last executed line and process call at the time a fatal error such as integer overflow or array bounds is detected. This provides the developer with a starting point for debugging and assists with the correction of many trivial errors; however, it does not elucidate the, often complex, interactions which lead up to application failure, nor does it provide any assistance for non-fatal errors. The work presented in this paper is an extension of earlier work and is aimed at providing the developer with uniform and accessible tracing and replay facilities for occam-π applications.

Without any explicit debugging support developers must implement their own tracing support. This typically takes the form of a shared messaging channel along which component processes emit their status. The programmer writes "print" commands into critical sections of the application and then views the trace output. Concurrency, however, creates many issues for this debugging approach. Buffering on the messaging channel or the presence of multiple threads of execution will cause the trace output to deviate from the actual execution flow of the application. This hinders the detection of errors in the interaction between processes. In turn, the introduction of the messaging channel itself, a new shared resource on which component processes synchronise, subtly changes the scheduling behaviour of the program obscuring race condition related faults. This is often call the *probe effect* and we describe it further in section 5.1.

In this paper we briefly review previous work in the field of parallel application debugging, and more specifically the debugging of occam programs (section 1). Following on we introduce a new method for run-time monitoring of occam-π applications, based on the Transterpreter [3] virtual machine interpreter. Section 3 describes how virtual machine instances can interact with the interpreter to intercede on each other's execution. This allows an occam-π process to mediate the execution of another occam-π program, a model which bears similarities to that applied in debugging embedded systems with a JTAG connection [4]. Our new extensible bytecode format which provides access to debugging information is detailed in section 4. In section 5 we describe additional run-time debugging support we have added to the Transterpreter virtual machine, and then in section 6 we apply it and virtual machine introspection to the task of tracing occam-π programs. Conclusions are presented in section 7, and pointers to future work in section 8.

1. Related Work

Viewing the state of an executing program is one method of obviating its behaviour and debugging program code. Past work in the area of debugging occam has concentrated heavily on networks of Transputers, as these were the primary target for large scale occam application development. As such, tracing and debugging of program execution across networks of processors increased the complexity of past solutions. In addition to this observation, past solutions can be divided into those that traced and interacted with running programs and those that acted on stored post-mortem traces.

Stepney's GRAIL [5] extracted run-time data from a running network of Transputers, instrumenting the target occam program at compile-time with a modified version of the INMOS compiler and extracting the data over a custom bus external to the Transputer network [6], so as not to interrupt the communications occurring on the communications links. Maintaining the original timings and communication semantics as a program not being debugged is critical.

Cai and Turner identified in [7] the danger of added debugging hook processes altering the timing of a parallel program and propose a model under which a *monitor* has execution control over all processes and a logical clock is maintained for the execution of each process. The logical clock provides a way to measure and balance the interaction added to the network by the monitor hooks, and is similar to the approach we propose in section 5.1. The paper also identifies the problem of maintaining timings for real-time systems whilst monitoring the system, suggesting the use of dummy test-sources or buffering of real-time data.

Debugging methodologies that require annotations or changes to the code have the potential to introduce additional complexity, changing the run-time dynamics of the program. May's Panorama [8], designed for the debugging of message-passing systems, identifies an approach for post-mortem debugging which records a log of communications. Replaying

these communications offers a look at the particular execution of the program whilst introducing only a minimal overhead at run-time. The Panorama system had the downside of requiring all communications to be modified to use a different call and recompiled against a custom library. This need for program modification, instrumentation or annotation is common to many approaches that do not have the benefit of an interpreted run-time environment.

The INMOS Transputer Debugging System (TDS) [9] allowed the user to inspect the occam source and display run-time values of variables. Programs for use with TDS were compiled in a debug mode which embedded extra information: workspace offsets, types and values of constants, protocol definitions, and workspace requirements for procedures and functions. The TDS provides access to running processes in the network by "jumping through" channels to the process on the other end of the channel. Deadlock detection requires source modification, as a deadlocked process can not be jumped to using the channel jump system. Processes suspected of causing deadlock must be modified to include a secondary process and channel which will not deadlock and allows a jump into the process for state inspection.

Finally, Zhang and Marwaha's Visputer [10] provided a highly developed visual tool for editing and analysing occam process networks. An editing suite allowed the assisted building of programs using toolkit symbols. A network editor facilitated the configuration of networks of Transputer processors. Pre-processing of source files inserted instrumentation library calls to support post-mortem debugging and performance profiling. A static analyser also predicted performance and detected deadlocks. Importantly Zhang and Marwaha pointed out that occam "has a language structure that is highly suitable for graphical representation". The work presented in this paper is intended to provide a means of extracting the information required to exploit this feature of occam.

2. The Transterpreter Virtual Machine

The Transterpreter, or TVM (Transterpreter Virtual Machine), is a virtual machine interpreter for running occam-π applications compiled to a modified form of Transputer bytecode [3]. Written in platform independent ANSI C, the TVM emulates a hybrid T8 Transputer processor. Most T8 instructions are supported, with additional instructions added to support dynamics and mobility added by occam-π.

The Transputer was a three-place stack machine, and executed a bytecode where the most common instructions and their immediates required only a single byte to represent. Large instructions were composed from sequences of smaller instructions prior to execution. Hence the Transputer bytecode is compact, and simple to interpret. A modified version of the INMOS compiler with occam-π support is used to generated Extended Transputer Code (ETC), which is then converted by a linker into bytecode for the TVM.

The TVM has a very small memory footprint making it suitable for use on small embedded systems, such as the LEGO Mindstorms RCX [11] and Surveyor SRV-1 [12] mobile robotics platform. The work presented in this paper focuses on execution on desktop class machines; however, the portability of the bytecode representation used by the TVM allows emulation of embedded systems on a desktop machine. We envisage that the techniques detailed here be used in such a manner as to debug embedded application code via emulation when the target platform has insufficient resources to support debugging in-place.

3. Introspection

Recently, we have transitioned the TVM from a static library design to a fully re-entrant implementation. This shift allows us to run multiple virtual Transputers concurrently, or at least with the appearance of concurrency. The virtual Transputers can communicate with each

other, or more specifically processes in one TVM instance can communicate with processes in another, efficiently and transparently. Being software defined, our Transputers are not restricted in the number of communications links they support, and hence one virtual link is provided per-shared channel, freeing the programmer from multiplexing a fixed number of links. Mobile data [13] and mobile channels [14] are also supported, allowing the construction of arbitrary process networks which in turn span multiple virtual machines.

Each TVM instance has its own registers and run-queues. Instances can be scheduled cooperatively (only switched when they relinquish control), or pre-emptively (time-sliced). The specific algorithm used to schedule between TVM instances is defined by the virtual machine wrapper, for most purposes a round-robin algorithm is used (the same as occam processes); however, priority is also supported.

On a small robotics platform, the Surveyor Corporation SRV-1 [12], we have used multiple TVM instances to execute a cooperatively scheduled firmware and a time-sliced user application. The user application is loaded at run-time, and may be swapped out, terminate or even crash, without interfering with the firmware. The firmware executes as required, mediating access to the hardware on behalf of the user application.

Our virtual Transputers, TVM instances, give us the isolation and encapsulation we need to start allowing one occam-π program to mediate and intercede on the execution of another. This concept, of multiple concurrent virtual machine interpreters, underpins the debugging framework presented in the rest of this paper.

3.1. Interface

The application running in each virtual machine instance can access the virtual machine runtime via a special `PLACED` channel. Channel read requests to the `PLACED` channel return a *channel bundle* (channel type [14]) which provides request and response channels for the manipulation of the current virtual machine instances and the creation of new sub-instances. For each sub-instance created a further channel bundle is returned that can be used to control the instruction-by-instruction execution of the sub-instance.

The virtual machine interface channel also provides access to the interpreter's bytecode decoder. Using this interface, a virtual machine can load bytecode into new sub-instances, and access additional information stored in the bytecode. Details of our bytecode format and decoder interface can be found in section 4.

Having decoded bytecode we created a VM instance associated with it. Once *top level process* parameters are supplied, and the instance started, the control interface can be used to mediate its execution in a number of ways. The following subsections detail requests which can be used to control execution.

3.1.1. run

Execute bytecode for a number of instruction dispatches, or until a breakpoint or error is encountered. With the exception of breakpoints this causes the sub-instance to act as a normal virtual machine instance. At present this operation is implemented synchronously and is uninterruptable, blocking execution of the parent virtual machine. However, there is no reason that this could not be enhanced to permit interleaved execution. Where processing facilities exist the sub-instance could also be executed concurrently on a separate processor.

3.1.2. step

Decode and dispatch a single bytecode instruction. Feedback is given when the instruction is decoded and then after it is dispatched, allowing the supervising process to keep track of execution state without querying the entire virtual machine state.

3.1.3. dispatch

Execute, *dispatch*, an instruction not contained in the program bytecode. This can be used to alter program flow, for example to execute a *stop process* instruction to pause the running process and scheduling the next. Alternatively this request can be used to inject a completely new stream of instructions into the virtual machine instance.

3.1.4. get.state / set.state

Get and set requests for the state provide access to the virtual machine registers, instruction pointer, operand stack and clock. By combining these requests with the *dispatch* request a debugger can save virtual machine state, change processes and later restore state.

3.1.5. read / write

Read and write requests for all basic types provide access to the virtual machine's memory. As these can only be executed when the virtual machine is stopped, they have predictable results outside their influence on program behaviour. If virtual memory is in use then address translation occurs transparently.

4. Bytecode

To support our debugging enhancements we have developed a new extensible bytecode format for the TVM. Until the introduction of this new format, a number of different fixed formats were used for the different platforms supported by the TVM. By creating a new format we have unified bytecode decoding support code, and have removed the need to rewrite the decoder when the bytecode is extended.

4.1. Encoding

We call our new encoding *TEncode*. TEncode is a simple binary markup language, a modified version of IFF.

TEncode streams operate in either 16-bit or 32-bit mode, which affects the size of integers used. A stream is made up of *elements*. Each element consists of a 4-byte identifier, followed by an integer, then zero or more bytes of data. Elements always begin on integer aligned boundaries, the preceding element is padded with null bytes ("\0") to maintain this. All integers are big-endian encoded, and of consistent size (2-bytes or 4-bytes) throughout a stream. Integers are unsigned unless otherwise stated.

```
Identifier  := 4 * BYTE
Integer     := INT16 / INT32
Data        := { BYTE }
Padding     := { "\0" }
Element     := Identifier, Integer, Data, Padding
```

Identifiers are made up of four ASCII encoded characters stored in 8-bit bytes, and are case-sensitive. The last byte indicates the type of data held in the element. The following types are defined:

Byte string . Integer field encodes number of bytes in Data field, excluding padding null-bytes.

Integer . Integer field encodes signed numeric value, no Data bytes follow.

List of elements . Integer field encodes number of bytes in Data field which will be a multiple of the integer size, Data field contains nested elements, and may be parsed as a new or sub-Stream.

String (UTF8 null-terminated) . Integer field encodes number of bytes in Data including null terminator, but not padding.

Unsigned integer . Integer field encodes unsigned numeric value, no Data bytes follow.

With the exception of signed and unsigned integer types, the Integer of all elements defines the unpadded size of the Data which follows. Decoders may use this relationship to skip unknown element types, therefore this relationship must be preserved when adding new types.

A TEncode stream begins with the special `tenc` or `TEnc` element, the integer of which indicates the number of bytes which follow in the rest of the stream. A lower-case `tenc` indicates that integers are small (16-bit), whereas an upper-case `TEnc` indicates integers are large (32-bits). The `TEnc` element contains all other elements in the stream.

```
ByteString   := 3 * BYTE, "B", Integer, Data [, Padding ]
SignedInt    := 3 * BYTE, "I", Integer
ElementList  := 3 * BYTE, "L", Integer, Stream
UTF8String   := 3 * BYTE, "S", Integer, {<character byte>},
                "\0" [, Padding ]
UnsignedInt  := 3 * BYTE, "U", Integer

Element      := ByteString / SignedInt / ElementList / UTF8String /
                UnsignedInt

Header       := ("tenc", INT16) / ("TEnc", INT32)
Stream       := { Element }
TEncode      := Header, Stream
```

Decoders ignore elements they do not understand or care about. If multiple elements with the same identifier exist in the same stream and the decoder does not expect multiple instances then the decoder uses the first encountered. When defining a stream, new elements may be freely added to its definition across versions; however, the order of elements must be maintained in order to keep parsing simple.

4.2. Structure

The base Transterpreter bytecode is a TEncode stream defined as follows:

```
TEnc <stream length>
  tbcL <length>
    endU <endian (0=little, 1=big)>
    ws U <workspace size (words)>
    vs U <vectorspace size (words)>
    padB <length> <padding>
    bc B <length> <bytecode>
```

As previously stated, the `TEnc` element marks the beginning of a TEncode stream. The stream contains a number of `tbcL` elements, each defines a bytecode chunk. A stream may contain multiple bytecode chunks in order to support alternative compilations of the same code, for example with different endian encodings. The `endU` element specifies the endian type of the bytecode. The `ws U` and `vs U` elements specify the workspace and vectorspace memory requirements in words. The `padB` element ensures there is enough space to in-place decode the stream. Finally, the `bc B` element contains the bytecode which the virtual machine interpreter executes.

Following the mandatory elements a number of optional elements specify additional properties of a chunk of bytecode. By placing these elements after the mandatory elements a stream decoder on a memory constrained embedded system can discard all unnecessary

elements, such as debugging information, once the mandatory elements have been received and decoded.

A `tlpL` element defines the arguments for the *top level process* (entry point). The foreign function interface table, and associated external library symbols are provided in a `ffiL` element. A symbol table, defining the offsets, memory requirements and type headers of processes within the bytecode is provided by a `stbL` element. Finally a `dbgL` element specifies debugging information.

The debugging information takes the following form:

```
dbgL <length>
  // File names
  fn L <length>
    fn S <length> <file name>
  // Line Numbering Data
  lndB <length>
    <bytecode offset> <file index> <line number>
    ... further entries ...
```

A table of source file names is defined by `fn S` elements. A table of integer triples is then specified by the `lndB` element. The integers in each triple correspond to a bytecode offset, the index of the source file (in the source file name table), and the line number in the specified source file. The table is arranged such that bytecode offsets are ascending and offsets that fall between entries in the table belong to the last entry before the offset. For example if entries exist for offsets 0 and 10, then offset 5 belongs to offset 0.

4.3. In-place Decoding

On memory constrained embedded platforms it is important to minimise and preferably remove dynamic memory requirements. For this reason we designed our new bytecode format so as not to require dynamic memory for decoding. We achieve this by providing for rewriting of the bytecode in memory as it is decoded. The C structure `tbc_t` is placed over the memory of the `tbcL` element of the TEncode stream and the component fields written as their TEncode elements are decoded. The `bytecode` field is a pointer to the memory address of the data of the `bc B` element. `tlp`, `ffi`, `symbols` and `debug` fields are pointers to the in-place decodes of their associated elements. The pointers are `NULL` if the stream does not contain the associated element.

```
struct tbc_t {
  unsigned int  endian;
  unsigned int  ws;
  unsigned int  vs;

  unsigned int  bytecode_len;
  BYTE         *bytecode;

  tbc_tlp_t    *tlp;
  tbc_ffi_t    *ffi;
  tbc_sym_t    *symbols;
  tbc_dbg_t    *debug;
};
```

4.4. Interface

As with the introspection interface discussed in section 3, the bytecode decoder also has a channel interface accessible from within a virtual machine instance. When passing an encoded bytecode array to the virtual machine, a channel bundle is returned which can be used

to access the decoded bytecode. The following subsections detail some of the available requests.

4.4.1. create.vm

Create a new virtual machine instance based on this bytecode. A control channel bundle is returned for the new instance.

4.4.2. get.file

Translate a source file index to its corresponding file name. The file name is returned as a mobile array of bytes.

4.4.3. get.line.info

Look up the line numbering information for a bytecode address. The source file index and line number are returned if the information exists.

4.4.4. get.symbol / get.symbol.at

Look up a process symbol by name or bytecode address. The symbols bytecode offset, name, type description and memory requirements are returned if the symbol exists.

4.4.5. get.tlp

Access information on the top-level-process (entry point), if one is defined in the bytecode. If defined, the format mask and symbol name of the process are returned. This information is required to setup the entry point stack.

5. Debugging Support

We have added additional features to the TVM in order to support debugging. The following section details some of the more significant changes we have made and how they may be applied to the problem of debugging occam-π programs.

5.1. The Probe Effect

The *probe effect* is a term used to describe the consequence that observing a parallel or distributed system may influence its behaviour. This is a reminiscent of the Heisenberg uncertainty principle as applied in quantum physics. If the decisions made in non-deterministic elements of a program differ between runs under and not under observation, then the observed behaviour will not be the actual operation behaviour of the program. The impact of this on debugging is that, on observation of a known faulty program, errors do not occur or different errors appear.

To prevent the occurrence of the probe effect we must ensure the non-deterministic behaviour of the program is not altered by observation. In occam-π there are two sources of programmed non-determinism: alternation constructs and timers. A program free of these constructs is completely deterministic; in fact a program is still deterministic in the presence of simple timer delays [15].

Previous work has shown that by maintaining the sequence of events in a program and recording the decisions made by non-deterministic elements, it is possible to replay parallel programs [16]. In the work presented here we do not attempt to constrain the non-determinism which occurs from external inputs to the program, only to control the internal non-determinism. Given that the implementation of virtual machine constructs such as alter-

nation is unaffected by changes made to monitor program execution, we need only manage the non-determinism caused by timers.

Our present approach to constraining timer based non-determinism is to use a logical time clock, as opposed to a real time clock [7]. This logical clock ticks based on the count of instructions executed. By applying a suitable multiplier the instruction count is scaled to give a suitable representation of time.

The accuracy of the logical clock's representation of time depends on the scaling value used. At program start-up the virtual machine (or firmware components of it), calculate the average execution time of a spread of instructions, and use this to derive a scaling factor. This scaling factor is then periodically adjusted and offset, such that the clock time matches the actual time required to execute the program so far. If virtual machine execution pauses, for example waiting on the timer queue, then an adjustment need only be stored to the new time when execution resumes.

In order to replay a program's execution we need only store the initial scaling factor and the instruction offset and value of subsequent adjustments. If adjustments are stored in three 32-bit integers, then any single adjustment will be 12 bytes in size. Assuming adjustments are made approximately once a second then around 40KiB of storage is required for every hour of execution. As the program is replayed, adjustments are applied at their associated instruction offsets irrespective of the actual execution speed of the program.

5.2. Memory Shadowing

We have added support for shadowing each location of the *workspace* (process stack) with type information of the value it holds. This information assists in the debugging of programs in the absence of additional compiler information about the memory layout. The type mapping is maintained at run-time for the virtual machine registers and copied to and from the shadow map as instructions are interpreted. For example if a pointer to a workspace memory location is loaded and then stored to another memory location, then the type shadow will indicate the memory is a workspace pointer. The shadow types presently supported are:

Data: The memory location holds data of unknown type.
Workspace: Pointer to a workspace memory location.
Bytecode: Pointer to a location in bytecode memory, e.g. an instruction or constant pointer.
Mobile: Pointer into mobile memory.
Channel: Memory is being used as a channel.
Mobile type: Mobile type pointer or handle.

By iterating over the shadow of the entire memory, a debugger can build an image of a paused or crashed program. To assist in this, the memory shadow holds flags about whether the memory is utilised or not. Workspace words are marked as in-use when they write to, and marked as free whenever a process releases them via the *adjust workspace* instruction. As memory known to be unused need not be examined, false positives and data ghosts are reduced.

In addition to a utilisation flag, call stack and channel flags are also recorded. Whenever a *call* instruction is executed, the workspace word used to store the return address is marked. This allows later reconstruction of the call stack of processes. Any word used as a channel in a channel input or output instruction is also marked to facilitate the building of process network graphs. If a process's workspace holds a channel word, or a pointer to a channel word, then we can assume it is a communication partner on that channel. When more than two such relationships exist then pointers can be assumed to represent the most recent partners on the channel, for example a channel declared in a parent process being used by two processes running in parallel beneath it.

6. A Tracing Debugger

The virtual machine extensions and interfaces so far detailed can be used to implement various development and debugging tools. In this section we will detail a tracing debugger.

6.1. Instancing Bytecode

First we must decode the bytecode we intend to run, this is done by making a `decode.bytecode` request on the introspection channel. Having successfully decoded bytecode we can request a virtual machine instance to execute it. The following code example demostrates decoding bytecode and creating a virtual machine instance. A virtual machine introspection bundle is assumed to be defined as `vm`.

```
MOBILE []BYTE data:
INT errno:
SEQ
  ... load bytecode into data array ...
  vm[request]  ! decode.bytecode; data
  vm[response] ? CASE
    CT.BYTECODE! bytecode:
    bytecode; bytecode
      -- bytecode decoded
      SEQ
        bytecode[request]  ! create.vm
        bytecode[response] ? CASE
          CT.VM.CTL! vm.ctl:
          vm; vm.ctl
            -- VM instance created
          error; errno
            ... handle VM creation error, e.g. out of memory ...
    error; errno
      ... handle decoding error, e.g. invalid bytecode ...
```

6.2. Executing Bytecode

Before we can execute bytecode in our newly created virtual machine instance we must supply the parameters for the top-level-process. In a standard occam-π program these provide access to the keyboard input, and standard output and error channels. In our example tracing code we will assume the program being processed has a standard top-level-process; however, to handle alternate parameter combinations the `get.tlp` (section 4.4.5) request would be used to query the bytecode. In the following code example we supply the top level parameters as channel bundles, preparing the bytecode for execution:

```
CT.CHANNEL? kyb.scr, scr.svr, err.svr: -- server ends
CT.CHANNEL! kyb.cli, scr.cli, err.cli: -- client ends
SEQ
  ... allocate channels ...
  -- fork off handling processes
  FORK keyboard.handler (kyb.cli, ...)
  FORK screen.handler (scr.svr, ...)
  FORK error.handler (err.svr, ...)
  -- set top level parameters
  vm.ctl[request]  ! set.param.chan; 0; kyb.scr
  vm.ctl[response] ? CASE ok
  vm.ctl[request]  ! set.param.chan; 1; scr.cli
  vm.ctl[response] ? CASE ok
```

```
vm.ctl[request]  ! set.param.chan; 2; err.cli
vm.ctl[response] ? CASE ok
```

Having provided the top level parameters we can begin executing instructions from the bytecode. We want to trace the entire execution of the program thus we use the `step` request on the VM control channel bundle. If we did not require information about the execution of every instruction then we could use the `run` request. The following code snippet demonstrates the `step` request, which returns two responses, one on instruction decode and one on dispatch:

```
IPTR iptr:
ADDR wptr:
INT op, arg:
SEQ
  vm.ctl[request]  ! step
  vm.ctl[response] ? CASE decoded; iptr; op; arg
  -- decoded instruction "op", with argument "arg"
  --   new instruction pointer: iptr
  vm.ctl[response] ? CASE dispatched; iptr; wptr
  -- dispatched instruction
  --   new instruction pointer: iptr
  --   new workspace pointer: wptr
```

6.3. Tracking Processes

We want to create a higher level view of the program execution than the raw instruction stream as it executes. Ideally we want to know the present line of execution of all the processes. To do this we need to track the executing processes and attribute instruction execution to them. We need to monitor *start process* and *end process* instructions, additionally we must track any instruction that alters the workspace pointer (stack) such as *call* and *adjust workspace pointer*.

The following code snippet converts the decoding and dispatching of instructions into a stream of `executing`, `start.process`, `end.process` and `rename.process` tagged protocol messages. Each message carries the process workspace, which acts as the process identifier. `rename.process` messages indicate a change of process identifier, or a shift in workspace pointer.

```
ADDR id:
IPTR iptr:
WHILE TRUE
  ADDR new.id:
  SEQ
    -- report present process position
    out ! executing; id; iptr
    -- start process and end process
    vm.ctl[request]  ! step
    vm.ctl[response] ? CASE decoded; iptr; op; arg
    IF
      op = INS.OPR
        VM.STATE state:
        SEQ
          vm.ctl[request]  ! get.state
          vm.ctl[response] ? CASE state; state
          CASE arg
            INS.STARTP
```

```
          -- process workspace is on the operand stack
          out ! start.process; state[stack][0]
        INS.ENDP
          out ! end.process; id
        ELSE
          SKIP
  TRUE
    SKIP
-- workspace pointer altering operations
vm.ctl[response] ? CASE dispatched; iptr; new.id
IF
  (op = INS.ADJW) OR (op = INS.CALL)
    out ! rename.process; id; new.id
  (op = INS.OPR) AND (arg = INS.RET)
    out ! rename.process; id; new.id
  TRUE
    SKIP
-- update workspace pointer
id := new.id
```

6.4. Visualisation

The bytecode decoder interface detailed in section 4.4 is used to look up the present source position of each process. This allows us to generate output on the current source file and line being executed. Visualising this information as a graph of active processes we can see the executing program, although the specifics of this visualisation is an area for future work (see section 8). Call graphs can be generated by monitoring *call* and *return* instructions, and by recording the process which executes a *start process* instruction as the parent of the process started, we can see the program structure.

Adding to the graph of active processes, we also build a graph of relationships between processes. Whenever a channel communication instruction is executed we record the channel pointer in a channel table. Processes which access the same channel become related; edges can be drawn between them in the graph of process nodes. On *adjust workspace pointer* instructions, and other memory deallocation instructions channel words which fall out of scope are deleted from the channel table. This information can be used to visualise the relationships between the active processes, either in real-time as communication occurs, or statically as a cumulative image recorded over time.

Figure 1 applies these techniques to generate an idealised visualisation of the *commstime.occ* example from the KRoC compiler distribution.

7. Conclusions

We have created a method for occam-π programs to inspect and intercede on the execution of other occam-π programs. This access is driven through a channel based interface; which permits reasoning about programs which use it, in-line with that applicable to a standard occam-π program. Using this new interface it is possible to develop debugging tools for occam-π using occam-π, which allows us to use a full parallel programming environment in order to tackle the challenges we face developing such tools.

By constraining the probe effect caused by non-determinism related to the use of time in programs, we in turn permit cyclic debugging: the process of repeated execution of a program with slight modifications to its composition of inputs in an attempt to locate and fix an error. Additionally, we expose as much memory typing information as we can in order to help build complete debugging tools.

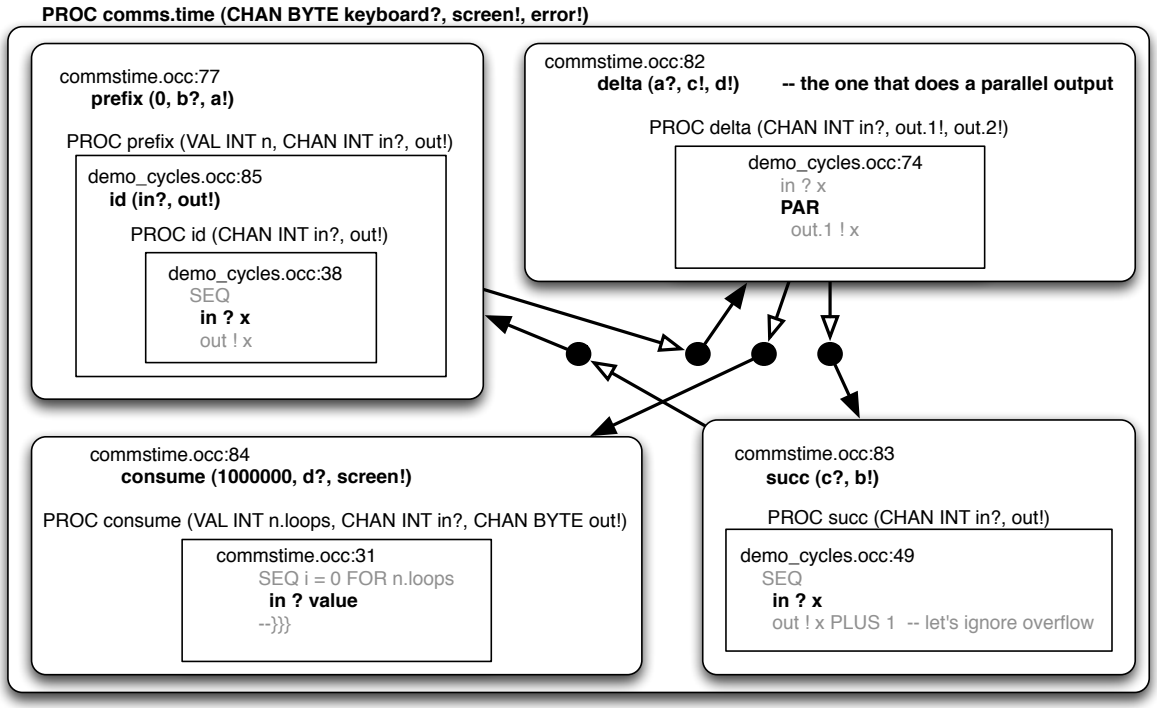

Figure 1. Idealised visualisation of commstime.occ. Curved boxes represent processes. Squared boxes represent calls in the call stack. Bold text is the present line being executed, as taken from the source file. Dots are channels, with arrowed lines representing communication requests. Hollow arrows are previously executed input or output, and filled arrows are active blocked requests.

Overall, we hope that the work presented in this paper will form the basis for developing visualising and debugging facilities for the occam-π tool-chain.

8. Future Work

Virtual machine sub-instances presently run synchronously to their parents, a logical enhancement to our work is the multiplex execution of virtual machine instances. Further to this, by applying algorithms developed for the KRoC runtime support library, CCSP, we expect to be able to extend the TVM to execute multiple instances concurrently on multi-core shared-memory hardware. These concurrently executing instances will be able to communicate, wait and synchronise on each other safely while maximising use of any underlying hardware parallelism.

Given the powerful underlying tool for inspecting the behaviour of running parallel programs we have developed, we would ideally like to make this power available in an easy to interpret graphical form. Exton identified four main areas of importance in the visualisation of program execution: abstraction, emphasis, representation and navigation [17]. Being able to provide a good abstraction level for observation of run-time activity, and the ability to adjust the level in tune with the user's expertise is critical to sensibly observing executing networks of processes. The design of a suitable visual model to represent the state of a running occam-π program is an as yet unexplored area, and may potentially provide useful insight to inform related work in visual design of parallel programs.

Dynamicism of the network topology and processes provides additional complexity to the debugging process beyond that encountered with occam 2 running on the Transputer. Dijkstra [18] states that "our powers to visualise processes evolving in time are relatively poorly developed", a statement that encapsulates some of the difficulty encountered when trying to reason about systems using dynamic process creation and mobile channels. Working

to allow end-users to explore the execution of their program and interact with its run-time behaviour to diagnose problems and mistakes would begin to provide some semblance of modern, user-friendly debugging to the occam-π tool-chain.

Acknowledgements

This work was funded by EPSRC grant EP/D061822/1. This work would not be possible if not for the past work and input of other researchers at the University of Kent and elsewhere, in particular, Christian Jacobsen, Matt Jadud and Damian Dimmich. We also thank the anonymous reviewers for comments which helped us improve the presentation of this paper.

References

[1] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.

[2] David C. Wood and Frederick R. M. Barnes. Post-Mortem Debugging in KRoC. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 179–192, sep 2000.

[3] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In Dr. Ian R. East, Prof David Duce, Dr Mark Green, Jeremy M. R. Martin, and Prof. Peter H. Welch, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 99–106. IOS Press, Amsterdam, September 2004.

[4] *IEEE Standard 1149.1-1990 Test Access Port and Boundary-Scan Architecture-Description*. IEEE, 1990.

[5] Susan Stepney. GRAIL: Graphical representation of activity, interconnection and loading. In Traian Muntean, editor, *7th Technical meeting of the occam User Group,Grenoble, France*. IOS Amsterdam, 1987.

[6] Susan Stepney. Understanding multi-transputer execution. In *IT UK 88, University College Swansea, UK*, 1988.

[7] Wentong Cai and Stephen J. Turner. An approach to the run-time monitoring of parallel programs. *The Computer Journal*, 37(4):333–345, March 1994.

[8] John May and Francine Berman. Panorama: a portable, extensible parallel debugger. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pages 96–106, New York, NY, USA, 1993. ACM Press.

[9] C. O'Neil. The TDS occam 2 debugging system. In Traian Muntean, editor, *OUG-7: Parallel Programming of Transputer Based Machines*, pages 9–14, sep 1987.

[10] K. Zhang and G. Marwaha. Visputer–A Graphical Visualization Tool for Parallel Programming. *The Computer Journal*, 38(8):658–669, 1995.

[11] Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. Mobile Robot Control: The Subsumption Architecture and occam-pi. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 225–236, Amsterdam, The Netherlands, September 2006. IOS Press.

[12] Surveyor Corporation SRV-1: `http://www.surveyor.com/blackfin/`.

[13] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 243–264, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[14] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part I. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 331–361, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[15] A. Cimitile, U. De Carlini, and U. Villano. Replay-based debugging of occam programs. *Software Testings, Verfication and Reliability*, 3(2):83–100, 1993.

[16] T.J. Leblanc and J.M. Mellor-Crummey. Debugging parallel programs with instant replay. *Computers, IEEE Transactions on*, C-36(4):471–482, April 1987.

[17] Chris Exton and Michael Kölling. Concurrency, objects and visualisation. In *ACSE '00: Proceedings of the Australasian conference on Computing education*, pages 109–115, New York, NY, USA, 2000. ACM Press.

[18] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.