**Extending programs with debug-related features, with application to hardware development**

Nik Sultana*b* Salvator Galea*b* David Greaves*b* Marcin Wójcik*b* Noa Zilberman*b* Richard Clegg*#*

Luo Mai*q* Richard Mortier*b* Peter Pietzuch*q* Jon Crowcroft*b* Andrew W Moore*b*

*b*Cambridge University *q*Imperial College, London *#*Queen Mary University, London

# Abstract

arXiv:1705.09902v1 [cs.PL] 28 May 2017

The capacity and programmability of reconfigurable hard- ware such as FPGAs has improved steadily over the years, but they do not readily provide any mechanisms for mon- itoring or debugging running programs. Such mechanisms need to be *written into* the program itself. This is done us- ing *ad hoc* methods and primitive tools when compared to CPU programming. This complicates the programming and debugging of reconfigurable hardware.

We introduce *Program-hosted Directability* (PhD), the extension of programs to interpret *direction commands* at runtime to enable debugging, monitoring and profiling. Nor- mally in hardware development such features are fixed at compile time. We present a language of directing commands, specify its semantics in terms of a simple controller that is embedded with programs, and implement a prototype for directing network programs running in hardware. We show that this approach affords significant flexibility with low im- pact on hardware utilisation and performance.

***Keywords*** debugging, FPGA, program directing, profiling, high-level synthesis, aspect-oriented programming

# Introduction

When debugging and monitoring programs running on mi- croprocessors we usually benefit from hardware support that is leveraged by an Operating System to inspect and modify running processes [(Gagnon et al. 2007).](#_bookmark44) But when a program is run on *reconfigurable hardware* platforms, one does not usually have an operating system, a notion of process, nor any hardware support for debugging. Programs must contain additional logic to enable *debugging*, *monitoring*, and *profil- ing* during their execution, because the environment does not provide visibility into running programs by default.

Field-Programmable Gate Array devices (FPGAs) are a form of programmable hardware consisting of a grid of logic blocks whose function and wiring can be flexibly recon- figured. FPGAs are used to perform functions for which a full-featured general-purpose CPU is not appropriate. For such functions, FPGAs can operate at a higher throughput and consume much less electricity than CPUs [(Mittal 2014,](#_bookmark62)

§4.4). This makes FPGAs especially appealing for some problems and environments, a recent example being data- centres [(Putnam et al. 2014).](#_bookmark71)

Despite their appeal as a computing device, the pro- grammability of FPGAs has been hampered by the need for low-level hardware-description languages traditionally used to program them, such as Verilog and VHDL, which “requires programmers to assume the role of hardware de- signers” [(So et al. 2002).](#_bookmark79) Research has yielded various ap- proaches for *high-level synthesis* (HLS): this tends to consist of using (fragments of) existing languages, such as C or Java, to describe hardware.

In addition to a *programmability* gap between FPGAs and CPUs, there is a *debuggability* gap that has received far less attention. The programmability of FPGAs has im- proved over the years, but they are not *debuggable* by de- fault [(Potkonjak et al. 1995).](#_bookmark69) FPGAs provide no visibility into the running program, and their standard tooling provides very limited support for this. For full visibility one could *simulate* the program, (e.g., on a workstation) but the simu- lation can be slower by a factor of 106 [(Camera et al. 2005)](#_bookmark33) because of the sheer amount of detail that must be simulated.

In this paper we propose to improve the debuggability of programs running on FPGAs by using a domain-specific control language inspired by *program directing* [(Sosicˇ 1992)](#_bookmark81) to generate in-program support for debugging. In addition to debugging, this can be used to monitor and profile programs.

We call our approach *Program-hosted Directability* (PhD). It involves extending the user’s program to service *direc- tion commands* at runtime. Extending the program involves inserting (i) named *extension points* which can contain runtime-modifiable code in a computationally weak lan- guage (no recursion), and (ii) state to be used for book- keeping by that code, to implement direction features.

For example, the direction command “trace *X* max\_trace\_idx” (where *X* is a variable in the program’s source code) logs updates made to a variable, and appends these updates to a buffer (of size “max\_trace\_idx” for later inspection). We translate this command into (i) a snippet that gets injected into the program (Figure [1)](#_bookmark0) and (ii) a mod- ification of the program to allocate space for this snippet,

**if** V\_ trace\_ idx < max\_ trace\_ idx **then** V\_ trace\_ buf [ V\_ trace\_ idx ] := V; **inc** V\_ trace\_ idx ;

**continue else**

**inc** V\_ trace\_ overflow ;

**break**

**Figure 1.** Code that implements the direction command “trace *X* max\_trace\_idx”. If the buffer is not full then the new value of X is logged, the index incremented, and control is handed back to the program that hosts this code. Other- wise indicate depletion of the associated buffer resource and break the program’s execution.

and for state require by this snippet: such as the variable V\_trace\_idx, and array V\_trace\_buf. More examples are given in [§3,](#_bookmark3) and the overall scheme is shown in Figure [2.](#_bookmark1)

Idiomatic direction features (such as tracing, breakpoints, etc) are compiled down to a weak language and executed by an interpreter embedded in the program. For consistency with the description of the interpreter’s behaviour as hard- ware, we refer to it as a *controller*. The controller is invoked by extension points, which are inserted into the program through transformation. In the above example, the extension points would be added after each *update* to a variable, to en- sure that the update is considered for logging.

Extension points are added at compile time depending on what direction commands we want to support at run- time. For example, starting with the sequence of statements

*. . .* ; *si*; *si*+1; *. . .* we insert two extension points @*L* and

@*M* , resulting in the program *. . .* ; *si*; @*L*; *si*+1; @*M* ; *. . .*.

When extension point *L* is reached at runtime, a stored pro- cedure associated with *L* is executed by the controller. This procedure ultimately hands back control to the host pro- gram, or starts an interactive session with a *program direc- tor*, which can send the controller further commands, and update its stored procedures.

Our work systematises *ad hoc* debugging and profiling extensions to programs, and generalises the facilities cur- rently made available for hardware development. Moreover, we can include only the features needed, thus improving the utilisation of the hardware, and its power consumption. The approach is extensible: one can code additional direc- tion commands, or variations [(§3.3).](#_bookmark10) In this paper we show how to give semantics to program direction commands in terms of the placement of extension points, the code that is to be run by the controller, and the interaction between the controller and the direction tool that manages it.

We prototype PhD using an HLS and obtain a uniform interface for directing the software and hardware instances of the same program, allowing us to unify the debugging of these instances (which otherwise require diverse tools).

We believe that PhD can yield practical benefits in hard- ware development and deployment. As a technique, PhD is vendor-neutral and compiler-neutral, and the communica-

***Hosted directability***

Controller

Program

(Normal interaction with external world)

***Original program behaviour***

Director

**Figure 2.** A *controller* is embedded into the program, and acts as the agent of the *director*. The director and controller implement a protocol to exchange commands and their out- puts.

tion between the director and controller can be adapted to suit the program. For example, in our prototype we send di- rection commands via the network. PhD can be implemented in different ways: in our prototype we implemented it en- tirely in a high-level language via HLS, but one could also bolt it onto more finely engineered hardware blocks. Despite running as hardware, the directability features can be en- abled and reconfigured at runtime: this consists of updating the code stored by the controller. In contrast, existing tech- niques for FPGAs involve including *fixed function* circuits in the design. Finally, PhD extends a program with a *direction mode*, thus facilitating the in-field debugging of programs.

Through PhD we hope to contribute to the convergeance between the debuggability of programs on FPGAs, and those on CPUs, which enjoy extensive hardware and OS support, and which in turn benefits sophisticated monitoring systems such as DTrace [(Gregg and Mauro](#_bookmark49) [2011).](#_bookmark49) The ideas de- scribed in this paper are not necessarily tied to the languages, compilers, FPGA or other equipment we used in our proto- type. We make the following contributions:

* + We describe how to translate familiar high-level idioms for debugging, profiling and monitoring [(§3.1),](#_bookmark4) which we call *direction commands*, into a low-level language for controlling program state at runtime [(§3.2),](#_bookmark8) for an example language [§3.4.1.](#_bookmark12)
  + We relate the direction commands with our low-level lan- guage through a specification [(§3.4)](#_bookmark11) in which programs are ordered by the directability features they support. This provides a basis on which we can reason that one program is “more debuggable” than another at runtime.
  + A prototype implementation and its evaluation [(§4),](#_bookmark17) where we measure the effect of directability on FPGA utilisation and performance.

# FPGA debugging gap

Irrespective of how we write our program as a hardware description, once a so-called “bitstream” is generated that configures an FPGA to run our program, the program can only be tested as a black box, and we cannot understand or further influence its behaviour at that stage.

The need for better FPGA debugging features is becom- ing more urgent since:

* + FPGA chips are getting larger, which allows them to run more complex programs. Complex programs are more likely to be buggy, which necessitates more debugging.
  + FPGAs are also used for simulating hardware designs, since behavioural simulations are very slow. [(Wang et al. 2011;](#_bookmark86) [Chu et al. 2015)](#_bookmark34)
  + FPGAs are being deployed in large production environ- ments, such as Microsoft’s datacentres [(Putnam 2014).](#_bookmark70)

Hardware development has driven the development of formal methods to establish system correctness [(Fix 2008),](#_bookmark42) which enabled the development for methods for software [(Ball](#_bookmark29) [et al. 2006;](#_bookmark29) [Godefroid et al. 2012).](#_bookmark45) Unfortunately the ver- ification is done on the Register-Transfer Level (RTL), a higher-level description of a hardware circuit in languages such as Verilog, and not on the generated bitstream. Thus debugging may still be needed.

### Debugging(FPGA) != Debugging(CPU)

Debugging concepts from software do not correspond di- rectly with debugging the same program on an FPGA. A lot of the core issues involved were discussed in the pioneering work of [Koch et al. (1998)](#_bookmark57) and in many other work on HLS debugging [(Goeders and Wilton 2014;](#_bookmark46) [Monson and Hutch-](#_bookmark64) [ings 2015).](#_bookmark64) These are the main points:

* + - Multiple source lines might be executed concurrently in hardware. Code is represented at the source, register- transfer, and gate levels. This has important conse- quences for debugging, described in the following points. The correspondence between these levels necessitates keeping metadata from compilation for debugging.
    - Depending on the debugged artefact, *stepping* by (source) line might be less useful than stepping by cycle.
    - *Breakpoints* become more tricky to interpret, since they are usually set on a specific line of code. In hardware there may be several overlapping lines being executed at a breakpoint [(Koch et al. 1998,](#_bookmark57) §4.3). Moreover, the out- put of operations on previous lines might only be avail- able *after* some clock cycles have elapsed. Depending on what the user wishes to do, they might prefer if the break- point is triggered after the elapsing of these cycles. Fur- thermore, part of the next line of source code might have started executing. This suggests that a strict indication of sequentiality needs to be communicated to the HLS com- piler if the usual breakpoint semantics are desired.
    - As mentioned in the introduction, FPGAs do not provide hardware support such as debug registers to assist with analysing running programs.

### Current techniques for FPGA debugging

Some existing techniques help narrow the debugging gap on FPGAs. *Co-simulation* involves comparing the behavioural simulation between HLS and RTL. This can be considered

a special case of *relative debugging* [(Sosicˇ and Abramson](#_bookmark80) [1997)](#_bookmark80) but it does not provide visibility into the hardware instance of a program. Another technique involves *in-system* testing: testing a large part of the system, though possibly not all. This does not provide visibility into the hardware either. Current practice employs two techniques for FPGA de- bugging. **Trace buffers** are the most popular technique for debugging FPGAs. It requires a programmer to identify sig- nals of interest in the circuit at compile time, then an *em- bedded logic analyser* is synthesised that uses on-chip mem- ory to record traces for these signals. This suffers from two problems: only a limited number of signals may be viewed (limited by on-chip memory), and traces have a limited win- dow size (for the same reason). Traces may be conditional, to avoid using up buffer space unnecessarily, but this technique is difficult to use because it involves generating the bitstream each time. **Register scanning** allows you to see the values of all registers on the FPGA, but requires “stopping the world” to enable reading and sending it off-chip. This slows down tests, and thus register scanning has been supplanted by trace

buffers.

Both register scanning and trace buffers usually send recorded data off-chip via the JTAG (Joint Test Action Group) interface, a standardised instrumentation method [(Ben-](#_bookmark31) [netts and Osseyran 1991).](#_bookmark31) This method is not scalable, since its transfer rate is far less than the FPGA throughput.

In summary, existing techniques consist of including “fixed function” modules as part of your hardware de- scription at compile time. This has the advantage of being lightweight since these circuits are specialised to perform a single function, but it has the disadvantage of being inflex- ible. Generating a hardware bitstream can take hours, and the added overhead costs for runtime-reconfigurable debug- gability and monitoring features might not be affordable in some use-cases. Furthermore, these techniques cannot be used in production environments.

### High-Level Synthesis (HLS)

HLS involves the use of a high-level language, such as Java, C, C++, and OCaml, to write a hardware description. This takes advantage of the features of, and tooling available for, the high-level language. An RTL description is then generated from the high-level description.

Using an HLS to describe hardware enables one to run the HLS description as a software program, and to debug it as such, by using standard tools to compile and debug Java programs for instance. This prunes bugs from the eventual bitstream and avoids regenerating the bitstream. Irrespective of whether testing is directed at software or hardware, it can take many tests to find a fault. Software-based testing could help detect logic errors in our code, but it could not help us find some important classes of problems: ‘[Testing in] the silicon mode permits the analysis of bugs that are “invisible” at the RTL level’ [(Calagar et al. 2014).](#_bookmark32) We outline the main cases below:

1. *Interface mismatch.* We need to understand whether a problem occurs because of a mismatch between one mod- ule and the rest of the circuit. Recall that behavioural sim- ulation might not be applied to the whole design, and in- correct assumptions about the enclosing circuit can result in the simulation test succeeding but the hardware tests failing.
2. *Reproducability.* Some faults are triggered during high- throughput tests, and are difficult to find when test- ing other instances of the program. Other faults result from features of the hardware and transient environmen- tal states – such as “Single Event Effects” manifested through the flipped or stuck bits from the interaction of charged particles with semiconductors [(Sari et al. 2014;](#_bookmark74) [Krishnaswamy et al. 2008).](#_bookmark58)
3. *Toolchain problems.* Diagnosing bugs in the compiler toolchain becomes easier if we can see into the compiled program’s operation. Bugs are not unusual in both HLS and RTL toolchains.

### Current research on FPGA debugging

Various improvements have been explored for the techniques described above. For example, one could write summaries to the trace-buffer, rather than the explicit trace [(Goeders](#_bookmark46) [and Wilton 2014).](#_bookmark46) Another idea consists of multiplexing all signals and choosing which to observe at debug time rather than at compile time [(Hung and Wilton](#_bookmark53) [2013),](#_bookmark53) enabling “observation without recompilation”. An additional idea is to buffer into a fast external memory (P[anjkov et al. 2015).](#_bookmark67)

Table [1](#_bookmark6) surveys closely related work, classifying it ac- cording to the features they provide. Related work is con- trasted in more detail in [§5.](#_bookmark23) In Table [1](#_bookmark6) *extension points* means being able to extend the program at certain points at runtime. *Interruption* refers to asynchronous interruption of a program by a debugger. *Fine granularity* means that we can look at arbitrary parts of a program; for instance [McKechnie et al. (2009)](#_bookmark60) only allow inspecting at the module boundary. *Software instance* means that the program can be run on the CPU as a process, independent from the FPGA. *Network/Control* refers to how a technique is implemented: as a control loop or as a network that feeds signals to a logic analyser. *Use leftover resource* means that the debug circuitry does not compete with the program’s circuit. And *embed at Source/HDL* refers to whether the source code or its HDL image is updated to include the debug circuitry.

### PhD design features

PhD came about after we found ourselves extending our *ad hoc* debugging and monitoring code to support additional features, instigating us to study the problem more rigorously.

Table [1](#_bookmark6) shows the features supported by PhD. Initially it supported state inspection and update. Updating state en-

abled us to support more features, such as assertion check- ing, which involves breaking if a condition is satisfied.

As an idea, PhD is neither committed to HLS nor HDL description. In our prototype we implemented it in HLS for convenience, but often better performance can be gained by relying partly or fully on modules written in HDL.

# A language and model for program directing

In this section we describe D, a language of direction com- mands that we extracted by analysing the commands com- monly given to profilers and debuggers such as gdb.[1](#_bookmark5) In Fig- ure [1](#_bookmark0) we described one such command and how we code

the high-level direction command as a program that will be executed by the controller.

### Direction language D

Our direction language D consists of the commands listed in Table [2.](#_bookmark7) These commands can have three kinds of parame-

ters: (i) symbols relating to the program, (such as variable or function names, or labels), (ii) relations over program vari- ables, which we denote by the symbol *B* , indicating the conditions on which statements might apply, and (iii) mea- sures of resource, which we symbolise by $ , indicating a finite resource that is allocated for the execution of a com- mand. Symbol *X* is a metavariable ranging over variable identifiers in the source program, and *L* is a metavariable ranging over *labels* in the source program. A label is asso- ciated with a single position in the program, (e.g., line 5 in function “main”), but a single position might be associated with multiple labels. below.

*( )*

*( )*

Let D *B* be the set of possible conditions that can be used. We assume that at least true D *B* . We can also allow additional truth conditions, and in this model we will have (*V*1 = *V*2) D *B* for arbitrary *V*1*, V*2 ranging over program values or variables. For example, “watch *v* (*v* =

*∈ ( )*

*( )*

*∈ ( )*

5)” would instruct the controller to watch a variable *v*, and switch to interactive directing when *v* = 5.

D $ N describes the maximum quantity of some re- source when carrying out a direction command. This value

*( ) ∈*

must be less than the compile-time allocation of the re- source, to ensure the provision of sufficient resource for the command at runtime. This is needed to size the buffers used for tracing. For example, “count reads *v* true 5000” will count the number of reads of *v*, and break after 5000 reads have been made. This could be done to avoid over- flow, or to capture some behaviour of interest. Similarly, “trace start *v* true 500” breaks after 500 instances of *v* have been recorded in the trace. The trace buffer must ac- commodate at least 500 entries.

abled us to influence data-dependant control flow. Extending

the controller to include branching and extension points en-

1 [https://www](http://www.gnu.org/software/gdb/).gnu.or[g/software/gdb/](http://www.gnu.org/software/gdb/)

### System

[(Sosicˇ 1992)](#_bookmark81) **Dynascope**

C C

C C C C C C C C C

**Features** state inspection trace recording state updating extension points break points stepping interruption

fine granularity assertion checking hang detection timing checks software instance

runtime reconfigurable

HLS (vs HDL)

**N**etwork/**C**ontrol use leftover resource

embed at **S**ource/**H**DL

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [(Goeders and Wilton 2014)](#_bookmark46) **HLS-Scope**  [(Calagar et al. 2014)](#_bookmark32) **Inspect**  (P[anjkov et al. 2015)](#_bookmark67) [(Hung and Wilton 2014)](#_bookmark54) **QuickTrace** [(Koch et al. 1998)](#_bookmark57) **SLE/CADDY** [(Monson and Hutchings 2015)](#_bookmark64)  [(Curreri et al. 2011)](#_bookmark36) [(Camera et al. 2005)](#_bookmark33) **BORPH** | C C C C C C  C | C  C C | C | C C  C C | C C  C C | C | C C  C C  C | C | C | C | C  C C | C | C C C C C C C | C C C N C C  C | C | S H S H H S  H |
| C |  | H |

(see [§2.5)](#_bookmark2) **PhD** C C C C C C C C C C C

**Table 1.** Survey of features provided by debugging systems. Blacked-out boxes mean “not applicable”.

### Command Behaviour

the counters, get their current value, and find out if a maximum value has been reached.

print *X*

unbreak *L*

break *L (B)*

backtrace *(*$*)*

watch *X (B)*

unwatch *X*

count writes *X (B) (*$*)*

Print the value of variable *X* from the source program. Activate a (conditional) breakpoint at the position of label *L*. Deactivate a breakpoint.

Print the “function call stack”.

Break when *X* is updated and satisfies a given condition. Cancel the effect of the “watch” command.



reads *X (B) (*$*)*



calls *fname (B) (*$*)*





Count the reads or writes to a variable *X*, or the calls to a function *fname*.

trace clear *X*

stop *X*

start *X (B) (*$*)*

Trace a variable, subject to a condition being satisfied, and up to trace some length. Stop tracing a variable.

Clear a variable’s trace buffer.

Print the contents of a variable’s trace buffer. Check if a variable’s trace buffer is full.

**Table 2.** Directing commands making up language D. Note that count has similar subcommands to those of trace, to clear



print *X*

full *X*

### Controller

High-level direction commands such as those in Table [2](#_bookmark7) are ultimately translated into programs that run on a simple con- troller embedded in the program. We model the controller as a *CASP* machine (for “Counters, Arrays, and Stored Pro- cedures”, the constituents of the machine’s memory). CASP machines are very weak. They are more structurally complex than register machines [(Shepherdson and Sturgis 1963)](#_bookmark76) since they have separate memories for storing arrays and regis- ters, but CASP machines are computationally much weaker than register machines, unable to encode partial computable functions. A limited form of memory indirection is permit- ted through a collection of arrays. The language lacks any

means for defining recursive functions, or branching to ar- bitrary addresses. Any more complex computation must be done by the director; the controller simply provides a con- trolled access to the program’s memory.

We describe the language of CASP programs in Figure [3.](#_bookmark9) We rely on the following meta-variables and syntax cate- gories: *P* programs, *E* expressions, *I* indices, *U* updatable values, *V* values, *N* numerals (corresponding to Z), *X* vari- able identifiers, and *R* the array identifiers, where the names for variables and those for arrays are disjoint, *X R* = .

*∩ ∅*

@*L* : *P* is a *placement* command: it updates the code at extension point having label *L* to be *P* . Note that placement commands may not be nested in our model: for instance, this is not a valid program:

*{ }*

*P* ::= *E*

*| U* := *E*

*| op U*

*op ∈ {*inc*,* dec*}*

*| P*1; *P*2

*|* if *E* then *P*1

else *P*2

*|* break

*|* continue

*|* @*L* : *{P j}*

*E* ::= *V*

*| −V*

*|*

*V*1 *op V*2

*op* =*, <*

*∈ { }*

*I* ::= *N X*

*|*

1. ::= *X*

*R*[*I*]

*|*

1. ::= *I*

*| R*[*I*]

***Profiling.*** The command “count writes *v*” causes a counter to be incremented each time a variable is updated:

**if** V\_ count\_ writes > V\_ count\_ writes\_ max **then break**

**else**

**inc** V\_ count\_ writes ;

**continue**

***Watchpoints.*** The command “watch *v B* ” causes the program to break (for interactive guidance) when a vari- able’s value satisfies some predicate. Let *B* be the CASP- level value to the D-level *B* parameter (we give such a function in [§3.5.1).](#_bookmark14) The code is similar to that in the profil-

*( )*

*( )*

**Figure 3.** Syntax for CASP programs.

@*L* : *{*if *x* = 1 then @*M* : *{*break*}*

else @*M* : *{*continue*}}*

We do not want to allow programs to be self-modifying in this way, since it complicates reasoning about them.

***Ending programs.*** Both continue and break indicate the end of a CASP program, but differ in what happens before resuming the host program (in which the controller is embedded). continue simply resumes where the host program left off, whereas break switches into an interactive direction mode. In this mode, the controller may receive commands from the director, execute them, and send an acknowledgement back.

The remaining syntax forms and constants used above are standard and intuitive. Owing to its simplicity the semantics of this language are straightforward, and are given in [§C.1.](#_bookmark94)

### Examples

CASP programs are to D what microprograms are to an In- struction Set Architecture [(Smith and van Leeuwen 1974).](#_bookmark78) We give some examples of coding program direction com-

mands below, before describing the behaviour of program direction commands in more detail in the next two sections.

***Conditional tracing.*** Let BASIC represent the program that codes the behaviour of “trace *V...*” from Figure [1.](#_bookmark0) We can code the *conditional* variant of this command, for example when *V* is less than some threshold value V\_trace\_threshold (and where *>*= is syntactic sugaring):

**if** V >= V\_ trace\_ threshold **then continue**

**else**

BASIC

***Sampled tracing.*** The *sampled* variant involves allocating an additional variable to count the interval between samples, and storing the desired sample interval:

**if** V\_ trace\_ samp = 0 **then**

V\_ trace\_ samp := V\_ trace\_ samp\_ interval ; BASIC

**else**

**dec** V\_ trace\_ samp ;

**continue**

ing example above, except that *B* is user-provided.

**if** *B* **then break else continue**

***Breakpoints.*** “break *L B* ” causes the program to break when it reaches a specific label, and if condition *B* is sat- isfied. The coding is identical to that in the watchpoint example, but they differ in their *placement*: breakpoints are placed at programmer-specified positions in the code whereas watchpoints are associated with labels where vari- ables are updated. This difference cannot be seen from the snippet, but will become evident in the formalisation of the program direction commands, which we start next.

*( )*

### Directability ordering

In this section we define a relation *x* m *xj* to mean “*xj* is more directable than *x*”, where *x, xj* are triples ( *, , p*) and ( *j, j, pj*), each representing three interdependent par- ties: the *director*, *controller* and *program*. The user (or their

*D C*

*D C*

agent) issues direction commands to the director, which in- teracts with the program’s state via its agent, the controller, embedded in the program. We use this relation to give se- mantics to the direction commands in terms of interaction with CASP machines.

Our directability relation gathers information about the three parties involved, and describes how their interdepen- dence is revealed by the directing commands: for example, the director would not be able to execute trace *X* if the pro- gram did not have a variable called *X*, or if the controller had not been allocated a trace buffer.

In our notation, represents the director’s state (a set of facts representing its knowledge about the controller’s state, such as which breakpoints exist, and whether they are active or not). is the controller’s state, consisting of a (*C, A, SP* ) machine, cf [§3.2.](#_bookmark8) *p* is a program. We will define an example language in [§3.4.1](#_bookmark12) to aid our formalisation.

*D*

*C*

We also include in the relation some information about

*why* one triple is less directable than the other. We therefore index the relation by (i) C D the direction commands [(§3.1)](#_bookmark4) supported by ( *, , p*), (ii) *c* D the additional command supported by ( *j, j, pj*), and *Dc* ( *j j*)

*D C ∈ D → D*

*D C ∈*

*⊆*

the semantics of this command. Note that C denotes the set

of direction commands that is supported *simultaneously* by

(*D, C, p*), i.e., these commands are allocated separate state.

Written out in full, we obtain this relation:

(*D, C, p*) mC (*Dj, Cj, pj*) : *Dc*

*c*

In [§3.5](#_bookmark13) we will instantiate such a relation by formalising commands from D in terms of CASP machines. Note that this describes how the directing commands are translated into CASP programs, but we do not fully formalise the director: *Dc* is written in an ML-like pseudocode.

Our formalisation is devised in away that avoids the mu- tual interference of direction commands. That is, the same program can be subjected to any combination of direction commands. To make this non-interference more precise, we

introduce some definitions. Let ˘ = *j* and ˘ = *j* . We say that *Dc* is *relevant* to *j* if it only manipu-

*D \D*

*D D \D C C \C*

lates state or elements introduced in *j* and *j*. Furthermore commands are *disjoint* if they introduce non-overlapping state. That is, for any two commands *c*1 and *c*2, for any

*D C*

prior states *C* and *D* their respective new states are disjoint:

(*D*1*\D*) *∩* (*D*2*\D*) = *∅* = (*C*1*\C*) *∩* (*C*2*\C*).

### Program language

In this section we specify a first-order imperative language to support our formalisation of program direction commands. Unlike CASP machines [(§3.2)](#_bookmark8) this language is computa- tionally strong: recursive functions over the integers can be encoded. The language’s simplicity enables the relation of program direction with CASP machines, while avoiding ex- cessive formal complexity. Formalising transformations for realistic languages—even simple transformations [(Schäfer](#_bookmark75) [et al.](#_bookmark75) [2008)—is](#_bookmark75) usually fraught with complex definitions, and we avoid that here.

The language grammar is given next. Note that for sim- plicity we deliberately overlay the meta-variables for vari- ables and numerals (*X* and *N* ) over those for CASP ma- chines. This simplifies the interface with CASP programs, which will be executed at extension points within host pro- grams. *p* ranges over programs, *s* over statements, *e* over expressions, *τ* over types, and *vdecl* and *fdecl* over the dec- laration of variables and functions respectively.

*p* ::= *vdecl fdecl* return *fname*(*e*¯)

*vdecl* ::= *τ X*

*fdecl* ::= *τ fname*(*x*¯) *s*; return *e s* ::= skip

*{ }*

*| X* := *e*

*|* if *e* then *s*; *s*

*| { }*

extend *L*1*, . . . , Ln*

*e* ::= *N*

*| X*

*| fname*(*e*¯)

*| e*1 *op e*2

*op ∈ {*+*, −,* ==*, <}*

In this language the only data type in *τ* is the inte- ger type. The only unusual construct in this language is

extend *L*1*, . . . , Ln* . This indicates an *extension point*, where control is passed to the CASP machine [(§3.2).](#_bookmark8) As before, *L* is a metavariable ranging over *labels* drawn from a denumerable set.

The semantics of this language are straightforward, and are given in [§C.2.](#_bookmark93) Intuitively, the behaviour of extend *L*1*,*

*{ }*

*{*

*. . . , Ln* is as follows. If *n* = 0 then the command has no effect. Otherwise, the stored procedure associated with each *Li* is called, in any order, and run to completion, noting the last instruction of each *Li*. The last instruction of any CASP program is either ‘continue’ or ‘break’ [(§3.2).](#_bookmark8) If all *Li* end in ‘continue’, then the behaviour of extend *L*1*, . . . , Ln* is to continue executing the next statement in the host pro- gram. Otherwise, if at least one *Li* ends in ‘break’, then the controller switches into interactive mode. In this mode, con- trol remains with the controller, until the director sends it a ‘continue’ command, at which point control is returned to the program.

*}*

*{ }*

We make the simplifying assumption that all the state- ments in the user’s program are interspersed with ‘extend’: that is, if the user writes *s*0; *. . .* ; *sn* then this is translated into extend ; *s*0; extend ; *. . .* ; extend ; *sn*; extend . In the next section we populate these extension points with labels, to extend the directability of a program.

*{} {} {} {}*

### Semantics for D

In this section we encode program direction commands [(§3.1)](#_bookmark4) into interactions between the director and controller [(§3.2).](#_bookmark8) Note that this describes how the directing commands are translated into CASP programs, but we do not fully for- malise the director: *Dc* is written in an ML-like pseudocode.

### Break

We start by formalising the meaning of the “break” com- mand. Intuitively, this command adds a breakpoint to a pro- gram: an extension point is at the position of the breakpoint is labelled with *L*, and the associated state is set up in the director and controller. To support this command:

* + - * A program *p* is extended to include a label *L* at the posi- tion where the breakpoint is to be placed. This extension is formalised by the premise *p <*1 *pj*, which means that *p*

*L*

is identical to *pj* except for the label *L* occurring at some

extension point. This is defined formally in [§B](#_bookmark92) along with

related definitions, such as that of a *position* in the pro- gram.

* + - * The label *L* must not appear in the original program. We write this as *L ƒ∈ p* using an abbreviation defined in [§B.](#_bookmark92)
      * The controller’s state is extended to store the procedure associated with *L*. Furthermore, the breakpoint is acti- vated by default. We use the abbreviated notation ˘ = *SP* [*L* break] to indicate this extension, where *SP* is the stored-procedure memory in the CASP.

*{ ›→ }*

*C*

* + - * The director’s *state* is extended to encode whether the breakpoint is currently active or not. It is activated by

where

default, thus: *D*˘ = *{*(«bp»*, L,* 1)*}*, where “«bp»” is a

¢ ¢

unique token we use for breakpoints, and 1 is a token we use to indicate that the breakpoint is active. Soon we will formalise the “unbreak” command, which flips this value to 0.

* + - * The director’s *behaviour* for the “break” command, *Dc*, involves activating the breakpoint unless already active.

In the definition of *Dc* below we use the notation *P* ~ *N* , which we use to mean that the director sent the CASP program *P* to the controller, and received the reply *N* . Thus “@*L* : break ~ ’*L*’” means that the director instructed the controller to store the program break at *L*, and that it expects to get ’*L*’ back (which is a code indicating where

*{ }*

the program is stored, as formalised in [§C.1).](#_bookmark94)

In *Dc* we use the notation “(«bp»*, L,* 0 1) : *j*” to abbreviate («bp»*, L,* 1) ( *j* («bp»*, L,* 0) ).

*{ } ∪ D \{ }*

*›→ ∈ D*

The formalisation of the “break” command follows. We

can use a simplified notation since our commands will be

both relevant to *Dj\D* and disjoint:

*L*

unbreak *L* = @*L* : *{*continue*}*

Note that the program, and the states of the controller and director are not changed. The only extension is made to the behaviour of the director, which is extended with a function to unset the breakpoint. This time we didn’t set anything in the controller since there is no default behaviour and additional state required for unsetting a breakpoint. This is because unbreaking a breakpoint doesn’t establish the breakpoint, whereas breakpoint does.

To print the value of a variable *X*, that variable needs to exist in the program (*X* Var*p*), and we need to have at least one extension point (through which we can send the print command).

*∈*

(break *L (B)*) *∈* C *X ∈* Var*p*



**** *D* = *{}*

print *X*

*p* C

*p*  *C*˘ = *{}*

˘

m

*j*

*L ƒ∈ p p <*1 *pj*

**** *Dc* = *λD . X* ~ *N* ;

  print(*N* )

**** ˘ ¢ ¢

*p* break *L (B) pj*

mC

****

*D ∈ D D*

where

*D*˘ = *{*(«bp»*, L,* 1)*}*

= *SP* [*L* break *L B SP* ]

*C { ›→ ( ) }*

*Dc* = *λ j.* if («bp»*, L,* 1) *j* then *j*

else

break *L (B)* ~ ’*L*’; («bp»*, L,* 0 *›→* 1) :*∈ Dj*

¢ ¢

*∈*

This time we didn’t use a placement command to update the behaviour of the controller; we simply ran a query. print(*N* ) is pseudocode that uses a print function in the director. Re- call that we formalise directions in terms of CASP machines, and don’t formalise the director’s behaviour. *X* Var*p* en- sures that *X* is a variable in *p*. Var*p* is defined in [§B.](#_bookmark92)

break *L (B) SP* = conditional *(B)* break

¢ ¢

¢break *L (B)*¢ = @*L* : *{*¢break *L (B)*¢*SP }*

We use break *L (B)*¢ to denote the meaning of

¢

“break *L B* ” to the director, as a CASP program.

is the value used to initialise the stored

*( )*

### Trace

The most important command related to tracing is “trace

start”; the other commands depend on it.

break *L B SP*

¢ *( )*¢

program associated with *L*.

The meaning of *B* is translated for inclusion in the CASP program by the following function:

*( )*

conditional *(B) t* =



if *(B)* = true

if *(B)* = (*I*1 = *I*2)

*t*

if *I*1 == *I*2 then *t*

*X ∈* Var*p ∀L ∈ XL. L ƒ∈ p*

Positions*pt* (*XL*) = PostUpdate*pt* (*X*) *p <XL pj*

*p* trace start *X (B) (*$*) pj*

mC

 else continue 

We now turn to the “unbreak” direction. To be able to issue this direction, the breakpoint needs to exist—thus we have a dependency on the “break” direction earlier:

****  

**** *C*˘ =  *SP* [*L ›→* ¢trace start *X (B) (*$*)*¢*SP* ]

(break *L (B)*) *∈* C

****

*D*˘ = *{}*

**** *C*˘ = *{}*

m

*D ∈ D D*

*D*˘ = *{*(«t»*, X,* 1)*}*

*C* [*X*i] = 0*, C* [*X*of ] = 0*, A*[*X*a[*(*$*)*]]*,* 

****  for each *L XL* 

*Dc* = *λ j.* if («t»*, X,* 1) *j* then *j*

*D ∈ D D*

*∈*

****

else

for each *L ∈ XL*:

*{*¢ *( ) ( )*¢ *}*

****

unbreak *L*

¢ ¢

*p* C *p*

****

*Dc* = *λ j.* if («bp»*, L,* 0) *j* then *j*

else unbreak *L* ~ ’*L*’;

(«bp»*, L,* 1 *›→* 0) :*∈ Dj*

****

@*L* : trace start *X B* $ *SP*

~ ’*L*’;

(«t»*, X,* 0 *›→* 1) :*∈ Dj*

trace start *X (B) (*$*) SP* =



Controller

Program

Program

¢ ¢

conditional *(B)*



if *X <* $ then 

i *( )*

*X*a[*X*i] := *X*;

 inc *X*i; 

continue

else





inc *X*of ;

break

Applying this rule depends on a set of labels *XL* in *pj* that don’t exist in *p*. *pj* is the least extension of *p* that includes these labels. Furthermore, these labels coincide with the

positions in the program occurring after *X* has been updated. The controller’s state is extended with the buffer index, *X*i, initialised to 0; the overflow indicator *X*of , initialised to 0 (for false); and *X*a is an array that can hold $ elements.

*( )*

Each *L XL* labels the positions immediately after the variable has been updated.

*∈*

(trace start *X (B) (*$*)*) *∈* C

Positions*p*(*XL*) = PostUpdate*p*(*X*)

trace stop *X*

**Figure 4.** Transformation of the program to include a con- troller. Normal packets are handled as normal, but *direction packets* are passed to the controller. Pink dots represent *ex- tension points*, one of which is added within the control flow of the original program in this illustration.

***Focus.*** From its description, the PhD idea is not con- strained to a specific kind of program. In our prototype we focussed on using it to work with *network* programs how- ever, for two reasons:

1. It allows us to test remote directing over standard net- work equipment. In our survey of tools and techniques (Table [1)](#_bookmark6) only Dynascope has network access, but it does not work for FPGAs.
2. It allows us to use industrial high-precision network mea-

*D*˘ = *{}*

****

*C*˘ = *{}*

****

 *Dc* = *λDj.* if («t»*, X,* 0) *∈ Dj* then *Dj*

*p* mC *p*

suring equipment to see the effects on the program host- ing a controller.

***Use-cases.*** As programs we used implementations of DNS and Memcached that we had written previously to run on FP-

GAs, as part of earlier research. DNS (Domain Name Sys- tem) is a ubiquitous name-resolution system used on pri-

else

for each *L ∈ XL*:

¢ ¢

@*L* : *{* trace stop *X SP }* ~ ’*L*’; («t»*, X,* 1 *›→* 0) :*∈ Dj*

****

vate and public packet-switched networks such as the In- ternet [(Mockapetris 1987).](#_bookmark63) This implementation was around 700 lines of C#. Memcached [(Fitzpatrick 2004)](#_bookmark41) is a well-

where

¢trace stop *X*

¢*SP*

= continue.

known, in-memory key/value store that caches read results in memory to quickly respond to queries. The protocol uses a

The remaining commands are formalised in [§A,](#_bookmark91) sup-

ported by program-level predicates and functions defined in [§B.](#_bookmark92)

# Implementation and Evaluation

We prototyped the ideas described in the previous section by extending network programs with program-hosted di- rectability, and compiled them to run on an FPGA. We then evaluated the effect of these directability features on the pro- gram in which they are embedded. Our approach enables us to make fine-grained modifications to directability, and we evaluate the overhead from supporting different CASP ma- chine instructions.

### Prototype

In our prototype we manually transformed programs to in- clude the controller. This transformation was straightfor- ward: we wrote the controller (implementing a CASP ma- chine) and added extension points to the program (consisting of calls to the controller to execute a stored procedure).

number of basic commands such as GET (retrieve a value as- sociated with the provided key), SET (store a key/value pair) and DELETE (remove a key/value pair) and has both ASCII and binary protocols. In this work as proof-of-concept we have implemented a limited version of Memcached support- ing GET/SET/DELETE using the binary protocol over UDP and supporting 6 byte keys and 8 byte values. This imple- mentation was almost 900 lines of C#.

***Method.*** We transformed the DNS and Memcached imple- mentations in two ways: (i) adding code to check whether a received packet is a *direction packet* intended for the con- troller (see Figure [4),](#_bookmark16) in which case the controller (and not the original program) processes the packet; (ii) adding an ex- tension point in the body of the (DNS or Memcached) main loop, allowing us to influence and observe the program from that point. We form an enumerated type that corresponds to the program variables whose values the controller may ac- cess and change at runtime. The code for each value of the enumerated type is used to refer to the program value, to

this module, the packets are passed to the physical interfaces and are transmitted.



Output Queues

Input Arbiter

Main Logical Core

|  |
| --- |
| 10G Port |
| 10G Port |
| 10G Port |
| 10G Port |
| PCIe & DMA |

|  |
| --- |
| 10G Port |
| 10G Port |
| 10G Port |
| 10G Port |
| PCIe & DMA |

 

**Figure 5.** NetFPGA Reference Pipeline. Each of the imple- mentations in our prototype consists of a separate Main Log- ical Core.

instruct the controller to increment it, for example. Building tool support to automate parts of this process seems feasible.

***Direction packets.*** Direction packets are network packets in a custom and simple packet format, whose payload con- sists of (i) code to be executed by the controller, or (ii) sta- tus replies from the controller to the director. It enables us to remotely direct a running program, similar to gdb’s ‘re- mote serial protocol’.[2](#_bookmark19) Our design uses a simple direction language, and works for instances of a program that run both as software and hardware (on FPGAs), whereas gdb requires special backends for each architecture.

### Evaluation

We evaluate our prototype by carrying out a quantitative analysis of the impact that the controller has on the program in which it is embedded. This impact is measured in terms of *utilisation* of resources on the FPGA, and the *performance* of the host program.

An FPGA consists of an interconnected grid of *logic blocks*, which in turn contain resources such as memory and logic functions (in the form of so-called *look-up tables*). Flip-flops are primitive storage circuits.

Table [3](#_bookmark21) shows the utilisation and performance for DNS and Memcached, extended with different controller features. These features show a fine-grained decomposition of the instructions supported by the controller: reading a variable, writing a variable, and incrementing a variable. We see that the impact on utilisation and performance is minimal.

Performance is analysed in three ways. The *duration* is the number of clock cycles that are needed by the program

***Controller.*** Our controller follows the description of CASP machines very closely [(§3.2).](#_bookmark8) It has two features. First, memory is organised into *Counters*, *Arrays* and *Stored pro- cedures*. Counters include variables in the original program, as well as extra registers used for program directing. In this prototype we only support numeric datatypes; structured datatypes could be encoded in principle. Secondly, a func- tion that interprets the language of CASP machines. This is used to branch to stored procedures when their correspond- ing extension points are reached.

***Tools and equipment.*** We wrote our programs in C#, and used the Kiwi *high-level synthesis* (HLS) system [(Singh and](#_bookmark77) [Greaves 2008)](#_bookmark77) that statically recompiles .NET bytecode into Verilog. The Verilog code generated by Kiwi was slotted into open-source reference code from the NetFPGA project,[3](#_bookmark20) and compiled to run on the NetFPGA SUME board [(Zilber-](#_bookmark89) [man et al.](#_bookmark89) [2014),](#_bookmark89) a low-cost, PCIe host adapter card able to support 40Gb/s and 100Gb/s applications. At the core of the board is a Xilinx Virtex-7 690T FPGA device. The NetFPGA reference designs share the same FPGA architec- ture, illustrated in Figure [5,](#_bookmark18) of multiple physical interfaces surrounding a logical data-path. Traffic to the FPGA enters through one of four 10Gb/s ports or from a PCIe interface, and is passed into the main data-path by an input arbiter, which arbitrates between all ingress interfaces. Packets are then processed within the main logical core, and are diverted to their destination through an output queues module. From

2 [http://www.embecosm.com/appnotes/ean4/](http://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.html)

[embecosm-howto-rsp-server-ean4-issue-2.html](http://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.html)

3 <http://netfpga.org/>

to process a packet within the main logical core, as extracted from simulation. In our hardware, each clock cycle takes 10ns. *Latency* is the time taken for a program to service requests from the network. *Throughput* is the rate of requests that can be serviced by the program (before packets start to be dropped).

Using the controller we can read and change the pro- gram’s state at the packet rate. Using JTAG and the Virtex-7 FPGA we can read data at up to 66Mbps, three orders of magnitude less than the maximum throughput that the NetF- PGA can sustain over high speed serial interface –e.g. the PCIe channel. In principle the direction controller could use any slice of that, subject to not interfering with the hosting program too much.

In Table [4](#_bookmark22) we compare DNS extended with one extension point, against DNS extended with an *embedded logic anal- yser* (ELA). An ELA is a standard technique in hardware development, and consists of a circuit that passively mon- itors the program, creating a trace of the program’s execu- tion. DNS+2e is the DNS extended with one extension point that only contains a NOP. In (Count) the extension point’s stored procedure is changed into a counter, and in (Trace) it is changed to emulate the behaviour of the ELA.

These results confirm that the resources overhead is min- imal, making PhD a feasible solution. We note that in the use-cases detailed below the FPGA resources were never ex- hausted, and consumed less than 25% of the logic resources, even for complex services.

In addition to the quantitative evaluation, we note that PhD is vendor-neutral and runtime-reconfigurable, and can be used for remote in-field debugging, whereas standard techniques for FPGA debugging do not provide this.

**Artefact Utilisation (%) Performance**

of languages for debugging is a decades old subject [(Balzer](#_bookmark30)

Logic Flip-flops

Duration (#cycles)

Latency (*µ*s)

Queries- per-sec (KQPS)

[1969)](#_bookmark30) and includes sophisticated languages for high-level querying of programs [(Johnson](#_bookmark55) [1977;](#_bookmark55) [Golan and Hanson](#_bookmark48) [1993;](#_bookmark48) [Winterbottom](#_bookmark87) [1994).](#_bookmark87) Compared to that work, our

DNS 100.00 100.00 57 1.85 1176

+**R** 103.40 102.76 - 1.85 1176

work separates more starkly between the role (and language) of the director, and that of the controller, which have separate

**Table 3.** Profile of utilisation and performance. **R**ead, **W**rite, and **I**ncrement are instructions supported by the con- troller. Latency is indicated at the 99*th* percentile. The hard- ware generation process involves an optimisation step to place and route components, and on occasion this results in more utilisation-efficient allocations. The duration for exten- sions to DNS (clock cycles) did not change since the critical path of the circuit was not affected.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| +**W** | 115.05 | 106.04 | - | 1.84 | 1176 | languages. This separation guided our modelling in [§3.](#_bookmark3) |
| +**I** | 109.79 | 106.12 | - | 1.84 | 1176 | ***Dynascope.*** Dynascope [(Sosicˇ 1992)](#_bookmark81) provides an extremely |
| Memcached | 100.00 | 100.00 | 64 | 2.03 | 952 | fine grained *execution stream* of events–at the level of ma- |
| +**R** | 99.17 | 100.29 | - | 2.03 | 952 | chine instructions–providing a complete description of a |
| +**W** | 99.80 | 100.74 | - | 2.04 | 952 | program’s runtime behaviour. We provide selective (and |
| +**I** | 100.63 | 100.69 | - | 2.03 | 952 | programmable) visibility by default, in the interest of perfor- |

**Artefact Utilisation (%) Performance**

mance. A more detailed stream could be produced if wished. Dynascope is generative by default (since you need to set fil- ters to ignore events) whereas we are generative by direction (you only get events that you inserted code to generate).

***High-level synthesis and runtime debugging.*** The closest work is that by [McKechnie et al. (2009)](#_bookmark60) who adapt a debug- ging paradigm first developed for network-on-chip devices (NoC). They provide *transaction-level* granularity, consist- ing of domain-specific (high-level) events–less fine grained than Dynascope’s execution stream. Compared to our work

Logic Flip-flops

Duration (#cycles)

Latency (*µ*s)

Queries- per-sec (KQPS)

they offer a less flexible interface, but they take a more de- tailed account of different sorts of interconnects between components–formats such as LocalLink and PLB.

Many other systems inject code to emit and observe

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DNS+ELA | 99.74 | 100.40 | 57 | 1.83 | 1176 | events to aid with debugging of complex designs, such as |
| DNS+2e | 234.61 | 151.06 | 57 | 1.86 | 1176 | “System on Chip” designs [(Lee and Lysecky 2015).](#_bookmark59) SeaCu- |
| **(Count)** | 234.46 | 151.81 | 62 | 1.94 | 1064 | cumber [(Tripp et al. 2002)](#_bookmark85) was the first to support source- |
| **(Trace)** | 218.30 | 151.84 | 70 | 1.99 | 1010 | level (HLS) debugging, both during behavioural simula- |

**Table 4.** Utilisation and performance profile of the DNS+ELA against the DNS having one extension point, where the extension point is NOP, packet counting, or vari- able tracing. Latency is indicated at the 99*th* percentile.

***Test setup.*** We use a host running Ubuntu server 14.04LTS, kernel version 4.4.0-45-generic. The host hardware is a sin- gle 3.5GHz Intel Xeon E5-2637 v4 on a SuperMicro X10- DRG-Q motherboard. An Endace 9.2SX2 DAG card (7.5ns time-stamping resolution) and a NetOptics passive-optical tap are used to intercept client-server traffic and permit in- dependent measurement of client & server latency. For the throughput tests, OSNT [(Antichi et al. 2014)](#_bookmark26) is used to con- trol the rate of the generated requests.

# Related work

***Portable debugging, program directing, and debugging languages.*** We were inspired by previous work on portable debugging [(Ramsey and Hanson](#_bookmark72) [1992;](#_bookmark72) [Hood](#_bookmark52) [1996;](#_bookmark52) [Han-](#_bookmark51) [son and Raghavachari 1996)](#_bookmark51) and program directing [(Sosicˇ](#_bookmark82) [1995).](#_bookmark82) That work usually makes the assumption that soft- ware is compiled to run on a general-purpose CPU however, whereas we also target reconfigurable hardware. The study

tion and during hardware execution. [Monson and Hutchings](#_bookmark64) [(2015)](#_bookmark64) take a different approach: source-code transformation (at the HLS level) to introduce “event observability ports” to enable runtime visibility of variables’ values –but note that it’s not always possible to observe an expression an inter- est. We took the approach of [Monson and Hutchings (2015),](#_bookmark64) relying on source transformation rather than on the HLS system.

Overhead can be reduced by being selective about what to monitor, rather than monitoring everything by default. This was studied by [Goeders and Wilton (2016)](#_bookmark47) who compared the amount of visibility afforded by different schemes when recording events such as reads and updates. We left the choice of what to observe to the programmer.

At present the LegUp HLS system [(Calagar et al. 2014)](#_bookmark32) appears to provide the best support for debugging. In a con- tribution to that system, [Hung and Wilton](#_bookmark54) [(2014)](#_bookmark54) take a different approach to us: they use a two-step incremental compilation, the second step of which compiles the moni- toring system by reclaiming unused FPGA resources. This approach is less likely to interfere with the timing behaviour of the observed circuit. Their model is more specialised to trace-buffers, and it would be interesting to generalise it to support our directing controller.

Testing of hardware is traditionally concentrated on the RTL description, where tools and techniques have been de- veloped to verify designs prior to synthesis (F[oster 2008;](#_bookmark43) [Fix](#_bookmark42) [2008)](#_bookmark42) . The difficulty of checking hardware entirely prior to synthesis has led to research into the inclusion of run- time monitors in hardware [(Todman et al.](#_bookmark84) [2015).](#_bookmark84) [Curreri](#_bookmark36) [et al. (2011)](#_bookmark36) describe a system that translates source-level assertions into monitors. This technique is also used to de- tect hangs and possible timing overruns. We currently do not support such monitoring, since it requires a source-level no- tion of time that we currently do not provide, and “watchdog threads” that we currently do not include.

Finally, [Camera et al. (2005)](#_bookmark33) described a hardware OS, the Berkeley Operating system for ReProgrammable Hard- ware (BORPH), on which user programs were run. Their *stitcher* extend the user program to support debugging, pro- viding a rich system. Like BORPH we require the designer to provide “hints to the system of what aspects of the design may need to be explored at runtime.”

***Formalising program directors.*** Since ‘debugging’ is such a vague term (compared to ‘compiling’, which has a clearer functional behaviour), its verification objective is hard to for- malise. Perhaps as a consequence of this there has been lit- tle work on formalising and verifying debuggers, and usu- ally entirely theoretical [(Zhu 2001).](#_bookmark88) [Kishon et al. (1991)](#_bookmark56) de- scribe transformations over functional programs to include monitoring behaviour. Compared to this work, our transfor- mations are not based on continuations (and we deliberately avoid needing first-order functions to avoid departing too much from the conventional hardware programming mind- set). We use an approach based on operational semantics, similar to that used by [da Silva (1992),](#_bookmark37) but different in sev- eral ways: (i) we support a different set of directing com- mands, (ii) we don’t insert commands into the bytecode for the debugger to keep track of separations between subex- pressions, and to keep track of the path through the ex- pression (program), (iii) [da Silva (1992),](#_bookmark37) introduces a lan- guage for specifying debuggers whereas we introduce an operational language for inspecting and changing the pro- gram’s runtime state, (iv) we do not consider the equiva- lence between debuggers. [Sansom and Peyton Jones (1997)](#_bookmark73) describe the profiling of functional programs by using “cost centres”, a paradigm to identify locations in a program at an expression-level (rather than functional-level) granularity. Others have continued that to make it more practical (F[ad-](#_bookmark40) [degon and Chitil 2015).](#_bookmark40) All the techniques described above rely on program transformation. They all use functional lan- guages as examples whereas we use a simpler imperative language, to model the paradigm of register-level hardware programming more directly. Because of the weaving of de- bugging code into the program all these techniques can (ret- rospectively) be seen as special-cases of aspect-oriented pro- gramming, described next.

***Aspect-oriented programming (AOP).*** AOP involves the inclusion (“weaving”) of code (called “advice”) during compile-time or run-time, depending on whether certain compile- or run-time conditions are satisfied. AOP is a linch- pin paradigm for tracing and monitoring [(Avgustinov et al.](#_bookmark27) [2006;](#_bookmark27) [Hamlen and Jones 2008)](#_bookmark50) but advice can be arbitrary functions, and as a consequence they might have undesirable effects on runtime [(Avgustinov et al. 2007).](#_bookmark28) This is an impor- tant characteristic to our work, where we wanted to reduce the power of added code. [Djoko Djoko et al. (2012)](#_bookmark39) have categorised advice based on the degree of influence they can have on the observable behaviour of a program; and [Dantas](#_bookmark38) [and Walker (2006)](#_bookmark38) characterise less intrusive advice in their work on “harmless advice”.

# Conclusion

Having poor programming and debugging support hinders the potential of computing architectures such as FPGAs, which are gaining importance in modern datacentres. By using a *program directing* approach we subsume several activities that can involve the interactive runtime analyses of programs [(Sosicˇ 1992).](#_bookmark81) Our language-based approach has two extensibility benefits: (i) more efficient controllers can be implemented without changing the source language, and

(ii) third-parties could extend or customise the language of direction commands without changing the controller that these commands compile to.

Using a program director brings security risks, since it may alter the control-flow of a program, and this may be ex- ploited [(Abadi et al. 2005).](#_bookmark24) We sought to mitigate this risk by making the presence of the debug mode very apparent, to reduce the chances of excessive debug functionality being included in deployment (P[auli 2016).](#_bookmark68) Dynascope was used to diagnose errors in the Dynascope compiler itself [(Sosicˇ](#_bookmark81) [1992,](#_bookmark81) §3.1), but bugs could render such a task impossible, for reasons similar to the hereditary potential of vulnera- bilities in compilers [(Thompson 1984).](#_bookmark83) This highlights the preference to have the code for the director be “correct by construction”, preferably automatically generated.

In work such as this we invariable come across the *ob- server effect* [(Mytkowicz et al. 2008)](#_bookmark65) that monitoring code has on the monitored code. This is known to affect the vis- ibility of timing bugs in software [(Neville-Neil 2014),](#_bookmark66) but even “invisible” such bugs can leak important information to an adversary [(Cock et al. 2014).](#_bookmark35)

The overhead due to monitoring is a very important con- sideration when evaluating system measurements, and mon- itoring systems try to do their utmost to reduce it [(Gregg and](#_bookmark49) [Mauro 2011;](#_bookmark49) [Anderson et al. 2014).](#_bookmark25) A key weakness of our prototype is that it does not use low-level (RTL-level) tech- niques to reduce overhead, but we mitigate this by allowing extension points to be specialised, e.g., these may only be a breakpoint, a watchpoint, or both, etc. This simplifies the circuitry we get. A more sophisticated approach would in-

volve organising the director to operate in a different clock domain if possible, following [McKechnie et al. (2009).](#_bookmark60) Not having to sustain a large clock-distribution network leads to power saving, since unused logic blocks don’t need to be switched, and less heat is dissipated from leakage. It also makes placement and routing easier, which usually reduces the compilation time.

The main strength of our approach is that it provides a uniform interface for the flexible directing of software and hardware instances of programs at runtime. To our knowl- edge no other system provides this. We used an HLS system that allows us to run the resulting code both on software and on hardware, but the debugging methods used for either were hitherto separate.

# Acknowledgments

We thank the many people who contributed to this paper. Matthew Grosvenor helped us with evaluation ideas and reusing the QJump infrastructure. Olaf Chitil, Paolo Costa, Klaus Gleissenthall, Tim Harris, Simon Moore, and Robert Soulé helped improve the paper through their feedback.

This work has received funding from the EPSRC NaaS grant EP/K034723/1, European Union’s Horizon 2020 re- search and innovation programme 2014-2018 under the SSI- CLOPS (grant agreement No. 644866), and the Leverhulme Trust Early Career Fellowship ECF-2016-289.

# References

M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Com- puter and Communications Security*, CCS ’05, pages 340–353, New York, NY, USA, 2005. ACM. ISBN 1-59593-226-7. doi: 10.1145/1102120.1102165. URL [http://doi.acm.org/10. 1145/1102120.1102165](http://doi.acm.org/10.1145/1102120.1102165).

J. Anderson, R. N. M. Watson, D. Chisnall, K. Gudka, I. Marinos, and B. Davis. TESLA: Temporally Enhanced System Logic Assertions. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys ’14, pages 19:1–19:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592801. URL [http://doi.acm.org/10. 1145/2592798.2592801](http://doi.acm.org/10.1145/2592798.2592801).

G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington,

M. Bruyere, N. McKeown, N. Feamster, B. Felderman, M. Blott, et al. OSNT: Open source network tester. *IEEE Network*, 28(5): 6–12, 2014.

P. Avgustinov, E. Bodden, E. Hajiyev, L. Hendren, O. Lhoták,

1. de Moor, N. Ongkingco, D. Sereni, G. Sittampalam, J. Tib- ble, and M. Verbaere. Aspects for Trace Monitoring. In *Proceedings of the First Combined International Conference on Formal Approaches to Software Testing and Runtime Ver- ification*, FATES’06/RV’06, pages 20–39, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-49699-8, 978-3-540-49699- 1. doi: 10.1007/11940197\_2. URL [http://dx.doi.org/10. 1007/11940197\_2](http://dx.doi.org/10.1007/11940197_2).
2. Avgustinov, J. Tibble, and O. de Moor. Making Trace Monitors Feasible. *SIGPLAN Not.*, 42(10):589–608, Oct. 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297070. URL [http://doi. acm.org/10.1145/1297105.1297070](http://doi.acm.org/10.1145/1297105.1297070).

T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. Mc- Garvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thor- ough Static Analysis of Device Drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, Apr. 2006. ISSN 0163-5980. doi: 10.1145/1218063.1217943. URL [http://doi.acm.org/10. 1145/1218063.1217943](http://doi.acm.org/10.1145/1218063.1217943).

R. M. Balzer. EXDAMS: EXtendable Debugging and Monitor- ing System. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, AFIPS ’69 (Spring), pages 567– 580, New York, NY, USA, 1969. ACM. doi: 10.1145/1476793. 1476881. URL [http://doi.acm.org/10.1145/1476793. 1476881](http://doi.acm.org/10.1145/1476793.1476881).

R. G. Bennetts and A. Osseyran. IEEE standard 1149.1-1990 on boundary scan: History, literature survey, and current status. *Journal of Electronic Testing*, 2(1):11–25, 1991. ISSN 1573- 0727. doi: 10.1007/BF00134941. URL [http://dx.doi.org/ 10.1007/BF00134941](http://dx.doi.org/10.1007/BF00134941).

N. Calagar, S. D. Brown, and J. H. Anderson. Source-level debug- ging for FPGA high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2014. doi: 10.1109/FPL.2014.6927496.

K. Camera, H. K.-H. So, and R. W. Brodersen. An Integrated De- bugging Environment for Reprogrammble Hardware Systems. In *Proceedings of the Sixth International Symposium on Auto- mated Analysis-driven Debugging*, AADEBUG’05, pages 111– 116, New York, NY, USA, 2005. ACM. ISBN 1-59593-050-7. doi: 10.1145/1085130.1085145. URL [http://doi.acm.org/ 10.1145/1085130.1085145](http://doi.acm.org/10.1145/1085130.1085145).

T. V. Chu, S. Sato, and K. Kise. Ultra-fast NoC emulation on a single FPGA. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2015. doi: 10.1109/FPL.2015.7294021.

D. Cock, Q. Ge, T. Murray, and G. Heiser. The Last Mile: An Empirical Study of Timing Channels on seL4. In *Pro- ceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’14, pages 570–581, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660294. URL [http://doi.acm.org/10. 1145/2660267.2660294](http://doi.acm.org/10.1145/2660267.2660294).

J. Curreri, G. Stitt, and A. D. George. High-level Synthesis of In-circuit Assertions for Verification, Debugging, and Timing Analysis. *Int. J. Reconfig. Comput.*, 2011:1:1–1:17, Jan. 2011. ISSN 1687-7195. doi: 10.1155/2011/406857. URL [http:](http://dx.doi.org/10.1155/2011/406857)

[//dx.doi.org/10.1155/2011/406857](http://dx.doi.org/10.1155/2011/406857).

F. Q. B. da Silva. *Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics*. PhD thesis, School of Informatics, University of Edinburgh, Oct. 1992.

D. S. Dantas and D. Walker. Harmless Advice. In *Confer- ence Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’06, pages 383– 396, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2.

doi: 10.1145/1111037.1111071. URL [http://doi.acm.org/](http://doi.acm.org/10.1145/1111037.1111071) [10.1145/1111037.1111071](http://doi.acm.org/10.1145/1111037.1111071).

S. Djoko Djoko, R. Douence, and P. Fradet. Aspects Preserving Properties. *Sci. Comput. Program.*, 77(3):393–422, Mar. 2012. ISSN 0167-6423. doi: 10.1016/j.scico.2011.10.010. URL [http:](http://dx.doi.org/10.1016/j.scico.2011.10.010)

[//dx.doi.org/10.1016/j.scico.2011.10.010](http://dx.doi.org/10.1016/j.scico.2011.10.010).

M. Faddegon and O. Chitil. Algorithmic Debugging of Real-world Haskell Programs: Deriving Dependencies from the Cost Centre Stack. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’15, pages 33–42, New York, NY, USA, 2015. ACM. ISBN 978- 1-4503-3468-6. doi: 10.1145/2737924.2737985. URL [http:](http://doi.acm.org/10.1145/2737924.2737985)

[//doi.acm.org/10.1145/2737924.2737985](http://doi.acm.org/10.1145/2737924.2737985).

B. Fitzpatrick. Distributed caching with memcached. *Linux Jour- nal*, 2004(124), 2004.

L. Fix. *Fifteen Years of Formal Property Verification in Intel*, pages 139–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-69850-0. doi: 10.1007/978-3-540-69850-0\_8. URL <http://dx.doi.org/10.1007/978-3-540-69850-0_8>.

H. Foster. *Assertion-Based Verification: Industry Myths to Re- alities*, pages 5–10. Springer Berlin Heidelberg, Berlin, Hei- delberg, 2008. ISBN 978-3-540-70545-1. doi: 10.1007/ 978-3-540-70545-1\_3. URL [http://dx.doi.org/10.1007/ 978-3-540-70545-1\_3](http://dx.doi.org/10.1007/978-3-540-70545-1_3).

M. N. Gagnon, S. Taylor, and A. K. Ghosh. Software protection through anti-debugging. *IEEE Security Privacy*, 5(3):82–84, May 2007. ISSN 1540-7993. doi: 10.1109/MSP.2007.71.

P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM*, 55(3):40–44, Mar. 2012. ISSN 0001-0782. doi: 10.1145/2093548.2093564. URL <http://doi.acm.org/10.1145/2093548.2093564>.

J. Goeders and S. J. E. Wilton. Effective FPGA Debug for High- Level Synthesis Generated Circuits. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2014. doi: 10.1109/FPL.2014.6927498.

J. Goeders and S. J. E. Wilton. Quantifying observability for in- system debug of high-level synthesis circuits. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–11, Aug 2016. doi: 10.1109/FPL. 2016.7577371.

M. Golan and D. R. Hanson. Duel – A Very High-Level Debug- ging Language. In *USENIX Winter Conference*, pages 107–117, 1993.

B. Gregg and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

K. W. Hamlen and M. Jones. Aspect-oriented In-lined Reference Monitors. In *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS ’08, pages 11–20, New York, NY, USA, 2008. ACM. ISBN 978- 1-59593-936-4. doi: 10.1145/1375696.1375699. URL [http:](http://doi.acm.org/10.1145/1375696.1375699)

[//doi.acm.org/10.1145/1375696.1375699](http://doi.acm.org/10.1145/1375696.1375699).

D. R. Hanson and M. Raghavachari. A machine-independent de- bugger. *Softw., Pract. Exper.*, 26(11):1277–1299, 1996.

R. Hood. The P2D2 Project: Building a Portable Distributed De- bugger. In *Proceedings of the SIGMETRICS Symposium on Par-*

*allel and Distributed Tools*, SPDT ’96, pages 127–136, New York, NY, USA, 1996. ACM. ISBN 0-89791-846-0. doi: 10.

1145/238020.238058. URL [http://doi.acm.org/10.1145/](http://doi.acm.org/10.1145/238020.238058)

[238020.238058](http://doi.acm.org/10.1145/238020.238058).

E. Hung and S. J. Wilton. Towards Simulator-like Observability for FPGAs: A Virtual Overlay Network for Trace-buffers. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA ’13, pages 19–28, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1887-7. doi: 10.1145/2435264.2435272. URL [http://doi.acm.org/10. 1145/2435264.2435272](http://doi.acm.org/10.1145/2435264.2435272).

E. Hung and S. J. E. Wilton. Accelerating FPGA Debug: Increas- ing Visibility Using a Runtime Reconfigurable Observation and Triggering Network. *ACM Trans. Des. Autom. Electron. Syst.*, 19(2):14:1–14:23, Mar. 2014. ISSN 1084-4309. doi: 10.1145/ 2566668. URL <http://doi.acm.org/10.1145/2566668>.

M. S. Johnson. The Design of a High-level, Language-independent Symbolic Debugging System. In *Proceedings of the 1977 Annual Conference*, ACM ’77, pages 315–322, New York, NY, USA, 1977. ACM. ISBN 978-1-4503-3921-6. doi: 10. 1145/800179.810221. URL [http://doi.acm.org/10.1145/ 800179.810221](http://doi.acm.org/10.1145/800179.810221).

A. Kishon, P. Hudak, and C. Consel. Monitoring Semantics: A Formal Framework for Specifying, Implementing, and Rea- soning About Execution Monitors. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI ’91, pages 338–352, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: 10. 1145/113445.113474. URL [http://doi.acm.org/10.1145/ 113445.113474](http://doi.acm.org/10.1145/113445.113474).

G. H. Koch, W. Rosenstiel, and U. Kebschull. Breakpoints and Breakpoint Detection in Source-level Emulation. *ACM Trans. Des. Autom. Electron. Syst.*, 3(2):209–230, Apr. 1998. ISSN 1084-4309. doi: 10.1145/290833.290843. URL [http://doi. acm.org/10.1145/290833.290843](http://doi.acm.org/10.1145/290833.290843).

S. Krishnaswamy, I. L. Markov, and J. P. Hayes. On the Role of Timing Masking in Reliable Logic Circuit Design. In *Proceed- ings of the 45th Annual Design Automation Conference*, DAC ’08, pages 924–929, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6. doi: 10.1145/1391469.1391703. URL <http://doi.acm.org/10.1145/1391469.1391703>.

J. C. Lee and R. Lysecky. System-Level Observation Framework for Non-Intrusive Runtime Monitoring of Embedded Systems. *ACM Trans. Des. Autom. Electron. Syst.*, 20(3):42:1–42:27, June 2015. ISSN 1084-4309. doi: 10.1145/2717310. URL [http:](http://doi.acm.org/10.1145/2717310)

[//doi.acm.org/10.1145/2717310](http://doi.acm.org/10.1145/2717310).

P. E. McKechnie, M. Blott, and W. A. Vanderbauwhede. De- bugging FPGA-based Packet Processing Systems Through Transaction-level Communication-centric Monitoring. *SIG- PLAN Not.*, 44(7):129–136, June 2009. ISSN 0362-1340. doi: 10.1145/1543136.1542470. URL [http://doi.acm.org/10. 1145/1543136.1542470](http://doi.acm.org/10.1145/1543136.1542470).

1. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
2. Mittal. A survey of techniques for improving energy efficiency in embedded computing systems. *International Journal of Com- puter Aided Engineering and Technology*, 6(4):440–459, 2014.

P. Mockapetris. Domain names – concepts and facilities, Nov. 1987. <https://tools.ietf.org/html/rfc1034>.

J. S. Monson and B. L. Hutchings. Using Source-Level Transfor- mations to Improve High-Level Synthesis Debug and Validation on FPGAs. In *Proceedings of the 2015 ACM/SIGDA Interna- tional Symposium on Field-Programmable Gate Arrays*, FPGA ’15, pages 5–8, New York, NY, USA, 2015. ACM. ISBN 978- 1-4503-3315-3. doi: 10.1145/2684746.2689087. URL [http:](http://doi.acm.org/10.1145/2684746.2689087)

[//doi.acm.org/10.1145/2684746.2689087](http://doi.acm.org/10.1145/2684746.2689087).

T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan. Ob- server Effect and Measurement Bias in Performance Analysis. Technical Report 1042-08, University of Colorado at Boulder, 6 2008.

G. V. Neville-Neil. Outsourcing Responsibility. *Commun. ACM*, 57(10):28–29, Sept. 2014. ISSN 0001-0782. doi: 10.1145/ 2661051. URL <http://doi.acm.org/10.1145/2661051>.

Z. Panjkov, A. Wasserbauer, T. Ostermann, and R. Hagelauer. Hy- brid FPGA debug approach. In *2015 25th International Con- ference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2015. doi: 10.1109/FPL.2015.7294023.

D. Pauli. ‘Pork Explosion’ flaw splatters Foxconn’s Android phones. Oct 2016. [http://www.theregister.co.uk/2016/ 10/14/pork\_explosion\_foxconn\_flaw/](http://www.theregister.co.uk/2016/10/14/pork_explosion_foxconn_flaw/).

M. Potkonjak, S. Dey, and K. Wakabayashi. Design-for-debugging of Application Specific Designs. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-aided De- sign*, ICCAD ’95, pages 295–301, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7213-7. URL [http:](http://dl.acm.org/citation.cfm?id=224841.225054)

[//dl.acm.org/citation.cfm?id=224841.225054](http://dl.acm.org/citation.cfm?id=224841.225054).

A. Putnam. Large-Scale Reconfigurable Computing in a Microsoft Datacenter. In *Proceedings of the 26th IEEE HotChips Sympo- sium on High-Performance Chips (HotChips 2014)*. IEEE, Au- gust 2014.

A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constan- tinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal,

J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim,

S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong,

P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerat- ing large-scale datacenter services. In *2014 ACM/IEEE 41st In- ternational Symposium on Computer Architecture (ISCA)*, pages 13–24, June 2014. doi: 10.1109/ISCA.2014.6853195.

N. Ramsey and D. R. Hanson. A Retargetable Debugger. In *Pro- ceedings of the ACM SIGPLAN 1992 Conference on Program- ming Language Design and Implementation*, PLDI ’92, pages 22–31, New York, NY, USA, 1992. ACM. ISBN 0-89791-475- 9. doi: 10.1145/143095.143112. URL [http://doi.acm.org/ 10.1145/143095.143112](http://doi.acm.org/10.1145/143095.143112).

P. M. Sansom and S. L. Peyton Jones. Formally Based Profil- ing for Higher-order Functional Languages. *ACM Trans. Pro- gram. Lang. Syst.*, 19(2):334–385, Mar. 1997. ISSN 0164-0925. doi: 10.1145/244795.244802. URL [http://doi.acm.org/10. 1145/244795.244802](http://doi.acm.org/10.1145/244795.244802).

A. Sari, D. Agiakatsikas, and M. Psarakis. A Soft Error Vulnera- bility Analysis Framework for Xilinx FPGAs. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field- programmable Gate Arrays*, FPGA ’14, pages 237–240, New

York, NY, USA, 2014. ACM. ISBN 978-1-4503-2671-1. doi:

10.1145/2554688.2554767. URL [http://doi.acm.org/10.](http://doi.acm.org/10.1145/2554688.2554767)

[1145/2554688.2554767](http://doi.acm.org/10.1145/2554688.2554767).

M. Schäfer, T. Ekman, and O. de Moor. Challenge Proposal: Verification of Refactorings. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV ’09, pages 67–72, New York, NY, USA, 2008. ACM. ISBN 978- 1-60558-330-3. doi: 10.1145/1481848.1481859. URL [http:](http://doi.acm.org/10.1145/1481848.1481859)

[//doi.acm.org/10.1145/1481848.1481859](http://doi.acm.org/10.1145/1481848.1481859).

J. C. Shepherdson and H. E. Sturgis. Computability of Recursive Functions. *J. ACM*, 10(2):217–255, Apr. 1963. ISSN 0004-5411. doi: 10.1145/321160.321170. URL [http://doi.acm.org/10. 1145/321160.321170](http://doi.acm.org/10.1145/321160.321170).

S. Singh and D. J. Greaves. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *Field-Programmable Custom Computing Machines*, pages 3–12, 2008.

C. H. Smith and J. van Leeuwen. Microprogrammed Random Access Stored Program Machines. *SIGACT News*, 6(3):23–32, July 1974. ISSN 0163-5700. doi: 10.1145/1008311.1008315. URL <http://doi.acm.org/10.1145/1008311.1008315>.

B. So, M. W. Hall, and P. C. Diniz. A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Sys- tems. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI ’02, pages 165–176, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512550. URL [http:](http://doi.acm.org/10.1145/512529.512550)

[//doi.acm.org/10.1145/512529.512550](http://doi.acm.org/10.1145/512529.512550).

R. Sosicˇ and D. Abramson. Guard: A relative debugger. *Software: Practice and Experience*, 27(2):185–206, 1997.

R. Sosicˇ. Dynascope: A Tool for Program Directing. *SIGPLAN Not.*, 27(7):12–21, July 1992. ISSN 0362-1340. doi: 10. 1145/143103.143110. URL [http://doi.acm.org/10.1145/ 143103.143110](http://doi.acm.org/10.1145/143103.143110).

R. Sosicˇ. A Procedural Interface for Program Directing. *Software: Practice and Experience*, 25(7):767–787, 1995. ISSN 1097- 024X. doi: 10.1002/spe.4380250704. URL [http://dx.doi. org/10.1002/spe.4380250704](http://dx.doi.org/10.1002/spe.4380250704).

K. Thompson. Reflections on Trusting Trust. *Commun. ACM*, 27(8):761–763, Aug. 1984. ISSN 0001-0782. doi: 10. 1145/358198.358210. URL [http://doi.acm.org/10.1145/ 358198.358210](http://doi.acm.org/10.1145/358198.358210).

T. Todman, S. Stilkerich, and W. Luk. In-circuit Temporal Monitors for Runtime Verification of Reconfigurable Designs. In *Proceed- ings of the 52Nd Annual Design Automation Conference*, DAC ’15, pages 50:1–50:6, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3520-1. doi: 10.1145/2744769.2744856. URL <http://doi.acm.org/10.1145/2744769.2744856>.

J. L. Tripp, P. A. Jackson, and B. L. Hutchings. *Sea Cucumber: A Synthesizing Compiler for FPGAs*, pages 875–885. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540- 46117-3. doi: 10.1007/3-540-46117-5\_90. URL [http://dx. doi.org/10.1007/3-540-46117-5\_90](http://dx.doi.org/10.1007/3-540-46117-5_90).

D. Wang, N. E. Jerger, and J. G. Steffan. DART: A Pro- grammable Architecture for NoC Simulation on FPGAs. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS ’11, pages 145–152, New York,

NY, USA, 2011. ACM. ISBN 978-1-4503-0720-8. doi:

10.1145/1999946.1999970. URL [http://doi.acm.org/10.](http://doi.acm.org/10.1145/1999946.1999970)

[1145/1999946.1999970](http://doi.acm.org/10.1145/1999946.1999970).

P. Winterbottom. ACID: A Debugger Built From A Language. In

*D*˘ = *{}*

*C*˘ = *{}*

****

**** *j X* ~ *cur* \_*idx* ;*D*

*Dc* = *λ .* i

for *i* = *cur* \_*idx* to 0:

*USENIX Winter Conference*, pages 211–222, 1994.

M.-Y. Zhu. Formal Specifications of Debuggers. *SIGPLAN Not.*, 36(9):54–63, Sept. 2001. doi: 10.1145/609769.609778. URL

<http://doi.acm.org/10.1145/609769.609778>.

N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore.

****

### A.2 Watching

****

*X*a[*i*] ~ *N* ;

print(*N* );

*Dj*

NetFPGA SUME: Toward 100 Gbps as Research Commodity.

*IEEE Micro*, 34(5):32–41, 2014.

# More program direction commands

Program-level predicates and functions mentioned in these definitions are given in [§B.](#_bookmark92)

### Tracing

We continue from [§3.5.2,](#_bookmark15) where the symbols we use here are introduced (such as *X*i and *X*of ).

m

This bears some similarity to the rule to start tracing (in [§3.5.2)](#_bookmark15) since we inspect the variable after it’s updated, and carry out some action. When tracing, this action consists of writing to the trace buffer. When watching, it consists of breaking.

*X ∈* Var*p ∀L ∈ XL. L ƒ∈ p*

*j*

Positions*pt* (*XL*) = PostUpdate*pt* (*X*) *p <XL p*

mC

*p* watch *X (B) pj*

**** *D*˘ =.*{*(«w»*, X,* 1)*}* Σ

*D ∈ D D*

****

˘

*∈*

(trace start *X (B) (*$*)*) *∈* C

**** *C* =

*SP* [*L ›→* ¢watch *X (B)*¢*SP* ]

*D*˘ = *{}*

****

*C*˘ = *{}*

trace clear *X*

*p* C *p*

for each *L XL*

*Dc* = *λ j.* if («w»*, X,* 1) *j* then *j*

else

for each *L ∈ XL*:

where

(«w»*, X,* 0 *›→* 1) :*∈ Dj*

### 

**** *Dc* = *λDj.* ¢trace clear *X*¢ ~ 0;

¢ ¢

where

*Dj*

**** @*L* : *{*¢watch *X (B)*¢*SP }* ~ ’*L*’;

trace clear *X* =

*X*i := 0;

*X*of := 0

Next we add the command to check whether the trace buffer is full.

(trace start *X (B) (*$*)*) *∈* C

trace full *X*

watch *X (B) SP* = conditional *(B)* break

Unwatching a variable follows a pattern we’ve encoun-

¢ ¢

tered before, when stopping tracing for instance.

(watch *X (B)*) *∈* C

Positions*p*(*XL*) = PostUpdate*p*(*X*)

unwatch *X*

*D*˘ = *{}*

****

 *C*˘ = *{}*

*p* mC *p*

*D*˘ = *{}*

*C*˘ = *{}*

****

****

*p* mC *p*

**** *Dc* = *λDj.* ¢trace full *X*¢ ~ *N* ;

 *Dc* = *λDj.* if («w»*, X,* 0) *∈ Dj* then *Dj*

where

print(*N* );

*Dj*

else

for each *L ∈ XL*:

**** @*L* : unwatch *X* ~ ’*L*’;

*{ SP }*



¢ ¢

(«w»*, X,* 1 *›→* 0) :*∈ Dj*

trace full *X* = *X*of

¢ ¢

Finally the command to retrieve the contents of the trace buffer. Here we first find out the current index value for the trace buffer, then we work back and extract the buffer’s contents.

where

unwatch *X SP* = continue

¢ ¢

### A.3 Profiling

(trace start *X (B) (*$*)*) *∈* C

trace print *X*

m

*p* C *p*

We could profile programs to count the number of writes

and reads of variables, or function calls. We show the rule for counting the number of writes; counting variable reads

and function calls are similar, differing in the positioning of labels. That is, to count writes we position labels immedi- ately after each line that updates that variable–this is similar to tracing [(§3.5.2)](#_bookmark15) and watching [(§A.2).](#_bookmark90) To count variable

where *n* is *fdecl* , the number of function declarations in *p*. Let *fdecl* = *τ fname*(*x*¯) *s*; return *e* and *s* = *s*0; *. . .* ; *sn*.

Then

*{ }*

*| |*

reads we position after lines in which those variables appear. To count function calls we position just before the first line

Π*fdecl*

*def*

=

0*≤*[*i≤n*

(*{i} ×* Π*si*

) *∪ {n* + 1*}*

in the function.

## *X ∈* Var *∀L ∈ X . L ƒ∈ p*

*p*

*L*

Finally,

*def*  S0*≤i≤n* (*{i} ×* Π*ri* ) *∪ {n* + 1*}*

Positions*pt* (*XL*) = PostUpdate*pt* (*X*) *p <XL pj*

Π*s*

=

if *s* = if *e* then *r*0; *. . .* ; *rn*



 *∅* otherwise

*p* count write start *X (B) (*$*) pj*

mC

 *D*˘ =*{*(«c»*, X,* 1)*}*

*∈*

****

****  

Next we define a function Pos*p* maps a position *π* Π*p* in *p* to the corresponding statement *s*, in symbols: Pos*p*(*π*) = *s*.

˘ *C* [*X*count] = 0*, C* [*X*of ] = 0*, A*[*X*a[*(*$*)*]]*,*

for each *L XL*

*∈*

**** *C* =  *SP* [*L ›→* ¢count write start *X (B) (*$*)*¢*SP* ]

****

*Dc* = *λ j.* if («c»*, X,* 1) *j* then *j*

*D ∈ D D*

else

****

for each *L ∈ XL*:

****

*{*¢ *( ) ( )*¢ *}*

@*L* : count write start *X B* $ *SP*

~ ’*L*’;

****

 («c»*, X,* 0 *›→* 1) :*∈ Dj*

**Definition B.2** (Statement at a position)**.** Let

*p* = *vdecl fdecl* return *fname*(*e*¯)

Then we define

*def*

Pos*p*((*i, π*)) = FPos*fdecli* (*π*)

Let *fdecl* = *τ fname*(*x*¯) *s*; return *e* and *s* = *s*0; *. . .* ; *sn*. Then

*{ }*

where

¢count write start *X (B) (*$*)*¢*SP* =

conditional *(B)*





FPos

Finally,

*def*

*fdecl* ((*i, π*)) = SPos*si* (*π*)

*j*

if

*j*

if *X*count *< (*$*)* then

inc *X*count;





*def*

SPos*s*(*π*) =

SPos*ri* (*π* )

*π* = (*i, π* )

and *s* = if *e* then *r*0; *. . .* ; *rn*

continue

else





inc *X*of ;

break

*s* if *π* = *∅*

**Definition B.3** (Program *p* contains label *L*)**.** We write *L ∈*

*p* iff *∃π, S.* Pos*p*(*π*) = extend(*S*) *∧ L ∈ S*.



Stopping the count, clearing it, printing the current count, and checking whether the maximum value has been reached, is done in a very similar way to the rules for tracing [(§3.5.2).](#_bookmark15)

# Program analyses and transformations

A position of statement *s* in program *p* is a vector *π* N*d* where 1 *< d < ω*. Intuitively, the first position of the vector refers to the function in which the statement is positioned. The second until the last positions indicate statements (and substatements, in the case of if-then statements) in which the statement is positioned, moving from the outermost to the innermost containing statement. Within the innermost containing statement, if the statement number is *i*, then the last component of the vector is either *i*+0 or *i*+1, depending on whether it points to just before or just after the statement.

*∈*

**Definition B.1** (Positions)**.** Let

*p* = *vdecl fdecl* return *fname*(*e*¯)

Then

Note that a label *L* may only be used at most once in a program. We expect the following to hold for all programs. We do not include this as a premise to any of the rules, to reduce clutter.

*∀π*1*, π*2*, S*1*, S*2*.* (Pos*p*(*π*1) = extend(*S*1) *∧ L ∈ S*1)

*∧* (Pos*p*(*π*2) = extend(*S*2) *∧ L ∈ S*2)

*−→* (*S*1 = *S*2 *∧ π*1 = *π*2)

We now formalise the predicate that programs *p* and *pj* are identical save for the addition of *L* in one of the extension

points.

**Definition B.4** (Program identity modulo label *L*)**.**

*def*

1 *j*

*p < p* = *∀π, s, S.*

*L*

(Pos*p*(*π*) = *s −→*

((*s ƒ*= extend(*S*) *−→* Pos*pt* (*π*) = *s*)

*∧* (*s* = extend(*S*) *−→*

Pos*pt* (*π*) = extend(*Sj*) *∧* (*Sj* = *S ∨ Sj\S ⊆ {L}*))))

*∧* (Pos*pt* (*π*) = *s −→*

*∧* (*s* = extend(*Sj*) *−→*

Pos*p*(*π*) = extend(*S*) *∧* (*Sj* = *S ∨ Sj\S ⊆ {L}*))))

*def* [ . Σ

Π*p*

=

*{i} ×* Π*fdecli*

0*≤i≤n*

((*s* extend(*S*) *−→* Pos*p*(*π*) = *s*)

We define a related predicate that formalises whether two programs are related by the above predicate through an ‘interpolation’ of labels drawn from a set.

**Definition B.5** (Program identity modulo set of labels)**.** Let

*S* = *{L*0*, . . . , Ln}* is a set of labels. Then,

Note that the “continue” and “break” commands change the state of the *ia* component of the configuration, switching it to batch (resuming the program) and interactive respec- tively.

Note also that placement (@*L* : *{P }*) is only allowed during interactive mode (i.e., *ia* = *•*) since we do not

*j def*

1 1 1

1 *j* want the controller to normally run code that updates other

*p <S p*

= *∃p*1*, . . . , pn−*1*. p <L*0 *p*1 *<L*1 *p*2 *<L*2 *. . . <Ln p*

extension points’ code.

**Definition B.6** (Set of variables in program *p*)**.** Let *p* = *vdecl fdecl* return *fname*(*e*¯) and *vdecl* = *vdecl* 0 *. . . vdecl n*. Then

### C.2 Example language

The syntax for the example language was given in [§3.4.1.](#_bookmark12) In this language we have a valuation function for identi-

*∈*

Var*p*

*def*

=

0*≤*[*i≤n*

*{Xi | vdecl i* = *τi Xi}*

fiers *R* (*X ~* Z), and a configuration for the reduc- tion semantics consists of a triple *R, c*¯*, c*¯ where *c* is either a statement, a ‘return’ statement, or an expression:

**Definition B.7** (Set of variables in program *p*)**.**

*( )*

*def*

PostUpdate*p*(*X*) = *{π* + 1 *| ∃E.* Pos*p*(*π*) = (*X* := *E*)*}*

where if *π* = (*m*0*, . . . , mn*) then *π* +1 = (*m*0*, . . . , mn* +1).

The positions in which a label *L* occurs. This should be a singleton set.

**Definition B.8** (Positions of label *L*)**.**

*def*

Positions*p*(*L*) = *{π | ∃S.* Pos*p*(*π*) = extend(*S*) *∧ L ∈ S}*

We lift this to work on sets of labels and overload the nota- tion: if *SL* is a set of labels then

*c* ::=

*s* return *e e*. *c*¯ denotes a sequence of 1 or more such *c*: *c*0*, . . . , cn* where *n >* 0. We define the concatenation op- erator : such that if *d*¯ = *d*0*, . . . , dm* where *m >* 0, then *c*¯ : *d*¯ = *c*0*, . . . , cn, d*0*, . . . , dm*.

*| |*

Intuitively, *R, c*¯1*, c*¯2 describes a configuration where *c*¯1 is to be evaluated “now”, and *c*¯2 is to be executed after *c*¯1 has been evaluated. The evaluation rules are given in Figure [7.](#_bookmark96)

*( )*

To reduce clutter in the rules, we assume two pieces im- plicit state, that we avoid threading around the configura- tions. The first is a mapping *F* from function names to their bodies, and the second is the state of the CASP machine , which includes the valuation *R* that models the store for pro- gram variables.

*S*

We use the following evaluation context for syntactic ob-

*def*

Positions*p*(*SL*) =

*L*[*∈SL*

(Positions*p*(*L*))

jects in *c* (statements, ‘return’ statements, and expressions):

*E* ::= []

# Language semantics

### CASP machines

The specification of CASP machines was given in [§3.2.](#_bookmark8)

The machine’s *configuration* consists of the triple ( *, ia, P* ), where *ia ,* indicates whether the machine is oper- ating in batch or interactive modes respectively, and = (*C, A, SP* ) indicates the machine’s memory: *C* (*X ~* Z) are the counters, *A* (*R ~* Z *~* Z) the arrays, and *SP* (*L ~ P* ) the stored procedures.

*∈*

*∈*

*∈*

*S*

*∈ {◦ •}*

*S*

The machine’s big-step operational semantics are de- scribed using the notation *L* ( *, ia, P* ) = ( *j, iaj, N* ) to indicate that the machine operates in the context of a spe-

*€ S ⇒ S*

cific label *L* to evaluate program *P* into numeral *N* , and possibly updating the other components of its configura- tion. We use the notation *L* ( *, ia, P* ) *N* to abbreviate *L* ( *, ia, P* ) = ( *, ia, N* ) when and *ia* are unaffected by the evaluation.

*€ S ⇒ S S*

*€ S ⇓*

The semantic rules are given in Figure [6.](#_bookmark95) We use the notation ’*L*’ to denote a total injective map from labels to numerals. Thus we can identify which label is being executed. If in interactive mode the director sends “break” to the controller, the director learns the label that broke.

*| X* := *E*

*|* if *E* then *s*; *s*

*| fname*(*N*0*, . . . , Nn, E, e*0 *. . . , em*)

*| E op e*

*| N op E*

*op ∈ {*+*, −,* ==*, <}*

We handle ‘extend’ by expanding it to a sequence of singleton extends, and invoking the rule in Figure [8.](#_bookmark97)

Rules for m model the interactive mode between the di-

rector and the controller. In these semantics we only see evaluation from the point of view of the program: the pro- gram’s evaluation yields to the controller = at exten- sion points, which either yields back to the program (in case of continue) or else switches to interactive mode (in case of break), leading to an interleaving between m (obtaining a command from the director) and = (executing it). Note that the rules for m are side-effecting, and are formalised as such similar to streams as used by [Milner et al. (1990).](#_bookmark61) In the rules for m, *P* is a command obtained from the director, and *N* is a result sent back to the director.

*⇒*

*−→ ⇒*

*L* ( *, ia,* continue) = ( *, ,* ’*L*’) *L* ( *, ia,* break) = ( *, ,* ’*L*’) *L €* (*S, ia, V* ) *⇓ N*

*€ S ⇒ S ◦ € S ⇒ S •*

*L €* (*S, ia, −V* ) *⇓ −N*

.

*L €* (*S, ia, V*1) *⇓ N*1 *L €* (*S, ia, V*2) *⇓ N*2

*L €* (*S, ia, V*1 *op V*2) =*⇒* (*S, ia, N* )

*N* = 1 (*op* = (=) *∧ N*1 = *N*2) *∨* (*op* = (*<*) *∧ N*1 *< N*2)

*−*1 o/w

*L €* (*S, ia, U* ) *⇓ N L €* (*S, ia, op U* ) =*⇒* (*S*[*U ›→ M* ]*, ia, M* )

*M* = *N* + 1 *op* = inc

*N −* 1 *op* = dec

.

*S*(*C, X*) = *N*

*L €* (*S, ia, X*) *⇓ N*

*L €* (*S, ia, E*) *⇓ N L €* (*S, ia, U* := *E*) =*⇒* (*S*[*U ›→ N* ]*, ia, N* )

*L €* (*S, ia, I*) *⇓ N S*(*R*[*A*]*, N* ) = *M L €* (*S, ia, A*[*I*]) *⇓ M*

*L €* (*S, ia, E*) *⇓ N L €* (*S, ia, P* ) =*⇒* (*Sj, iaj, M* )

*L €* (*S, ia,* if *E* then *P*1 else *P*2) =*⇒* (*Sj, iaj, M* )

*P* = *P*1 *N* = 1

*P*2 *N* = *−*1

*L €* (*S, ia, P*1) =*⇒* (*Sj, iaj, N* ) *ia ƒ*= *iaj*

.

*L €* (*S, ia, P*1; *P*2) =*⇒* (*Sj, iaj, N* )

*L €* (*S, ia, P*1) =*⇒* (*Sj, iaj, M* ) *ia* = *iaj L €* (*Sj, iaj, P*2) =*⇒* (*Sjj, iajj, N* )

*L €* (*S, ia, P*1; *P*2) =*⇒* (*Sjj, iajj, N* )

*L €* (*S, •,* @*Lj* : *{P }*) =*⇒* (*S*[*SP* [*Lj*] *›→ P* ]*, •,* ’*Lj*’) *L €* (*S, ia, N* ) *⇓ N*

**Figure 6.** Dynamic semantics for CASP instructions [(§3.2)](#_bookmark8)

*(R,* skip*, s*; *rest) −→ (R, s, rest) (R, N, s*; *rest) −→ (R, s, rest) (R, s*; *r*¯*, rest) −→ (R, s, r*¯ : *rest)*

*(R, X* := *N, rest) −→ (R*[*X ›→ N* ]*, skip, rest) (R, X, rest) −→ (R, R*(*X*)*, rest)*

*N ≤* 0

*(R,* if *N* then *s*¯*, rest) −→ (R, skip, rest)*

*N >* 0

*(R,* if *N* then *s*¯*, rest) −→ (R, s*¯*, rest)*

*(R, e,* skip*) −→ (Rj, N,* skip*) (R, E*[*e*]*, rest) −→ (Rj, E*[*N* ]*, rest)*

****

*N*1 + *N*2 *op* = +

*N*1 *− N*2 *op* = *−*

****



*(R, N*1

*op N*2*,* skip*) −→ (R, M,* skip*) M* =

1 *op* = (==) *∧ N*1 = *N*2

0 *op* = (==) *∧ N*1 *ƒ*= *N*2

****

1 *op* = (*<*) *∧ N*1 *< N*2

0 *op* = (*<*) *∧ N*1 *≥ N*2

(*fname, s*¯; return *e*) *∈ F (R, s*¯; return *e,* skip*) −→ (Rj,* return *N,* skip*)* *(R, fname*(*N*¯ *, rest) −→ (Rj, N, rest)*

**Figure 7.** Dynamic semantics for the example language [(§3.4.1)](#_bookmark12)

*L €* (*S, ◦, SP* [*L*]) =*⇒* (*Sj, ◦, N* )

*(R,* extend*{L}, s*; *rest) −→ (Rj,* skip*, rest)*

*L €* (*S, •, P* ) =*⇒* (*Sj, •, N* ) *L €* (*Sj, •*) m (*Sjj, ◦*)

*L* *€* (*S, •*) m (*Sjj, ◦*)

*L €* (*S, •*) m (*Sj, ◦*)

*L €* (*S, ◦,* break) =*⇒* (*S, ◦,* continue)

*L €* (*S, •, P* ) =*⇒* (*Sj, ◦, N* )

*L €* (*S, •*) m (*Sj, ◦*)

**Figure 8.** Linking of the programming language evaluation with that of CASP controller