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

Nik Sultana[♭] Salvator Galea[♭] David Greaves[♭] Marcin Wójcik[♭] Noa Zilberman[♭] Richard Clegg[♯]
Luo Mai[♮] Richard Mortier[♭] Peter Pietzuch[♮] Jon Crowcroft[♭] Andrew W Moore[♭]
[♭]Cambridge University [♮]Imperial College, London [♯]Queen Mary University, London

Abstract

The capacity and programmability of reconfigurable hardware such as FPGAs has improved steadily over the years, but they do not readily provide any mechanisms for monitoring or debugging running programs. Such mechanisms need to be *written into* the program itself. This is done using *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. Normally 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 impact on hardware utilisation and performance.

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

1. Introduction

When debugging and monitoring programs running on microprocessors we usually benefit from hardware support that is leveraged by an Operating System to inspect and modify running processes (Gagnon et al. 2007). 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 *profiling* 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 reconfigured. 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, §4.4). This makes FPGAs especially appealing for some problems and environments, a recent example being datacentres (Putnam et al. 2014).

Despite their appeal as a computing device, the programmability 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 designers" (So et al. 2002). Research has yielded various approaches 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 improved over the years, but they are not *debuggable* by default (Potkonjak et al. 1995). 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 simulation can be slower by a factor of 10^6 (Camera et al. 2005) 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* (Sosič 1992) 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 *direction commands* at runtime. Extending the program involves inserting (i) named *extension points* which can contain runtime-modifiable code in a computationally weak language (no recursion), and (ii) state to be used for bookkeeping by that code, to implement direction features.

For example, the direction command "`trace` X $\mathrm{max\_trace\_idx}$" (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 "$\mathrm{max\_trace\_idx}$" for later inspection). We translate this command into (i) a snippet that gets injected into the program (Figure 1) and (ii) a modification 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. Otherwise 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, and the overall scheme is shown in Figure 2.

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 hardware, 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 ensure that the update is considered for logging.

Extension points are added at compile time depending on what direction commands we want to support at runtime. For example, starting with the sequence of statements $\ldots; s_i; s_{i+1}; \ldots$ we insert two extension points $@L$ and $@M$, resulting in the program $\ldots; s_i; @L; s_{i+1}; @M; \ldots$. When extension point L is reached at runtime, a stored procedure associated with L is executed by the controller. This procedure ultimately hands back control to the host program, or starts an interactive session with a *program director*, 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 currently 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 direction commands, or variations (§3.3). 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 hardware development and deployment. As a technique, PhD is vendor-neutral and compiler-neutral, and the communication between the director and controller can be adapted to suit the program. For example, in our prototype we send direction commands via the network. PhD can be implemented in different ways: in our prototype we implemented it entirely 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 enabled and reconfigured at runtime: this consists of updating the code stored by the controller. In contrast, existing techniques 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.

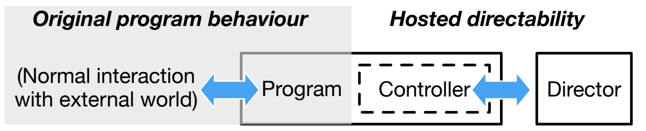
Original program behaviour
Hosted directability
(Normal interaction with external world)
Program
Controller
Director

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 outputs.

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 2011). The ideas described in this paper are not necessarily tied to the languages, compilers, FPGA or other equipment we used in our prototype. We make the following contributions:

- We describe how to translate familiar high-level idioms for debugging, profiling and monitoring (§3.1), which we call *direction commands*, into a low-level language for controlling program state at runtime (§3.2), for an example language §3.4.1.
- We relate the direction commands with our low-level language through a specification (§3.4) 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), where we measure the effect of directability on FPGA utilisation and performance.

2. 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 becoming 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; Chu et al. 2015)
- FPGAs are being deployed in large production environments, such as Microsoft's datacentres (Putnam 2014).

Hardware development has driven the development of formal methods to establish system correctness (Fix 2008), which enabled the development for methods for software (Ball et al. 2006; Godefroid et al. 2012). Unfortunately the verification 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.

2.1 Debugging(FPGA) != Debugging(CPU)

Debugging concepts from software do not correspond directly 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) and in many other work on HLS debugging (Goeders and Wilton 2014; Monson and Hutchings 2015). 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 consequences 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, §4.3). Moreover, the output of operations on previous lines might only be available *after* some clock cycles have elapsed. Depending on what the user wishes to do, they might prefer if the breakpoint is triggered after the elapsing of these cycles. Furthermore, 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 compiler 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.

2.2 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* (Sosič and Abramson 1997) 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 debugging. **Trace buffers** are the most popular technique for debugging FPGAs. It requires a programmer to identify signals of interest in the circuit at compile time, then an *embedded logic analyser* is synthesised that uses on-chip memory 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 window 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 (Bennetts and Osseyran 1991). 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 description 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 inflexible. Generating a hardware bitstream can take hours, and the added overhead costs for runtime-reconfigurable debuggability and monitoring features might not be affordable in some use-cases. Furthermore, these techniques cannot be used in production environments.

2.3 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). We outline the main cases below:

1. *Interface mismatch.* We need to understand whether a problem occurs because of a mismatch between one module and the rest of the circuit. Recall that behavioural simulation might not be applied to the whole design, and incorrect 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 testing other instances of the program. Other faults result from features of the hardware and transient environmental 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; Krishnaswamy et al. 2008).
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.

2.4 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 and Wilton 2014). Another idea consists of multiplexing all signals and choosing which to observe at debug time rather than at compile time (Hung and Wilton 2013), enabling "observation without recompilation". An additional idea is to buffer into a fast external memory (Panjkov et al. 2015).

Table 1 surveys closely related work, classifying it according to the features they provide. Related work is contrasted in more detail in §5. In Table 1 *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) 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.

2.5 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 shows the features supported by PhD. Initially it supported state inspection and update. Updating state enabled us to influence data-dependant control flow. Extending the controller to include branching and extension points enabled us to support more features, such as assertion checking, 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.

3. A language and model for program directing

In this section we describe $\mathfrak{D}$, a language of direction commands that we extracted by analysing the commands commonly given to profilers and debuggers such as gdb.[1] In Figure 1 we described one such command and how we code the high-level direction command as a program that will be executed by the controller.

3.1 Direction language $\mathfrak{D}$

Our direction language $\mathfrak{D}$ consists of the commands listed in Table 2. These commands can have three kinds of parameters: (i) symbols relating to the program, (such as variable or function names, or labels), (ii) relations over program variables, which we denote by the symbol $\langle B \rangle$, indicating the conditions on which statements might apply, and (iii) measures of resource, which we symbolise by $\langle \$ \rangle$, indicating a finite resource that is allocated for the execution of a command. 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 associated 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 $\mathfrak{D}\langle B \rangle$ be the set of possible conditions that can be used. We assume that at least $\mathtt{true} \in \mathfrak{D}\langle B \rangle$. We can also allow additional truth conditions, and in this model we will have $(V_1 = V_2) \in \mathfrak{D}\langle B \rangle$ for arbitrary V_1, V_2 ranging over program values or variables. For example, "$\mathtt{watch}\ v\ (v = 5)$" would instruct the controller to watch a variable v, and switch to interactive directing when $v = 5$.

$\mathfrak{D}\langle \$ \rangle \in \mathbb{N}$ describes the maximum quantity of some resource when carrying out a direction command. This value must be less than the compile-time allocation of the resource, to ensure the provision of sufficient resource for the command at runtime. This is needed to size the buffers used for tracing. For example, "$\mathtt{count\ reads}\ v\ \mathtt{true}\ 5000$" will count the number of reads of v, and break after 5000 reads have been made. This could be done to avoid overflow, or to capture some behaviour of interest. Similarly, "$\mathtt{trace\ start}\ v\ \mathtt{true}\ 500$" breaks after 500 instances of v have been recorded in the trace. The trace buffer must accommodate at least 500 entries.

[1] https://www.gnu.org/software/gdb/

System \ **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)	Network/Control	use leftover resource	embed at Source/HDL
(Sosič 1992) **Dynascope**	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓	■	■	■	■
(Goeders and Wilton 2014) **HLS-Scope**	✓	✓			✓	✓		✓						✓	C		S
(Calagar et al. 2014) **Inspect**	✓				✓	✓		✓				✓		✓	C		H
(Panjkov et al. 2015)	✓													✓	C		S
(Hung and Wilton 2014) **QuickTrace**	✓							✓				✓	✓	✓	N	✓	H
(Koch et al. 1998) **SLE/CADDY**	✓				✓	✓		✓				✓		✓	C		H
(Monson and Hutchings 2015)	✓	✓												✓	C		S
(Curreri et al. 2011)								■	✓	✓	✓			✓	C		H
(Camera et al. 2005) **BORPH**	✓	✓	✓		✓	✓	✓	✓							C		H
(see §2.5) **PhD**	✓	✓	✓	✓	✓		✓	✓	✓			✓	✓	■	C		■

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

Command	**Behaviour**
print X	Print the value of variable X from the source program.
break L $\langle B\rangle$	Activate a (conditional) breakpoint at the position of label L.
unbreak L	Deactivate a breakpoint.
backtrace $\langle\$\rangle$	Print the "function call stack".
watch X $\langle B\rangle$	Break when X is updated and satisfies a given condition.
unwatch X	Cancel the effect of the "watch" command.
count reads X $\langle B\rangle$ $\langle\$\rangle$ / writes X $\langle B\rangle$ $\langle\$\rangle$ / calls fname $\langle B\rangle$ $\langle\$\rangle$	Count the reads or writes to a variable X, or the calls to a function fname.
trace start X $\langle B\rangle$ $\langle\$\rangle$	Trace a variable, subject to a condition being satisfied, and up to trace some length.
trace stop X	Stop tracing a variable.
trace clear X	Clear a variable's trace buffer.
trace print X	Print the contents of a variable's trace buffer.
trace full X	Check if a variable's trace buffer is full.

Table 2. Directing commands making up language $\mathfrak{D}$. Note that count has similar subcommands to those of trace, to clear the counters, get their current value, and find out if a maximum value has been reached.

3.2 Controller

High-level direction commands such as those in Table 2 are ultimately translated into programs that run on a simple controller embedded in the program. We model the controller as a *CASP* machine (for "Counters, Arrays, and Stored Procedures", the constituents of the machine's memory).

CASP machines are very weak. They are more structurally complex than register machines (Shepherdson and Sturgis 1963) since they have separate memories for storing arrays and registers, but CASP machines are computationally much weaker than register machines, unable to encode partial computable functions. A limited form of memory indirection is permitted through a collection of arrays. The language lacks any means for defining recursive functions, or branching to arbitrary addresses. Any more complex computation must be done by the director; the controller simply provides a controlled access to the program's memory.

We describe the language of CASP programs in Figure 3. We rely on the following meta-variables and syntax categories: P programs, E expressions, I indices, U updatable values, V values, N numerals (corresponding to $\mathbb{Z}$), X variable identifiers, and R the array identifiers, where the names for variables and those for arrays are disjoint, $X \cap R = \emptyset$.

$@L : \{P\}$ 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:

$$\begin{array}{rcl@{\qquad}rcl}
P & ::= & E & E & ::= & V \\
 & | & U := E & & | & -V \\
 & | & \mathit{op}\ U & & | & V_1\ \mathit{op}\ V_2 \\
 & & \mathit{op} \in \{\texttt{inc}, \texttt{dec}\} & & & \mathit{op} \in \{=, <\} \\
 & | & P_1; P_2 & I & ::= & N \\
 & | & \texttt{if}\ E\ \texttt{then}\ P_1 & & | & X \\
 & & \quad\texttt{else}\ P_2 & U & ::= & X \\
 & | & \texttt{break} & & | & R[I] \\
 & | & \texttt{continue} & V & ::= & I \\
 & | & @L : \{P'\} & & | & R[I]
\end{array}$$

Figure 3. Syntax for CASP programs.

$$@L : \{\texttt{if}\ x = 1\ \texttt{then}\ @M : \{\texttt{break}\}\ \texttt{else}\ @M : \{\texttt{continue}\}\}$$

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.

3.3 Examples

CASP programs are to $\mathfrak{D}$ what microprograms are to an Instruction Set Architecture (Smith and van Leeuwen 1974). We give some examples of coding program direction commands 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. 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
```

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 $\langle B\rangle$" causes the program to break (for interactive guidance) when a variable's value satisfies some predicate. Let B be the CASP-level value to the $\mathfrak{D}$-level $\langle B\rangle$ parameter (we give such a function in §3.5.1). The code is similar to that in the profiling example above, except that B is user-provided.

```
if B then break else continue
```

Breakpoints. "break L $\langle B\rangle$" causes the program to break when it reaches a specific label, and if condition B is satisfied. 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 variables 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.

3.4 Directability ordering

In this section we define a relation $x \sqsubset x'$ to mean "x' is more directable than x", where x, x' are triples $(\mathcal{D}, \mathcal{C}, p)$ and $(\mathcal{D}', \mathcal{C}', p')$, each representing three interdependent parties: the *director*, *controller* and *program*. The user (or their agent) issues direction commands to the director, which interacts with the program's state via its agent, the controller, embedded in the program. We use this relation to give semantics 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 interdependence is revealed by the directing commands: for example, the director would not be able to execute trace X if the program did not have a variable called X, or if the controller had not been allocated a trace buffer.

In our notation, $\mathcal{D}$ 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). $\mathcal{C}$ is the controller's state, consisting of a (C, A, SP) machine, cf §3.2. p is a program. We will define an example language in §3.4.1 to aid our formalisation.

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) $\mathfrak{C} \subseteq \mathfrak{D}$ the direction commands (§3.1) supported by $(\mathcal{D}, \mathcal{C}, p)$, (ii) $c \in \mathfrak{D}$ the additional command supported by $(\mathcal{D}', \mathcal{C}', p')$, and $D_c \in (\mathcal{D}' \rightarrow \mathcal{D}')$ the semantics of this command. Note that $\mathfrak{C}$ denotes the set of direction commands that is supported *simultaneously* by $(\mathcal{D}, \mathcal{C}, p)$, i.e., these commands are allocated separate state.

Written out in full, we obtain this relation:

$$(\mathcal{D}, \mathcal{C}, p) \ \overset{c}{\sqsubset_{\mathfrak{C}}} \ (\mathcal{D}', \mathcal{C}', p') \ : \ D_c$$

In §3.5 we will instantiate such a relation by formalising commands from $\mathfrak{D}$ 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: D_c is written in an ML-like pseudocode.

Our formalisation is devised in away that avoids the mutual 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 $\breve{\mathcal{D}} = \mathcal{D}' \backslash \mathcal{D}$ and $\breve{\mathcal{C}} = \mathcal{C}' \backslash \mathcal{C}$. We say that D_c is *relevant* to $\mathcal{D}' \backslash \mathcal{D}$ if it only manipulates state or elements introduced in $\mathcal{D}'$ and $\mathcal{C}'$. Furthermore commands are *disjoint* if they introduce non-overlapping state. That is, for any two commands c_1 and c_2, for any prior states $\mathcal{C}$ and $\mathcal{D}$ their respective new states are disjoint: $(\mathcal{D}_1 \backslash \mathcal{D}) \cap (\mathcal{D}_2 \backslash \mathcal{D}) = \emptyset = (\mathcal{C}_1 \backslash \mathcal{C}) \cap (\mathcal{C}_2 \backslash \mathcal{C})$.

3.4.1 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) this language is computationally strong: recursive functions over the integers can be encoded. The language's simplicity enables the relation of program direction with CASP machines, while avoiding excessive formal complexity. Formalising transformations for realistic languages—even simple transformations (Schäfer et al. 2008)—is usually fraught with complex definitions, and we avoid that here.

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

$$\begin{array}{rcl}
p & ::= & \overline{vdecl}\ \overline{fdecl}\ \text{return}\ fname(\bar{e}) \\
vdecl & ::= & \tau\ X \\
fdecl & ::= & \tau\ fname(\bar{x})\{\bar{s}; \text{return}\ e\} \\
s & ::= & \text{skip} \\
 & \mid & X := e \\
 & \mid & \text{if}\ e\ \text{then}\ s; \bar{s} \\
 & \mid & \text{extend}\{L_1, \ldots, L_n\} \\
e & ::= & N \\
 & \mid & X \\
 & \mid & fname(\bar{e}) \\
 & \mid & e_1\ op\ e_2 \\
 & & op \in \{+, -, ==, <\}
\end{array}$$

In this language the only data type in τ is the integer type. The only unusual construct in this language is $\text{extend}\{L_1, \ldots, L_n\}$. This indicates an *extension point*, where control is passed to the CASP machine (§3.2). 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. Intuitively, the behaviour of $\text{extend}\{L_1, \ldots, L_n\}$ is as follows. If $n = 0$ then the command has no effect. Otherwise, the stored procedure associated with each L_i is called, in any order, and run to completion, noting the last instruction of each L_i. The last instruction of any CASP program is either '`continue`' or '`break`' (§3.2). If all L_i end in '`continue`', then the behaviour of $\text{extend}\{L_1, \ldots, L_n\}$ is to continue executing the next statement in the host program. Otherwise, if at least one L_i ends in '`break`', then the controller switches into interactive mode. In this mode, control 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 statements in the user's program are interspersed with 'extend': that is, if the user writes $s_0; \ldots; s_n$ then this is translated into $\text{extend}\{\}; s_0; \text{extend}\{\}; \ldots; \text{extend}\{\}; s_n; \text{extend}\{\}$. In the next section we populate these extension points with labels, to extend the directability of a program.

3.5 Semantics for $\mathfrak{D}$

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

3.5.1 Break

We start by formalising the meaning of the "break" command. Intuitively, this command adds a breakpoint to a program: 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 position where the breakpoint is to be placed. This extension is formalised by the premise $p <^1_L p'$, which means that p is identical to p' except for the label L occurring at some extension point. This is defined formally in §B along with related definitions, such as that of a *position* in the program.
- The label L must not appear in the original program. We write this as $L \notin p$ using an abbreviation defined in §B.
- The controller's state is extended to store the procedure associated with L. Furthermore, the breakpoint is activated by default. We use the abbreviated notation $\breve{\mathcal{C}} = \{SP[L \mapsto \texttt{break}]\}$ to indicate this extension, where SP is the stored-procedure memory in the CASP.

- The director's *state* is extended to encode whether the breakpoint is currently active or not. It is activated by default, thus: $\breve{\mathcal{D}} = \{(\text{«bp»}, L, 1)\}$, 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, D_c, involves activating the breakpoint unless already active.

In the definition of D_c below we use the notation $P \leadsto N$, which we use to mean that the director sent the CASP program P to the controller, and received the reply N. Thus "$@L : \{\texttt{break}\} \leadsto \ulcorner L \urcorner$" means that the director instructed the controller to store the program break at L, and that it expects to get $\ulcorner L \urcorner$ back (which is a code indicating where the program is stored, as formalised in §C.1).

In D_c we use the notation "$(\text{«bp»}, L, 0 \mapsto 1) :\in \mathcal{D}'$" to abbreviate $\{(\text{«bp»}, L, 1)\} \cup (\mathcal{D}' \backslash \{(\text{«bp»}, L, 0)\})$.

The formalisation of the "break" command follows. We can use a simplified notation since our commands will be both relevant to $\mathcal{D}' \backslash \mathcal{D}$ and disjoint:

$$\frac{L \notin p \qquad p <^1_L p'}{p \overset{\mathsf{break}\ L\ \langle B \rangle}{\sqsubset_{\mathfrak{C}}} p' \begin{cases} \breve{\mathcal{D}} = \{(\text{«bp»}, L, 1)\} \\ \breve{\mathcal{C}} = \{SP[L \mapsto \llbracket \mathsf{break}\ L\ \langle B \rangle \rrbracket_{SP}]\} \\ D_c = \lambda \mathcal{D}'.\ \text{if } (\text{«bp»}, L, 1) \in \mathcal{D}' \text{ then } \mathcal{D}' \\ \qquad\qquad \text{else} \\ \qquad\qquad\quad \llbracket \mathsf{break}\ L\ \langle B \rangle \rrbracket \leadsto \ulcorner L \urcorner; \\ \qquad\qquad\quad (\text{«bp»}, L, 0 \mapsto 1) :\in \mathcal{D}' \end{cases}}$$

where

$$\llbracket \mathsf{break}\ L\ \langle B \rangle \rrbracket_{SP} = \mathsf{conditional}\ \langle B \rangle\ \texttt{break}$$
$$\llbracket \mathsf{break}\ L\ \langle B \rangle \rrbracket = @L : \{\llbracket \mathsf{break}\ L\ \langle B \rangle \rrbracket_{SP}\}$$

We use $\llbracket \mathsf{break}\ L\ \langle B \rangle \rrbracket$ to denote the meaning of "$\mathsf{break}\ L\ \langle B \rangle$" to the director, as a CASP program. $\llbracket \mathsf{break}\ L\ \langle B \rangle \rrbracket_{SP}$ is the value used to initialise the stored program associated with L.

The meaning of $\langle B \rangle$ is translated for inclusion in the CASP program by the following function:

$$\mathsf{conditional}\ \langle B \rangle\ t\ = \begin{cases} t & \text{if } \langle B \rangle = \mathsf{true} \\ \texttt{if } I_1 \texttt{ == } I_2 \texttt{ then } t & \text{if } \langle B \rangle = (I_1 = I_2) \\ \quad \texttt{else continue} & \end{cases}$$

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:

$$\frac{(\mathsf{break}\ L\ \langle B \rangle) \in \mathfrak{C}}{p \overset{\mathsf{unbreak}\ L}{\sqsubset_{\mathfrak{C}}} p \begin{cases} \breve{\mathcal{D}} = \{\} \\ \breve{\mathcal{C}} = \{\} \\ D_c = \lambda \mathcal{D}'.\ \text{if } (\text{«bp»}, L, 0) \in \mathcal{D}' \text{ then } \mathcal{D}' \\ \qquad\qquad \text{else } \llbracket \mathsf{unbreak}\ L \rrbracket \leadsto \ulcorner L \urcorner; \\ \qquad\qquad\quad (\text{«bp»}, L, 1 \mapsto 0) :\in \mathcal{D}' \end{cases}}$$

where

$$\llbracket \mathsf{unbreak}\ L \rrbracket = @L : \{\texttt{continue}\}$$

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 ($X \in \mathrm{Var}_p$), and we need to have at least one extension point (through which we can send the print command).

$$\frac{(\mathsf{break}\ L\ \langle B \rangle) \in \mathfrak{C} \qquad X \in \mathrm{Var}_p}{p \overset{\mathsf{print}\ X}{\sqsubset_{\mathfrak{C}}} p \begin{cases} \breve{\mathcal{D}} = \{\} \\ \breve{\mathcal{C}} = \{\} \\ D_c = \lambda \mathcal{D}'.\ X \leadsto N; \\ \qquad\qquad \mathrm{print}(N) \end{cases}}$$

This time we didn't use a placement command to update the behaviour of the controller; we simply ran a query. $\mathrm{print}(N)$ is pseudocode that uses a print function in the director. Recall that we formalise directions in terms of CASP machines, and don't formalise the director's behaviour. $X \in \mathrm{Var}_p$ ensures that X is a variable in p. Var_p is defined in §B.

3.5.2 Trace

The most important command related to tracing is "trace start"; the other commands depend on it.

$$\frac{X \in \mathrm{Var}_p \qquad \forall L \in X_L.\ L \notin p \qquad \mathrm{Positions}_{p'}(X_L) = \mathrm{PostUpdate}_{p'}(X) \qquad p <_{X_L} p'}{p \overset{\mathsf{trace\ start}\ X\ \langle B \rangle\ \langle \$ \rangle}{\sqsubset_{\mathfrak{C}}} p'}$$

$$\begin{cases} \breve{\mathcal{D}} = \{(\text{«t»}, X, 1)\} \\ \breve{\mathcal{C}} = \left\{ \begin{array}{l} C[X_{\mathrm{i}}] = 0, C[X_{\mathrm{of}}] = 0, A[X_{\mathrm{a}}[\langle \$ \rangle]], \\ SP[L \mapsto \llbracket \mathsf{trace\ start}\ X\ \langle B \rangle\ \langle \$ \rangle \rrbracket_{SP}] \\ \quad \text{for each } L \in X_L \end{array} \right\} \\ D_c = \lambda \mathcal{D}'.\ \text{if } (\text{«t»}, X, 1) \in \mathcal{D}' \text{ then } \mathcal{D}' \\ \qquad\qquad \text{else} \\ \qquad\qquad\quad \text{for each } L \in X_L: \\ \qquad\qquad\qquad @L : \{\llbracket \mathsf{trace\ start}\ X\ \langle B \rangle\ \langle \$ \rangle \rrbracket_{SP}\} \\ \qquad\qquad\qquad\quad \leadsto \ulcorner L \urcorner; \\ \qquad\qquad\quad (\text{«t»}, X, 0 \mapsto 1) :\in \mathcal{D}' \end{cases}$$

$$\llbracket \mathsf{trace\ start}\ X\ \langle B \rangle\ \langle \$ \rangle \rrbracket_{SP} = \\ \mathsf{conditional}\ \langle B \rangle \left(\begin{array}{l} \texttt{if } X_{\mathrm{i}} < \langle \$ \rangle \texttt{ then} \\ \quad X_{\mathrm{a}}[X_{\mathrm{i}}] := X; \\ \quad \texttt{inc } X_{\mathrm{i}}; \\ \quad \texttt{continue} \\ \texttt{else} \\ \quad \texttt{inc } X_{\mathrm{of}}; \\ \quad \texttt{break} \end{array} \right)$$

Applying this rule depends on a set of labels X_L in p' that don't exist in p. p' 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 $\langle \$ \rangle$ elements. Each $L \in X_L$ labels the positions immediately after the variable has been updated.

$$\frac{(\textsf{trace start } X\ \langle B \rangle\ \langle \$ \rangle) \in \mathfrak{C} \qquad \mathrm{Positions}_p(X_L) = \mathrm{PostUpdate}_p(X)}{p \quad \overset{\textsf{trace stop } X}{\sqsubset_{\mathfrak{C}}} \quad p}$$

$$\begin{cases} \breve{\mathcal{D}} = \{\} \\ \breve{\mathcal{C}} = \{\} \\ D_c = \lambda \mathcal{D}'.\ \text{if } («\text{t}», X, 0) \in \mathcal{D}' \text{ then } \mathcal{D}' \\ \qquad\qquad\quad \text{else} \\ \qquad\qquad\qquad \text{for each } L \in X_L\text{:} \\ \qquad\qquad\qquad\quad @L : \{[\![\textsf{trace stop } X]\!]_{SP}\} \rightsquigarrow \ulcorner L \urcorner; \\ \qquad\qquad\quad («\text{t}», X, 1 \mapsto 0) :\in \mathcal{D}' \end{cases}$$

where $[\![\textsf{trace stop } X]\!]_{SP} = \texttt{continue}$.

The remaining commands are formalised in §A, supported by program-level predicates and functions defined in §B.

4. Implementation and Evaluation

We prototyped the ideas described in the previous section by extending network programs with program-hosted directability, and compiled them to run on an FPGA. We then evaluated the effect of these directability features on the program 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 machine instructions.

4.1 Prototype

In our prototype we manually transformed programs to include the controller. This transformation was straightforward: we wrote the controller (implementing a CASP machine) and added extension points to the program (consisting of calls to the controller to execute a stored procedure).

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

1. It allows us to test remote directing over standard network equipment. In our survey of tools and techniques (Table 1) only Dynascope has network access, but it does not work for FPGAs.
2. It allows us to use industrial high-precision network measuring equipment to see the effects on the program hosting a controller.

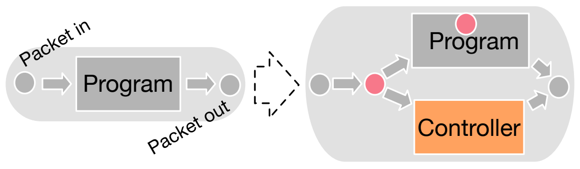

Figure 4. Transformation of the program to include a controller. Normal packets are handled as normal, but *direction packets* are passed to the controller. Pink dots represent *extension points*, one of which is added within the control flow of the original program in this illustration.

Use-cases. As programs we used implementations of DNS and Memcached that we had written previously to run on FPGAs, as part of earlier research. DNS (Domain Name System) is a ubiquitous name-resolution system used on private and public packet-switched networks such as the Internet (Mockapetris 1987). This implementation was around 700 lines of C$^{\#}$. Memcached (Fitzpatrick 2004) is a well-known, in-memory key/value store that caches read results in memory to quickly respond to queries. The protocol uses a number of basic commands such as GET (retrieve a value associated 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 supporting GET/SET/DELETE using the binary protocol over UDP and supporting 6 byte keys and 8 byte values. This implementation was almost 900 lines of C$^{\#}$.

Method. We transformed the DNS and Memcached implementations in two ways: (i) adding code to check whether a received packet is a *direction packet* intended for the controller (see Figure 4), in which case the controller (and not the original program) processes the packet; (ii) adding an extension 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 access and change at runtime. The code for each value of the enumerated type is used to refer to the program value, to 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 consists of (i) code to be executed by the controller, or (ii) status replies from the controller to the director. It enables us to remotely direct a running program, similar to gdb's 'remote serial protocol'.[2] Our design uses a simple direction language, and works for instances of a program that run both

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

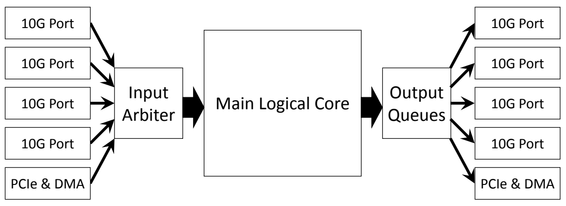

Figure 5. NetFPGA Reference Pipeline. Each of the implementations in our prototype consists of a separate Main Logical Core.

as software and hardware (on FPGAs), whereas gdb requires special backends for each architecture.

Controller. Our controller follows the description of CASP machines very closely (§3.2). It has two features. First, memory is organised into *Counters*, *Arrays* and *Stored procedures*. 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 function that interprets the language of CASP machines. This is used to branch to stored procedures when their corresponding extension points are reached.

Tools and equipment. We wrote our programs in C$^{\#}$, and used the Kiwi *high-level synthesis* (HLS) system (Singh and Greaves 2008) 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] and compiled to run on the NetFPGA SUME board (Zilberman et al. 2014), 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 architecture, illustrated in Figure 5, 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 this module, the packets are passed to the physical interfaces and are transmitted.

4.2 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 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 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 program'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 NetFPGA 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 we compare DNS extended with one extension point, against DNS extended with an *embedded logic analyser* (ELA). An ELA is a standard technique in hardware development, and consists of a circuit that passively monitors the program, creating a trace of the program's execution. 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 minimal, making PhD a feasible solution. We note that in the use-cases detailed below the FPGA resources were never exhausted, 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.

Test setup. We use a host running Ubuntu server 14.04LTS, kernel version 4.4.0-45-generic. The host hardware is a single 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 independent measurement of client & server latency. For the throughput tests, OSNT (Antichi et al. 2014) is used to control the rate of the generated requests.

[3] `http://netfpga.org/`

Artefact	**Utilisation (%)**		**Performance**		
	Logic	Flip-flops	Duration (#cycles)	Latency (μs)	Queries-per-sec (KQPS)
DNS	100.00	100.00	57	1.85	1176
+R	103.40	102.76	-	1.85	1176
+W	115.05	106.04	-	1.84	1176
+I	109.79	106.12	-	1.84	1176
Memcached	100.00	100.00	64	2.03	952
+R	99.17	100.29	-	2.03	952
+W	99.80	100.74	-	2.04	952
+I	100.63	100.69	-	2.03	952

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

Artefact	**Utilisation (%)**		**Performance**		
	Logic	Flip-flops	Duration (#cycles)	Latency (μs)	Queries-per-sec (KQPS)
DNS+ELA	99.74	100.40	57	1.83	1176
DNS+2e	234.61	151.06	57	1.86	1176
(Count)	234.46	151.81	62	1.94	1064
(Trace)	218.30	151.84	70	1.99	1010

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 variable tracing. Latency is indicated at the 99^{th} percentile.

5. Related work

Portable debugging, program directing, and debugging languages. We were inspired by previous work on portable debugging (Ramsey and Hanson 1992; Hood 1996; Hanson and Raghavachari 1996) and program directing (Sosič 1995). That work usually makes the assumption that software is compiled to run on a general-purpose CPU however, whereas we also target reconfigurable hardware. The study of languages for debugging is a decades old subject (Balzer 1969) and includes sophisticated languages for high-level querying of programs (Johnson 1977; Golan and Hanson 1993; Winterbottom 1994). Compared to that work, our work separates more starkly between the role (and language) of the director, and that of the controller, which have separate languages. This separation guided our modelling in §3.

Dynascope. Dynascope (Sosič 1992) provides an extremely fine grained *execution stream* of events–at the level of machine instructions–providing a complete description of a program's runtime behaviour. We provide selective (and programmable) visibility by default, in the interest of performance. A more detailed stream could be produced if wished. Dynascope is generative by default (since you need to set filters 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) who adapt a debugging paradigm first developed for network-on-chip devices (NoC). They provide *transaction-level* granularity, consisting of domain-specific (high-level) events–less fine grained than Dynascope's execution stream. Compared to our work they offer a less flexible interface, but they take a more detailed account of different sorts of interconnects between components–formats such as LocalLink and PLB.

Many other systems inject code to emit and observe events to aid with debugging of complex designs, such as "System on Chip" designs (Lee and Lysecky 2015). SeaCucumber (Tripp et al. 2002) was the first to support source-level (HLS) debugging, both during behavioural simulation and during hardware execution. Monson and Hutchings (2015) 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 interest. We took the approach of Monson and Hutchings (2015), 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) 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) appears to provide the best support for debugging. In a contribution to that system, Hung and Wilton (2014) take a different approach to us: they use a two-step incremental compilation, the second step of which compiles the monitoring 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 developed to verify designs prior to synthesis (Foster 2008; Fix 2008) . The difficulty of checking hardware entirely prior to synthesis has led to research into the inclusion of runtime monitors in hardware (Todman et al. 2015). Curreri et al. (2011) describe a system that translates source-level assertions into monitors. This technique is also used to detect hangs and possible timing overruns. We currently do not support such monitoring, since it requires a source-level notion of time that we currently do not provide, and "watchdog threads" that we currently do not include.

Finally, Camera et al. (2005) described a hardware OS, the Berkeley Operating system for ReProgrammable Hardware (BORPH), on which user programs were run. Their *stitcher* extend the user program to support debugging, providing 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 formalise. Perhaps as a consequence of this there has been little work on formalising and verifying debuggers, and usually entirely theoretical (Zhu 2001). Kishon et al. (1991) describe transformations over functional programs to include monitoring behaviour. Compared to this work, our transformations are not based on continuations (and we deliberately avoid needing first-order functions to avoid departing too much from the conventional hardware programming mindset). We use an approach based on operational semantics, similar to that used by da Silva (1992), but different in several ways: (i) we support a different set of directing commands, (ii) we don't insert commands into the bytecode for the debugger to keep track of separations between subexpressions, and to keep track of the path through the expression (program), (iii) da Silva (1992), introduces a language for specifying debuggers whereas we introduce an operational language for inspecting and changing the program's runtime state, (iv) we do not consider the equivalence between debuggers. Sansom and Peyton Jones (1997) 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 (Faddegon and Chitil 2015). All the techniques described above rely on program transformation. They all use functional languages 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 debugging code into the program all these techniques can (retrospectively) be seen as special-cases of aspect-oriented programming, 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 linchpin paradigm for tracing and monitoring (Avgustinov et al. 2006; Hamlen and Jones 2008) but advice can be arbitrary functions, and as a consequence they might have undesirable effects on runtime (Avgustinov et al. 2007). This is an important characteristic to our work, where we wanted to reduce the power of added code. Djoko Djoko et al. (2012) have categorised advice based on the degree of influence they can have on the observable behaviour of a program; and Dantas and Walker (2006) characterise less intrusive advice in their work on "harmless advice".

6. 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 (Sosič 1992). 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 exploited (Abadi et al. 2005). 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 (Pauli 2016). Dynascope was used to diagnose errors in the Dynascope compiler itself (Sosič 1992, §3.1), but bugs could render such a task impossible, for reasons similar to the hereditary potential of vulnerabilities in compilers (Thompson 1984). 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 *observer effect* (Mytkowicz et al. 2008) that monitoring code has on the monitored code. This is known to affect the visibility of timing bugs in software (Neville-Neil 2014), but even "invisible" such bugs can leak important information to an adversary (Cock et al. 2014).

The overhead due to monitoring is a very important consideration when evaluating system measurements, and monitoring systems try to do their utmost to reduce it (Gregg and Mauro 2011; Anderson et al. 2014). A key weakness of our prototype is that it does not use low-level (RTL-level) techniques 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 involve organising the director to operate in a different clock domain if possible, following McKechnie et al. (2009). 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 knowledge 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 research and innovation programme 2014-2018 under the SSICLOPS (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 Computer 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.

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.

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, O. de Moor, N. Ongkingco, D. Sereni, G. Sittampalam, J. Tibble, and M. Verbaere. Aspects for Trace Monitoring. In *Proceedings of the First Combined International Conference on Formal Approaches to Software Testing and Runtime Verification*, 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.

P. 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.

T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough 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.

R. M. Balzer. EXDAMS: EXtendable Debugging and Monitoring 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.

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.

N. Calagar, S. D. Brown, and J. H. Anderson. Source-level debugging 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 Debugging Environment for Reprogrammble Hardware Systems. In *Proceedings of the Sixth International Symposium on Automated 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.

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 *Proceedings 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.

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://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 *Conference 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/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://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://doi.acm.org/10.1145/2737924.2737985.

B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 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 Realities*, pages 5–10. Springer Berlin Heidelberg, Berlin, Heidelberg, 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.

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 Debugging 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://doi.acm.org/10.1145/1375696.1375699.

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

R. Hood. The P2D2 Project: Building a Portable Distributed Debugger. In *Proceedings of the SIGMETRICS Symposium on Parallel 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/10.1145/238020.238058.

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.

E. Hung and S. J. E. Wilton. Accelerating FPGA Debug: Increasing 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.

A. Kishon, P. Hudak, and C. Consel. Monitoring Semantics: A Formal Framework for Specifying, Implementing, and Reasoning 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.

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.

S. Krishnaswamy, I. L. Markov, and J. P. Hayes. On the Role of Timing Masking in Reliable Logic Circuit Design. In *Proceedings 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://doi.acm.org/10.1145/2717310.

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

R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

S. Mittal. A survey of techniques for improving energy efficiency in embedded computing systems. *International Journal of Computer 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 Transformations to Improve High-Level Synthesis Debug and Validation on FPGAs. In *Proceedings of the 2015 ACM/SIGDA International 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://doi.acm.org/10.1145/2684746.2689087.

T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan. Observer 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. Hybrid FPGA debug approach. In *2015 25th International Conference 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/.

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 Design*, ICCAD '95, pages 295–301, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7213-7. URL 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 Symposium on High-Performance Chips (HotChips 2014)*. IEEE, August 2014.

A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, 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 accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International 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 *Proceedings of the ACM SIGPLAN 1992 Conference on Programming 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.

P. M. Sansom and S. L. Peyton Jones. Formally Based Profiling for Higher-order Functional Languages. *ACM Trans. Program. 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.

A. Sari, D. Agiakatsikas, and M. Psarakis. A Soft Error Vulnerability 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/10.1145/2554688.2554767.

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://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.

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 Systems. 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://doi.acm.org/10.1145/512529.512550.

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

R. Sosič. 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.

R. Sosič. 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.

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.

T. Todman, S. Stilkerich, and W. Luk. In-circuit Temporal Monitors for Runtime Verification of Reconfigurable Designs. In *Proceedings 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.

D. Wang, N. E. Jerger, and J. G. Steffan. DART: A Programmable 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/10.1145/1999946.1999970.

P. Winterbottom. ACID: A Debugger Built From A Language. In *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. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, 2014.

A. More program direction commands

Program-level predicates and functions mentioned in these definitions are given in §B.

A.1 Tracing

We continue from §3.5.2, where the symbols we use here are introduced (such as X_{i} and X_{of}).

$$\frac{(\textsf{trace start } X\ \langle B\rangle\ \langle\$\rangle) \in \mathfrak{C}}{p \quad \overset{\textsf{trace clear } X}{\sqsubset_{\mathfrak{C}}} \quad p}$$

$$\left\{\begin{array}{l}
\breve{\mathcal{D}} = \{\} \\
\breve{\mathcal{C}} = \{\} \\
D_c = \lambda\mathcal{D}'.\ \llbracket \textsf{trace clear } X \rrbracket \rightsquigarrow 0; \\
\qquad\qquad \mathcal{D}'
\end{array}\right.$$

where

$$\begin{array}{l}
\llbracket \textsf{trace clear } X \rrbracket = \\
\qquad X_{\mathrm{i}} := 0; \\
\qquad X_{\mathrm{of}} := 0
\end{array}$$

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

$$\frac{(\textsf{trace start } X\ \langle B\rangle\ \langle\$\rangle) \in \mathfrak{C}}{p \quad \overset{\textsf{trace full } X}{\sqsubset_{\mathfrak{C}}} \quad p}$$

$$\left\{\begin{array}{l}
\breve{\mathcal{D}} = \{\} \\
\breve{\mathcal{C}} = \{\} \\
D_c = \lambda\mathcal{D}'.\ \llbracket \textsf{trace full } X \rrbracket \rightsquigarrow N; \\
\qquad\qquad \mathrm{print}(N); \\
\qquad\qquad \mathcal{D}'
\end{array}\right.$$

where

$$\llbracket \textsf{trace full } X \rrbracket = X_{\mathrm{of}}$$

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.

$$\frac{(\textsf{trace start } X\ \langle B\rangle\ \langle\$\rangle) \in \mathfrak{C}}{p \quad \overset{\textsf{trace print } X}{\sqsubset_{\mathfrak{C}}} \quad p}$$

$$\left\{\begin{array}{l}
\breve{\mathcal{D}} = \{\} \\
\breve{\mathcal{C}} = \{\} \\
D_c = \lambda\mathcal{D}'.\ X_{\mathrm{i}} \rightsquigarrow cur\_idx; \\
\qquad\qquad \text{for } i = cur\_idx \text{ to } 0\text{:} \\
\qquad\qquad\quad X_{\mathrm{a}}[i] \rightsquigarrow N; \\
\qquad\qquad\quad \mathrm{print}(N); \\
\qquad\qquad \mathcal{D}'
\end{array}\right.$$

A.2 Watching

This bears some similarity to the rule to start tracing (in §3.5.2) 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.

$$\frac{X \in \mathrm{Var}_p \qquad \forall L \in X_L.\ L \notin p \qquad \mathrm{Positions}_{p'}(X_L) = \mathrm{PostUpdate}_{p'}(X) \qquad p <_{X_L} p'}{p \quad \overset{\textsf{watch } X\ \langle B\rangle}{\sqsubset_{\mathfrak{C}}} \quad p'}$$

$$\left\{\begin{array}{l}
\breve{\mathcal{D}} = \{(\text{«w»}, X, 1)\} \\
\breve{\mathcal{C}} = \left\{\begin{array}{l} SP[L \mapsto \llbracket \textsf{watch } X\ \langle B\rangle \rrbracket_{SP}] \\ \quad \text{for each } L \in X_L \end{array}\right\} \\
D_c = \lambda\mathcal{D}'.\ \text{if } (\text{«w»}, X, 1) \in \mathcal{D}' \text{ then } \mathcal{D}' \\
\qquad\qquad \text{else} \\
\qquad\qquad\quad \text{for each } L \in X_L\text{:} \\
\qquad\qquad\qquad @L : \{\llbracket \textsf{watch } X\ \langle B\rangle \rrbracket_{SP}\} \rightsquigarrow \ulcorner L \urcorner; \\
\qquad\qquad\quad (\text{«w»}, X, 0 \mapsto 1) :\in \mathcal{D}'
\end{array}\right.$$

where

$$\llbracket \textsf{watch } X\ \langle B\rangle \rrbracket_{SP} = \textsf{conditional } \langle B\rangle\ \texttt{break}$$

Unwatching a variable follows a pattern we've encountered before, when stopping tracing for instance.

$$\frac{(\textsf{watch } X\ \langle B\rangle) \in \mathfrak{C} \qquad \mathrm{Positions}_p(X_L) = \mathrm{PostUpdate}_p(X)}{p \quad \overset{\textsf{unwatch } X}{\sqsubset_{\mathfrak{C}}} \quad p}$$

$$\left\{\begin{array}{l}
\breve{\mathcal{D}} = \{\} \\
\breve{\mathcal{C}} = \{\} \\
D_c = \lambda\mathcal{D}'.\ \text{if } (\text{«w»}, X, 0) \in \mathcal{D}' \text{ then } \mathcal{D}' \\
\qquad\qquad \text{else} \\
\qquad\qquad\quad \text{for each } L \in X_L\text{:} \\
\qquad\qquad\qquad @L : \{\llbracket \textsf{unwatch } X \rrbracket_{SP}\} \rightsquigarrow \ulcorner L \urcorner; \\
\qquad\qquad\quad (\text{«w»}, X, 1 \mapsto 0) :\in \mathcal{D}'
\end{array}\right.$$

where

$$\llbracket \textsf{unwatch } X \rrbracket_{SP} = \texttt{continue}$$

A.3 Profiling

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 immediately after each line that updates that variable–this is similar to tracing (§3.5.2) and watching (§A.2). To count variable reads we position after lines in which those variables appear. To count function calls we position just before the first line in the function.

$$\frac{X \in \mathrm{Var}_p \qquad \forall L \in X_L.\ L \notin p \qquad \mathrm{Positions}_{p'}(X_L) = \mathrm{PostUpdate}_{p'}(X) \qquad p <_{X_L} p'}{p \quad \overset{\textsf{count write start } X\ \langle B\rangle\ \langle\$\rangle}{\sqsubset_{\mathfrak{C}}} \quad p'}$$

$$\begin{cases} \breve{\mathcal{D}} = \{(\text{«c»}, X, 1)\} \\ \breve{\mathcal{C}} = \left\{ \begin{array}{l} C[X_{\text{count}}] = 0, C[X_{\text{of}}] = 0, A[X_{\text{a}}[\langle\$\rangle]], \\ SP[L \mapsto [\![\textsf{count write start } X \ \langle B\rangle \ \langle\$\rangle]\!]_{SP}] \\ \quad \text{for each } L \in X_L \end{array} \right\} \\ D_c = \lambda\mathcal{D}'.\ \text{if } (\text{«c»}, X, 1) \in \mathcal{D}' \text{ then } \mathcal{D}' \\ \qquad\qquad \text{else} \\ \qquad\qquad\quad \text{for each } L \in X_L\text{:} \\ \qquad\qquad\qquad @L : \{[\![\textsf{count write start } X \ \langle B\rangle \ \langle\$\rangle]\!]_{SP}\} \\ \qquad\qquad\qquad\qquad \rightsquigarrow \ulcorner L \urcorner; \\ \qquad\qquad\quad (\text{«c»}, X, 0 \mapsto 1) :\in \mathcal{D}' \end{cases}$$

where

$$\begin{array}{l} [\![\textsf{count write start } X \ \langle B\rangle \ \langle\$\rangle]\!]_{SP} = \\ \quad \textsf{conditional } \langle B\rangle \\ \quad \left(\begin{array}{l} \texttt{if } X_{\text{count}} < \langle\$\rangle \texttt{ then} \\ \qquad \texttt{inc } X_{\text{count}}; \\ \qquad \texttt{continue} \\ \texttt{else} \\ \qquad \texttt{inc } X_{\text{of}}; \\ \qquad \texttt{break} \end{array} \right) \end{array}$$

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).

B. Program analyses and transformations

A position of statement s in program p is a vector $\pi \in \mathbb{N}^d$ where $1 < d < \omega$. 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

$$p = \overline{vdecl}\ \overline{fdecl} \text{ return } \mathit{fname}(\bar{e})$$

Then

$$\Pi_p \stackrel{def}{=} \bigcup_{0 \le i \le n} (\{i\} \times \Pi_{fdecl_i})$$

where n is $|\overline{fdecl}|$, the number of function declarations in p.

Let $\mathit{fdecl} = \tau\ \mathit{fname}(\bar{x})\{\bar{s}; \text{return } e\}$ and $\bar{s} = s_0; \ldots; s_n$. Then

$$\Pi_{fdecl} \stackrel{def}{=} \bigcup_{0 \le i \le n} (\{i\} \times \Pi_{s_i}) \cup \{n+1\}$$

Finally,

$$\Pi_s \stackrel{def}{=} \begin{cases} \bigcup_{0 \le i \le n} (\{i\} \times \Pi_{r_i}) \cup \{n+1\} \\ \qquad \text{if } s = \text{if } e \text{ then } r_0; \ldots; r_n \\ \emptyset \quad \text{otherwise} \end{cases}$$

Next we define a function Pos_p maps a position $\pi \in \Pi_p$ in p to the corresponding statement s, in symbols: $\text{Pos}_p(\pi) = s$.

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

$$p = \overline{vdecl}\ \overline{fdecl} \text{ return } \mathit{fname}(\bar{e})$$

Then we define

$$\text{Pos}_p((i, \pi)) \stackrel{def}{=} \text{FPos}_{fdecl_i}(\pi)$$

Let $\mathit{fdecl} = \tau\ \mathit{fname}(\bar{x})\{\bar{s}; \text{return } e\}$ and $\bar{s} = s_0; \ldots; s_n$. Then

$$\text{FPos}_{fdecl}((i, \pi)) \stackrel{def}{=} \text{SPos}_{s_i}(\pi)$$

Finally,

$$\text{SPos}_s(\pi) \stackrel{def}{=} \begin{cases} \text{SPos}_{r_i}(\pi') & \text{if } \pi = (i, \pi') \\ & \text{and } s = \text{if } e \text{ then } r_0; \ldots; r_n \\ s & \text{if } \pi = \emptyset \end{cases}$$

Definition B.3 (Program p contains label L)**.** We write $L \in p$ iff $\exists \pi, S.\ \text{Pos}_p(\pi) = \textsf{extend}(S) \ \wedge \ L \in S$.

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.

$$\begin{array}{l} \forall \pi_1, \pi_2, S_1, S_2.\ (\text{Pos}_p(\pi_1) = \textsf{extend}(S_1) \ \wedge \ L \in S_1) \\ \quad \wedge (\text{Pos}_p(\pi_2) = \textsf{extend}(S_2) \ \wedge \ L \in S_2) \\ \quad \longrightarrow (S_1 = S_2 \wedge \pi_1 = \pi_2) \end{array}$$

We now formalise the predicate that programs p and p' are identical save for the addition of L in one of the extension points.

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

$$\begin{array}{l} p <^1_L p' \stackrel{def}{=} \forall \pi, s, S. \\ \quad (\text{Pos}_p(\pi) = s \longrightarrow \\ \qquad ((s \neq \textsf{extend}(S) \longrightarrow \text{Pos}_{p'}(\pi) = s) \\ \qquad \wedge (s = \textsf{extend}(S) \longrightarrow \\ \qquad\quad \text{Pos}_{p'}(\pi) = \textsf{extend}(S') \wedge (S' = S \vee S' \backslash S \subseteq \{L\})))) \\ \quad \wedge (\text{Pos}_{p'}(\pi) = s \longrightarrow \\ \qquad ((s \neq \textsf{extend}(S) \longrightarrow \text{Pos}_p(\pi) = s) \\ \qquad \wedge (s = \textsf{extend}(S') \longrightarrow \\ \qquad\quad \text{Pos}_p(\pi) = \textsf{extend}(S) \wedge (S' = S \vee S' \backslash S \subseteq \{L\})))) \end{array}$$

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 $S = \{L_0, \ldots, L_n\}$ is a set of labels. Then,

$$p <_S p' \stackrel{def}{=} \exists p_1, \ldots, p_{n-1}.\ p <^1_{L_0} p_1 <^1_{L_1} p_2 <^1_{L_2} \cdots <^1_{L_n} p'$$

Definition B.6 (Set of variables in program p)**.** Let $p = \overline{vdecl}\ \overline{fdecl}$ return $fname(\bar{e})$ and $\overline{vdecl} = vdecl_0 \ldots vdecl_n$. Then

$$\mathrm{Var}_p \overset{def}{=} \bigcup_{0\leq i\leq n} \{X_i \mid vdecl_i = \tau_i\ X_i\}$$

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

$$\mathrm{PostUpdate}_p(X) \overset{def}{=} \{\pi + 1 \mid \exists E.\ \mathrm{Pos}_p(\pi) = (X := E)\}$$

where if $\pi = (m_0, \ldots, m_n)$ then $\pi{+}1 = (m_0, \ldots, m_n{+}1)$.

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

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

$$\mathrm{Positions}_p(L) \overset{def}{=} \{\pi \mid \exists S.\ \mathrm{Pos}_p(\pi) = \mathsf{extend}(S)\ \wedge\ L \in S\}$$

We lift this to work on sets of labels and overload the notation: if S_L is a set of labels then

$$\mathrm{Positions}_p(S_L) \overset{def}{=} \bigcup_{L\in S_L} (\mathrm{Positions}_p(L))$$

C. Language semantics

C.1 CASP machines

The specification of CASP machines was given in §3.2.

The machine's *configuration* consists of the triple $(\mathcal{S}, ia, P)$, where $ia \in \{\circ, \bullet\}$ indicates whether the machine is operating in batch or interactive modes respectively, and $\mathcal{S} = (C, A, SP)$ indicates the machine's memory: $C \in (X \rightharpoonup \mathbb{Z})$ are the counters, $A \in (R \rightharpoonup \mathbb{Z} \rightharpoonup \mathbb{Z})$ the arrays, and $SP \in (L \rightharpoonup P)$ the stored procedures.

The machine's big-step operational semantics are described using the notation $L \vdash (\mathcal{S}, ia, P) \Longrightarrow (\mathcal{S}', ia', N)$ to indicate that the machine operates in the context of a specific label L to evaluate program P into numeral N, and possibly updating the other components of its configuration. We use the notation $L \vdash (\mathcal{S}, ia, P) \Downarrow N$ to abbreviate $L \vdash (\mathcal{S}, ia, P) \Longrightarrow (\mathcal{S}, ia, N)$ when $\mathcal{S}$ and ia are unaffected by the evaluation.

The semantic rules are given in Figure 6. We use the notation $\ulcorner L \urcorner$ 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.

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 respectively.

Note also that placement ($@L : \{P\}$) is only allowed during interactive mode (i.e., $ia = \bullet$) since we do not want the controller to normally run code that updates other extension points' code.

C.2 Example language

The syntax for the example language was given in §3.4.1. In this language we have a valuation function for identifiers $R \in (X \rightharpoonup \mathbb{Z})$, and a configuration for the reduction semantics consists of a triple $\langle R, \bar{c}, \bar{c}\rangle$ where c is either a statement, a 'return' statement, or an expression: $c ::= s \mid \mathrm{return}\ e \mid e$. $\bar{c}$ denotes a sequence of 1 or more such c: $c_0, \ldots, c_n$ where $n > 0$. We define the concatenation operator : such that if $\bar{d} = d_0, \ldots, d_m$ where $m > 0$, then $\bar{c} : \bar{d} = c_0, \ldots, c_n, d_0, \ldots, d_m$.

Intuitively, $\langle R, \bar{c}_1, \bar{c}_2\rangle$ describes a configuration where $\bar{c}_1$ is to be evaluated "now", and $\bar{c}_2$ is to be executed after $\bar{c}_1$ has been evaluated. The evaluation rules are given in Figure 7.

To reduce clutter in the rules, we assume two pieces implicit state, that we avoid threading around the configurations. The first is a mapping F from function names to their bodies, and the second is the state of the CASP machine $\mathcal{S}$, which includes the valuation R that models the store for program variables.

We use the following evaluation context for syntactic objects in c (statements, 'return' statements, and expressions):

$$\begin{array}{rcl} E & ::= & [] \\ & | & X := E \\ & | & \text{if } E \text{ then } s; \bar{s} \\ & | & fname(N_0, \ldots, N_n, E, e_0 \ldots, e_m) \\ & | & E\ op\ e \\ & | & N\ op\ E \\ & & op \in \{+, -, ==, <\} \end{array}$$

We handle 'extend' by expanding it to a sequence of singleton extends, and invoking the rule in Figure 8.

Rules for $\curvearrowright$ model the interactive mode between the director and the controller. In these semantics we only see evaluation from the point of view of the program: the program's evaluation $\longrightarrow$ yields to the controller $\Longrightarrow$ at extension 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 $\curvearrowright$ (obtaining a command from the director) and $\Longrightarrow$ (executing it). Note that the rules for $\curvearrowright$ are side-effecting, and are formalised as such similar to streams as used by Milner et al. (1990). In the rules for $\curvearrowright$, P is a command obtained from the director, and N is a result sent back to the director.

$$L \vdash (\mathcal{S}, ia, \texttt{continue}) \Longrightarrow (\mathcal{S}, \circ, \ulcorner L \urcorner) \qquad L \vdash (\mathcal{S}, ia, \texttt{break}) \Longrightarrow (\mathcal{S}, \bullet, \ulcorner L \urcorner) \qquad \frac{L \vdash (\mathcal{S}, ia, V) \Downarrow N}{L \vdash (\mathcal{S}, ia, -V) \Downarrow -N}$$

$$\frac{L \vdash (\mathcal{S}, ia, V_1) \Downarrow N_1 \qquad L \vdash (\mathcal{S}, ia, V_2) \Downarrow N_2}{L \vdash (\mathcal{S}, ia, V_1 \ op\ V_2) \Longrightarrow (\mathcal{S}, ia, N)} \quad N = \begin{cases} 1 & (op = (=) \wedge N_1 = N_2) \vee (op = (<) \wedge N_1 < N_2) \\ -1 & \text{o/w} \end{cases}$$

$$\frac{L \vdash (\mathcal{S}, ia, U) \Downarrow N}{L \vdash (\mathcal{S}, ia, op\ U) \Longrightarrow (\mathcal{S}[U \mapsto M], ia, M)} \quad M = \begin{cases} N+1 & op = \texttt{inc} \\ N-1 & op = \texttt{dec} \end{cases} \qquad \frac{\mathcal{S}(C, X) = N}{L \vdash (\mathcal{S}, ia, X) \Downarrow N}$$

$$\frac{L \vdash (\mathcal{S}, ia, E) \Downarrow N}{L \vdash (\mathcal{S}, ia, U := E) \Longrightarrow (\mathcal{S}[U \mapsto N], ia, N)} \qquad \frac{L \vdash (\mathcal{S}, ia, I) \Downarrow N \qquad \mathcal{S}(R[A], N) = M}{L \vdash (\mathcal{S}, ia, A[I]) \Downarrow M}$$

$$\frac{L \vdash (\mathcal{S}, ia, E) \Downarrow N \qquad L \vdash (\mathcal{S}, ia, P) \Longrightarrow (\mathcal{S}', ia', M)}{L \vdash (\mathcal{S}, ia, \texttt{if}\ E\ \texttt{then}\ P_1\ \texttt{else}\ P_2) \Longrightarrow (\mathcal{S}', ia', M)} \quad P = \begin{cases} P_1 & N = 1 \\ P_2 & N = -1 \end{cases}$$

$$\frac{L \vdash (\mathcal{S}, ia, P_1) \Longrightarrow (\mathcal{S}', ia', N) \qquad ia \neq ia'}{L \vdash (\mathcal{S}, ia, P_1; P_2) \Longrightarrow (\mathcal{S}', ia', N)}$$

$$\frac{L \vdash (\mathcal{S}, ia, P_1) \Longrightarrow (\mathcal{S}', ia', M) \qquad ia = ia' \qquad L \vdash (\mathcal{S}', ia', P_2) \Longrightarrow (\mathcal{S}'', ia'', N)}{L \vdash (\mathcal{S}, ia, P_1; P_2) \Longrightarrow (\mathcal{S}'', ia'', N)}$$

$$L \vdash (\mathcal{S}, \bullet, @L' : \{P\}) \Longrightarrow (\mathcal{S}[SP[L'] \mapsto P], \bullet, \ulcorner L' \urcorner) \qquad L \vdash (\mathcal{S}, ia, N) \Downarrow N$$

Figure 6. Dynamic semantics for CASP instructions (§3.2)

$$\langle R, \text{skip}, s; \mathit{rest} \rangle \longrightarrow \langle R, s, \mathit{rest} \rangle \qquad \langle R, N, s; \mathit{rest} \rangle \longrightarrow \langle R, s, \mathit{rest} \rangle \qquad \langle R, s; \bar{r}, \mathit{rest} \rangle \longrightarrow \langle R, s, \bar{r} : \mathit{rest} \rangle$$

$$\langle R, X := N, \mathit{rest} \rangle \longrightarrow \langle R[X \mapsto N], \mathit{skip}, \mathit{rest} \rangle \qquad \langle R, X, \mathit{rest} \rangle \longrightarrow \langle R, R(X), \mathit{rest} \rangle$$

$$\frac{N \leq 0}{\langle R, \text{if } N \text{ then } \bar{s}, \mathit{rest} \rangle \longrightarrow \langle R, \mathit{skip}, \mathit{rest} \rangle} \qquad \frac{N > 0}{\langle R, \text{if } N \text{ then } \bar{s}, \mathit{rest} \rangle \longrightarrow \langle R, \bar{s}, \mathit{rest} \rangle}$$

$$\frac{\langle R, e, \text{skip} \rangle \longrightarrow \langle R', N, \text{skip} \rangle}{\langle R, E[e], \mathit{rest} \rangle \longrightarrow \langle R', E[N], \mathit{rest} \rangle}$$

$$\langle R, N_1\ op\ N_2, \text{skip} \rangle \longrightarrow \langle R, M, \text{skip} \rangle \quad M = \begin{cases} N_1 + N_2 & op = + \\ N_1 - N_2 & op = - \\ 1 & op = (==) \wedge N_1 = N_2 \\ 0 & op = (==) \wedge N_1 \neq N_2 \\ 1 & op = (<) \wedge N_1 < N_2 \\ 0 & op = (<) \wedge N_1 \geq N_2 \end{cases}$$

$$\frac{(\mathit{fname}, \bar{s}; \text{return } e) \in F \qquad \langle R, \bar{s}; \text{return } e, \text{skip} \rangle \longrightarrow \langle R', \text{return } N, \text{skip} \rangle}{\langle R, \mathit{fname}(\bar{N}, \mathit{rest} \rangle \longrightarrow \langle R', N, \mathit{rest} \rangle}$$

Figure 7. Dynamic semantics for the example language (§3.4.1)

$$\frac{L \vdash (\mathcal{S}, \circ, SP[L]) \Longrightarrow (\mathcal{S}', \circ, N)}{\langle R, \text{extend}\{L\}, s; \mathit{rest}\rangle \longrightarrow \langle R', \text{skip}, \mathit{rest}\rangle} \qquad \frac{L \vdash (\mathcal{S}, \bullet) \curvearrowright (\mathcal{S}', \circ)}{L \vdash (\mathcal{S}, \circ, \texttt{break}) \Longrightarrow (\mathcal{S}, \circ, \texttt{continue})}$$

$$\frac{L \vdash (\mathcal{S}, \bullet, P) \Longrightarrow (\mathcal{S}', \bullet, N) \qquad L \vdash (\mathcal{S}', \bullet) \curvearrowright (\mathcal{S}'', \circ)}{L \vdash (\mathcal{S}, \bullet) \curvearrowright (\mathcal{S}'', \circ)} \qquad \frac{L \vdash (\mathcal{S}, \bullet, P) \Longrightarrow (\mathcal{S}', \circ, N)}{L \vdash (\mathcal{S}, \bullet) \curvearrowright (\mathcal{S}', \circ)}$$

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