Hardware Synthesis using SAFL and Application to Processor Design
(Invited Talk)

Alan Mycroft[1,2] and Richard Sharp[1]

[1] Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK

[2] AT&T Laboratories Cambridge
24a Trumpington Street, Cambridge CB2 1QA, UK

am@cl.cam.ac.uk
rws26@cl.cam.ac.uk

Abstract. We survey the work done so far in the FLaSH project (Functional Languages for Synthesising Hardware) in which the core ideas are (*i*) using a functional language SAFL to describe hardware computation; (*ii*) transforming SAFL programs using various meaning-preserving transformations to choose the area-time position (e.g. by resource duplication/sharing, specialisation, pipelining); and (*iii*) compiling the resultant program in a *resource-aware* manner (keeping the gross structure of the resulting program by a 1–1 mapping of function definitions to functional units while exploiting ease-of-analysis properties of SAFL to select an efficient mapping) into hierarchical RTL Verilog.
After this survey we consider how SAFL allows some of the design space concerning pipelining and superscalar techniques to be explored for a simple processor in the MIPS style. We also explore how ideas from partial evaluation (static and run-time data) can be used to unify the disparate approaches in Hydra/Lava/Hawk and SAFL and to allow processor specialisation.

1 Introduction

There are many formalisms which allow one to specify circuits. Commercially, the two most important are VHDL and Verilog. From the perspective of this work, they can be regarded as equivalent. They are rich languages which have multiple interpretations according to the intended use; different subsets are used according to the interpretation. It is worth summarising the interpretations/subsets here since we wish to use the terminology later:

Structural (netlist) subset: programs are seen as specifying the basic system components and their interconnecting wires.

RTL subset: extra primitives are provided to wait for events (e.g. a positive clock edge), compute expressions and transfer data between registers.

Behavioural subset: as well as the primitives available in the RTL subset, programs at the behavioural-level can exploit higher-level constructs such as sequencing, assignment and `while`-loops.

Simulation interpretation: in this interpretation the whole language is acceptable; programs are seen as input to a discrete event simulator (useful for debugging the above forms).

One reason for the success of VHDL and Verilog has been their ability to represent circuits composed partly of structural components and partly of behavioural components (like embedding assembly code in a higher-level language). Another benefit is that the languages provide a powerful set of non-synthesisable constructs which are useful for simulation purposes (e.g. printing values to the screen).

Large investments have been made in algorithms and optimisations for synthesising hardware from Verilog/VHDL. However, the optimisations typically depend on the source program being standard in some sense (e.g. multiple components sharing a common clock event). Often one finds that less standard designs such as those forming asynchronous systems may not be synthesisable by a given tool, or worse still that a necessary asynchronous hardware cell required by a netlist is not in a vendor's standard cell library.

RTL-compilers, which have had a significant impact on the hardware design industry, translate RTL specifications to silicon (via a netlist representation). Tasks performed by an RTL-compiler include *Logic synthesis*—the translation of arithmetic and boolean expressions to efficient combinatorial logic and *Place and Route*—deciding where to position components on a chip and how to wire them up.

More recently *behavioural synthesisers* have become available. Behavioural synthesis (sometimes referred to as *high-level synthesis*) is the process of translating a program in a behavioural subset into a program in the RTL subset. Behavioural synthesis is often divided into three separate tasks [14]:

- *Allocation* is typically driven by user-supplied directives and involves choosing which resources will appear in the final circuit (e.g. three adders, two multipliers and an ALU).
- *Binding* is the process of assigning operations in the high-level specification to low-level resources—e.g. the `+` in line 4 of the source program will be computed by `adder_1` whereas the `+` in line 10 will be computed by the `ALU`.
- *Scheduling* involves assigning start times to operations in the flow-graph such that no two operations will attempt to access a shared resource simultaneously. Mutually-exclusive access to shared resources is ensured by statically serialising operations during scheduling.

It is our belief that even the higher-level behavioural forms of VHDL and Verilog offer little in the way of abstraction mechanisms. We justify this claim by observing that, when writing in VHDL or Verilog, low-level implementation details and circuit structure tend to become fixed very early in the design process. Another example is that the protocol for communication between two

functional units (often including exact timing relationships) is spread between their implementations and therefore is difficult to alter later. We develop this argument further in [25].

The problem we wish to identify is the conflict between (*i*) being able to make late, sweeping, changes to a system's implementation (though not to its logical specification); and (*ii*) the efficiency gained from low-level features (such as those found in VHDL and Verilog.) It is worth noting an analogy with programming languages here: efficiency can be gained by exploiting low-level features (e.g. machine code inserts or enthusiastic use of pointers) but the resulting system becomes harder for compilers or humans to analysis and optimise.

A principal aim of our work in the FLaSH project (Functional Languages for Synthesising Hardware), which started in 1999 at AT&T Research Laboratories Cambridge, was to adopt an aggressively high-level stance—we wanted to design a system in which (*i*) the programming language is clean (no 'implementation-defined subsets'); (*ii*) the *logical* structure can be specified but its realisation as *physical* structure can easily be modified even at late stages of design; and (*iii*) programs are susceptible to compiler analysis and optimisation facilitating the automatic synthesis of efficient circuits. We envisaged a single design framework in which we could, for example, select between synchronous and asynchronous design or rework the exact number of functional units, how they are shared and even the interconnection protocol (e.g. changing parallel-on-ready-signal to serial-starting-on-next-clock-tick) at any stage in the design process ('late-binding').

Of course the full aim as stated above has not yet been reached, but we have designed and implemented a language, SAFL 'Statically Allocated Parallel Function Language' embodying at least some of the principles above. One additional principle which we consider important about SAFL, or at least its intended implementation, is that it is *resource-aware*. By this we mean that its constructs map in a transparent way to hardware blocks, so that a program's approximate area-time consumption is clear from the SAFL source—compare the way that the space and time complexity of a C program is reasonably manifest in the program source because its primitives are associated with small machine code sequences.

The FLaSH project's optimising *compiler* translates SAFL into hierarchical RTL Verilog in a resource-aware manner. Each function definition compiles into a single hardware block; function calls are compiled into wiring between these functional blocks. In this framework multiple calls to the same source function corresponds to resource-sharing at the hardware level. Our compiler automatically deals with sharing issues by inserting multiplexers and static fixed-priority arbiters where required. Optimisations which minimise these costs have been implemented.

Accompanying the compiler is a *transformer*. This tool is intended to make semantics-preserving transformations on the SAFL source, typically with user-guidance, to facilitate architectural exploration. Although we have not yet built the transformer tool, we have published various transformation techniques which

embody a wide range of implementation tradeoffs (e.g. functional unit duplication/sharing [19] and hardware-software co-design [20]). A key advantage of this approach is that it factors the current black-box user-view of synthesis tools ('this is what you get') into a source-to-source transformation tool which makes global changes visible to users followed by a more local-basis compilation tool.

Although a full comparison with other work is given in Section 1.1, we would like here to draw a distinction between this work and the framework used in Hydra [22], Lava [2] and Hawk [12]. Our aim in SAFL is to create a high-level *behavioural* language which is compiled into RTL Verilog whose compiler acts as an optimising back-end targeting silicon. SAFL can also run as an ML-style functional language for *simulation*. The Hydra/Lava/Hawk framework on the other hand uses the power of the functional language essentially for interconnecting lower-level components, and thus it is a *structural* language in our taxonomy. Note that the use of alternate interpretations of basis-functions (e.g. `and`, `or` etc.) means that Lava programs can be used for simulation as well as synthesis. We do not wish to claim either work is 'better', merely that they address different issues: we are looking for an "as high-level as possible" language from which to synthesise hardware and rely on others' compilers from RTL downwards; Hydra/Lava/Hawk concentrates on being able to describe hardware at a structural level (possibly down to geometric issues) in a flexible manner.

Although we have found SAFL a very useful vehicle to study issues of high-level behavioural synthesis, it is not the final word. In particular the function model used has a call-and-wait-for-result interface which does not readily interface with external state or multiple clock domains which are essential for a serious hardware design language. Recent work [25] has demonstrated that adding process-calculus features, even including restricted channel passing, can add to the expressive power of SAFL without losing resource-awareness; the resultant language is called SAFL+ and provides both I/O and interaction with components having state. However it is not so clear how much this increased expressive power costs in terms of the increased complexity of validating source-to-source transformations.

The following survey section of this paper draws on ideas from our other work: giving a general overview [18] at the start of the project, examining the theoretical basis of static allocation with parallelism in SAFL [19], describing the FLaSH compiler [23], examining hardware-software co-design [20] and studying the concept (so-called by analogy with soft typing) of soft scheduling [24].

After examining other related work in Section 1.1, the remainder of the paper is structured as follows: Section 2 introduces the SAFL language, how it might be compiled naïvely for synchronous hardware, the rôle of resource-awareness, how analyses aid compiler optimisations and concludes with a look at compiling to asynchronous or GALS (globally synchronous locally synchronous) hardware. Section 3 studies how source-to-source transformations at the SAFL level (together with the resource-awareness assumption) can reposition the hardware implementation on the area-time spectrum. These ideas are applied in Section 4 when a simple CPU is defined and SAFL transformations demonstrated which

introduce pipeline and superscalar features. Section 5 considers how the ideas in Hydra/Lava/Hawk of using a functional language to express the *structural* composition of a system can be merged with the SAFL idea of *behavioural* definition; this then leads on to a type system and to an investigation of partial evaluation in hardware. Finally, Section 6 concludes.

1.1 Comparison with Other Work

In the previous section we motivated our work by comparing SAFL to VHDL and Verilog. We also outlined the differences between our approach and that of Hydra/Lava/Hawk. This section continues by comparing SAFL with a number of other hardware design languages and methodologies.

We are not the first to observe that the mathematical properties of functional languages are desirable for hardware description and synthesis. A number of synchronous dataflow languages, the most notable being LUSTRE [5], have been used to synthesise hardware from declarative specifications. However, whereas LUSTRE is designed to specify reactive systems SAFL describes interactive systems (this taxonomy is introduced in [6]). Furthermore LUSTRE is inherently synchronous: specifications rely on the explicit definition of clock signals. This is in contrast to SAFL which could, for example, be compiled into either synchronous or asynchronous circuits.

The ELLA HDL is often described as functional. However, although constructs exist to define and use functions the language semantics forbid a resource-aware compilation strategy. This is illustrated by the following extract from the ELLA manual [17]: *"Once you have created a named function, you can use instances of it as required in other functions ... [each] instance of a function represents a distinct copy of the block of circuitry."* ELLA contains low-level constructs such as `DELAY` to create feedback loops, restricting high-level analysis. SAFL uses tail-recursion to represent loops at the semantic level; this strategy makes high-level analysis a more powerful technique.

Languages such as HardwareC [10] and Tangram [1] allow function definitions to be treated as shared resources. However, we feel that these projects have not gone as far as us in exploiting the potential benefits of resource-awareness. In particular:

- SAFL's dynamic scheduling policy (which relies on resource-awareness) leads to increased expressivity (and, in some cases, increased efficiency) over the more conventional static scheduling algorithms employed in HardwareC, Balsa and Tangram.
- We have developed a number of analyses and optimisations which are only made possible by structuring hardware as a series of function definitions [24, 23].
- We have investigated the impact of source-to-source transformations on SAFL and shown that it is a powerful tool for exploring the design-space. The functional properties of SAFL make it easier to apply transformations.

A number of languages have been developed which provide structural abstractions similar to those available in the Lava/Hawk framework. For example HML [11] is one such language based on Standard ML [15]; Jazz [9] combines a polymorphic type-system with object oriented features.

2 SAFL and its Compiler

In this section we introduce the SAFL language and show how it can be mapped to synchronous hardware. For space reasons, material described more fully elsewhere will only be summarised.

2.1 SAFL Language

SAFL has syntactic categories e (term) and p (program). First suppose that c ranges over a set of constants, x over variables (occurring in `let` declarations or as formal parameters), a over primitive functions (such as addition) and f over user-defined functions. For typographical convenience we abbreviate formal parameter lists $(x_1, \ldots, x_k)$ and actual parameter lists $(e_1, \ldots, e_k)$ to $\vec{x}$ and $\vec{e}$ respectively; the same abbreviations are used in `let` definitions. Then SAFL programs p are given by recursion equations over expressions e; these are given by:

$$\begin{aligned} e ::= {} & c \mid x \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \mid \texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0 \mid \\ & a(e_1, \ldots, e_{arity(a)}) \mid f(e_1, \ldots, e_{arity(f)}) \\ p ::= {} & \texttt{fun } f_1(\vec{x}) = e_1; \ \ldots \ ; \texttt{fun } f_n(\vec{x}) = e_n; \end{aligned}$$

It is sometimes convenient to extend this syntax slightly. In later examples `e1 ? e2:e3` is used as an abbreviated form of `if-then-else`; similarly we use a `case`-expression instead of iterated tests; we also write `e[n:m]` to select a bit-field `[n..m]` from the result of expression `e` (where `n` and `m` are integer constants).

There is a syntactic restriction that whenever a call to function f_j from function f_i is part of a cycle in the call graph of p then we require the call to be a tail call.[1] Note that calls to a function not forming part of a cycle can occur in an arbitrary expression context. This ensures that storage for the variables and temporaries of p can be stored statically—in software terms the storage is associated with the code of the compiled function; in hardware terms it is associated with the logic to evaluate the function body.

It is often convenient to assume that functions have been (re-)ordered so that functions in a strongly connected component of the call graph are numbered contiguously (we call such strongly connected components function *groups* [19]). Apart from calls within groups, functions f_i can only call smaller numbered $(j < i)$ functions f_j. It is also convenient to assume that the main entry point

[1] Tail calls consist of calls forming the whole of a function body, or nested solely within the bodies of `let-in` expressions and consequents of `if-then-else` expressions.

to be called from the external environment, often written `main()`, is the last function f_n.

The other main feature of SAFL, apart from static allocatability, is that its evaluation is limited only by data flow (and control flow at user-defined call and conditional). Thus, in a let-expression `let` $\vec{x} = (e_1, \ldots, e_k)$ `in` e_0 or in a call $f(e_1, \ldots, e_k)$ or $a(e_1, \ldots, e_k)$, all the $e_i (1 \leq i \leq k)$ are to be evaluated concurrently. The body e_0 of a `let` is also evaluated concurrently subject only to data flow. In the conditional `if` e_1 `then` e_2 `else` e_3 we first evaluate (only) e_1; one of e_2 or e_3 is evaluated after its result is known.

This brings forth one significant interaction between static allocation and concurrency: in a call such as `f(g(x),g(y))` the two calls to `g` must be serialised, since otherwise the same statically allocated storage (the space for `g`'s argument) would be required simultaneously by competing processes. In hardware terms (see next section) a critical region, and arbiter to control entry to it, is synthesised around the implementation of `g` [24]. Section 2.3 shows how these arbiters can often be eliminated from the design while reference [19] discusses the interaction between static allocation and concurrency in more detail.

Note in the above example `f(g(x),g(y))` there is little point in providing access to `g` by arbiter—simple compile-time choice of argument order is more effective. However in a more complicated case, such as `f(g(h(x)),g(k(x)))` where execution times for `h` and `k` are data-dependent, it can often make better sense for efficiency to keep run-time arbitration and to invoke first whichever call to `g` has its argument complete first, and then to invoke the other.

SAFL uses eager evaluation; we are sometimes asked why, especially given that Haskell-based tools (e.g. Lava and Hawk) are lazy. The resolution is that lazy-evaluation is problematic for our purposes. Firstly it is not so clear where to store the closures for suspended evaluations—simple tail recursion for n iterations in Haskell can often require $O(n)$ space; while this can sometimes be detected and optimised (strictness analysis) the formal problem is undecidable and so poses a difficulty for language design. Secondly lazy evaluation is inherently more sequential than eager evaluation—true laziness without strictness optimisation always has a single (i.e. sequential) idea of 'the next redex'.

Of course, a single tail-recursive function can be seen as a non-recursive function with a `while`-loop and assignment to parameters.[2] By repeatedly inline-expanding calls and making assignable local definitions to represent formal parameters, any SAFL program can be considered a nest of `while`-loops. However this translation has lost several aspects of the SAFL source: (*i*) the inlining can increase program size exponentially; (*ii*) not only is the association of meaningful names to loops lost, but the overall structure of the circuit (i.e. the way in which the program was partitioned into separate functional blocks) is also lost—thus

[2] A function group can first be treated as a single function having the disjoint union of its component functions' formal parameters and having a body consisting of the component functions' bodies enclosed within a `case`-expression. The `case`-expression activates the relevant original function body based on the disjoint union actual parameter.

resource-awareness is compromised; and (*iii*) concurrent function calls require a `par` construct to implement them as concurrent `while`-loops thus adding complexity.

2.2 Naïve Translation to Synchronous Hardware

In this section we assume a global clock *clk* which progresses computation. All signals apart from *clk* are clocked by it, i.e. they become valid a small number of gate propagation delays after it and are therefore (with a suitable clock frequency) settled before the setup time for the next *clk*. We adopt the protocol/convention that signal wires (e.g. *request* and *ready*) are held high for exactly one pulse.

A simple translation of a SAFL function definition

$$\texttt{fun}\ f(\vec{x}) = e$$

is to a functional unit H_f which contains:

- an input register file, r, clocked by *clk*, of width given by $\vec{x}$;
- a *request* signal which causes data to be latched into r and computation of e to start;
- an output port P, here just specification of output wires to hold the value of e;
- a *ready* signal which is asserted when P is valid.

The SAFL primitives are compiled each having an output port and *request* and *ready* control signals; they may use the output ports of their subcomponents:

constants c: the constant value forms the output port and the *request* signal is merely copied to the *ready* signal.

variables x: since variables have been clocked into a register then they can be treated as constants, save that the output port is connected to the register.

if e_1 **then** e_2 **else** e_3: first e_1 is computed; when it is ready its boolean output routes a *request* signal either to e_2 or to e_3 and also routes the *ready* signal and output port of e_2 or e_3 to the result.

let $\vec{x} = \vec{e}$ **in** e_0: the $\vec{e}$ are computed concurrently (all their *request*s are activated); when all are complete e_0 is activated. Note that the results of the $\vec{e}$ are not latched.

built-in function calls $a(\vec{e})$: these are expanded in-line; the $\vec{e}$ are computed concurrently and their outputs placed as input to the logic for a; when all are complete this logic is activated;

user function calls $g(\vec{e})$: in the case that the called function only has one call (being non-recursive) the effect is similar to a built-in call—the $\vec{e}$ are computed concurrently and their outputs connected as to the input register file for g; when all are complete the *request* for g is activated; its output port and *ready* signal form those for $g(\vec{e})$. More complex function calls require more sophisticated treatment—see below.

The above explanation of calls to user-defined functions was incomplete in that it did not explain (*i*) how to implement a recursive call (SAFL restricts these to tail calls) nor (*ii*) how to control access to *shared* function blocks.

In the former case, the tail call $g(\vec{e})$ merely represents a loop (see below for mutual recursion); hence hardware is generated to route the values of $\vec{e}$ back to the input register for g and to re-invoke the *request* signal for the body of g when all the $\vec{e}$ have completed. In case (*ii*) we need to arbitrate between the possibly concurrent activation of g by this call and by other calls. The solution is to build a (synchronous) arbiter which: accepts *request* lines from the k call sites and keeps a record of which (at most one) of these have a call active. When one or more *request*s are present and no call is active, an implementation-defined *request* is selected and its values routed to the input register for g. On completion of g the *ready* signal of its body is routed back to the appropriate caller; the result value port can simply be connected to all callers.

One final subtlety in this compilation scheme concerns the values returned by functions invoked at multiple call sites (here not counting internal recursive calls). Consider a SAFL definition such as

```
f(x,y) = g(h(x+1),k(x+1),k(y))
```

where there are no other calls to `h` and `k` in the program. Assuming `h` clocks `x+1` into its input register, then the output port $P_{\texttt{h}}$ of `h` will continue to hold its value until it is clocked into the input register for `g`. However, assuming we compute the value of `k(y)` first, its result value produced on $P_{\texttt{k}}$ will be lost on the subsequent computation of `k(x+1)`. Therefore we insert a clocked *permanisor* register within the hardware functional unit, $H_{\texttt{f}}$ corresponding to `f()`, which holds the value of `k(y)` (this should be thought of as a temporary used during expression evaluation in high-level languages). In the naïve compilation scheme we have discussed so far, we would insert a permanisor register for every function which can be called from multiple sites; the next section shows how static analysis can avoid this.

Adding permanisor registers at the output of resources, like `k` above, which may become valid and then invalid during a single call (to `f` above) explains our ability to avoid latching variables defined by `let`—permanisors have already ensured that signals of `let`-bound variables remain valid as long as is needed. The full details of the compilation process are described in [23].

2.3 Optimised Translation to Synchronous Hardware

Our compiler performs a number of optimisations based on whole-program analysis which improve the efficiency of the generated circuits (both in terms of time and area). This section briefly outlines some of these optimisations and refers the reader to papers which describe them in detail.

Removing Arbiters: Recall that our compiler generates (synchronous) arbiters to control access to shared function-blocks. In some cases we can infer that, even if a function-block is shared, calls to it will not occur simultaneously.

For example, when evaluating `f(f(x))` we know that the two calls to `f` must always occur sequentially since the outermost call cannot commence until the innermost call has been completed.

Whereas conventional high-level synthesis packages schedule access to shared resources by statically serialising conflicting operations, SAFL takes a contrasting approach: arbiters are automatically generated to resolve contention for all shared resources dynamically; static analysis techniques remove redundant scheduling logic. We call the SAFL approach *soft scheduling* to highlight the analogy with Soft Typing [4]: the aim is to retain the flexibility of dynamic scheduling whilst using static analysis to remove as many dynamic checks as possible. In [24] we compare and contrast soft scheduling to conventional static scheduling techniques and demonstrate that it can improve both the expressivity and efficiency of the language.

One of the key points of soft scheduling is that provides a convenient compromise between static and dynamic scheduling, allowing the programmer to choose which to adopt. For example, compiling `f(4)+f(5)` will generate an arbiter to serialise access to the shared resource $H_{\texttt{f}}$ *dynamically*. Alternatively we can use a `let`-declaration to specify an ordering *statically*. The circuit corresponding to `let x=f(4) in x+f(5)` does not require dynamic arbitration; we have specified a static order of access to $H_{\texttt{f}}$. Note that program transformation can be used to explore static vs. dynamic scheduling tradeoffs.

Register Placement: In our naïve translation to hardware (previous section) we noted that a caller latches the result of a call into a register. We call such registers *permanising registers* since they are required to keep the result of a call to H_f permanent even if the value on H_f's output port subsequently changes (e.g. due to another caller accessing shared resource H_f). However, in many cases we can eliminate permanising registers: if we can infer that the result of a call to function f is guaranteed to remain valid (i.e. if no-one else can invoke f whilst the result of the call is required) then the register can be removed [23].

Cycle Counting: Consider translating the following SAFL program into synchronous hardware:

```
fun f(x) = g(h(x+1), h(k(x+2)))
```

Note that we can remove the arbiter for `h` if we can infer that the execution of `k` always requires more cycles than the execution of `h`.

Zero Cycle Functions In the previous section we stated that it is the duty of a function to latch its arguments (this corresponds to callee-save in software terms). However, latching arguments necessarily takes time and area which, in some cases, may be considered unacceptable. For example, if we have a function representing a shared combinatorial multiplier (which takes a single cycle to compute its result), the overhead of latching the arguments (another cycle) doubles the latency.

The current implementation of the SAFL compiler [23] allows a user to specify, via pragma, certain function definitions as *caller-save*—i.e. it is then the duty of the caller to keep the arguments valid throughout the duration of the call. An extended register-placement analysis ensures that this obligation is kept, by adding (where necessary) permanising registers for such arguments at the call site. In some circumstances[3], this allows us to eliminate a resource's argument registers completely facilitating fine-grained, low-latency sharing of resources such as multipliers, adders etc.

There are a number of subtleties here. For example, consider a function `f` which adopts a caller-save convention and does not contain registers to latch its arguments. Note that `f` may return its result in the same cycle as it was called. Let us now define a function `g` as follows:

```
fun g(x) = f(f(x))
```

We have to be careful that the translation of `g` does not create a combinatorial loop by connecting `f`'s output directly back into its input. In cases such as this *barriers* are inserted to ensure that circuit-level loops always pass through synchronous delay elements (i.e. registers or flip-flops).

2.4 Translation to Asynchronous Hardware

Although this has not been implemented yet, note that the design philosophy outlined in Section 2.2 made extensive use of request/acknowledge signals. Our current synchronous compiler models control events as 1-cycle pulses. With the change to edge events (either 2-phase or 4-phase signalling) and the removal of the global clock the design becomes asynchronous. The implementation of an asynchronous SAFL compiler is the topic of future work.

Note that the first two optimisations presented in the previous section (removal of arbiters and permanising registers) remain applicable in the asynchronous case since they are based on the causal-dependencies inherent in a program itself (e.g. when evaluating `f(g(x))`, the call to `f` cannot be executed until that to `g` has terminated). Although we cannot use the "cycle counting" optimisation as it stands, detailed feedback from model simulations incorporating layout delays may be enough to enable a similar type of optimisation in the asynchronous case.

2.5 Globally Asynchronous Locally Synchronous (GALS) Hardware

One recent development has been that of Globally Asynchronous Locally Synchronous (GALS) techniques where a number of separately clocked synchronous subsystems are connected via an asynchronous communication architecture. The GALS methodology is attractive as it offers a potential compromise between (*i*) the difficulty of distributing a fast clock in large synchronous systems; and (*ii*)

[3] Note that if the function is tail-recursive we cannot eliminate its argument registers since they are used as workspace during evaluation.

the seeming area-time overhead of fully-asynchronous circuits. In a GALS circuit, various functional units are associated with different *clock domains*. Hardware to interface separate clock-domains is inserted at domain boundaries.

Our initial investigations of using SAFL for this approach have been very promising: clock domain information can be an annotation to a function definition; the SAFL compiler can then synthesise change-of-clock-domain interfaces exactly where needed.

3 Transformations in SAFL

As a result of our initial investigations, we believe that source-to-source transformation of SAFL is a powerful technique for exploring the design space. In particular:

- The functional properties of SAFL allow equational reasoning and hence make a wide range of transformations applicable (as we do not have to worry about side effects).
- The resource-aware properties of SAFL give fold/unfold transformations precise meaning at the design-level (e.g. we know that duplicating a function definition in the source is guaranteed to duplicate the corresponding resource in the generated circuit).

Although we have not yet built the transformer tool, we envisage it being driven with a GUI interface. There is also scope for semi-automatic exploration (cf. theorem proving), including perhaps hill-climbing.

In this section we give examples of a few of the transformations we have experimented with. We start with a very simple example, using *fold/unfold* transformations [3] to express resource duplication/sharing and unrolling of recursive definitions. Then a more complicated transformation is presented which allows one to collapse a number of function definitions into a single function providing their combined functionality. Finally we briefly outline a much larger, global transformation which allows one to investigate hardware/software co-design.

We have observed that the fold/unfold transformation is useful for trading area against time. As an example of this consider:

```
fun f x = ...
fun main(x,y) = g(f(x),f(y))
```

The two calls to `f` are serialised by mutual exclusion before `g` is called. Now use fold/unfold to duplicate `f` as `f'`, replacing the second call to `f` with one to `f'`. This can be done using an unfold, a definition rule and a fold yielding

```
fun f  x = ...
fun f' x = ...
fun main(x,y) = g(f(x),f'(y))
```

The second program has more area than the original (by the size of `f`) but runs more quickly because the calls to `f(x)` and `f'(y)` execute in parallel.

Note that fold/unfold allows us to do more than resource/duplication sharing tradeoffs; folding/unfolding recursive function calls before compiling to synchronous hardware corresponds to trading the amount of work done per clock cycle against clock speed. For example, consider the following specification of a shift-add multiplier:

```
fun mult(x, y, acc) =
  if (x=0 or y=0) then acc
     else mult(x<<1, y>>1, if y[0:0] then acc+x else acc)
```

These 3 lines of SAFL produce over 150 lines of RTL Verilog. Synthesising a 16-bit version of `mult`, using Mentor Graphics' *Leonardo* tool, yields 1146 2-input equivalent gates. We can mechanically transform `mult` into:

```
fun mult(x, y, acc) =
  if (x=0 or y=0) then acc
  else let (x',y',acc') = (x<<1, y>>1,
                           if y[0:0] then acc+x else acc) in
    if (x'=0 or y'=0) then acc'
    else mult(x'<<1, y'>>1, if y'[0:0] then acc'+x' else acc')
```

which uses almost twice as much area and takes half as many clock cycles.

Another transformation we have found useful in practice is a form of loop collapsing. Consider a recursive main loop (for example a CPU) which may invoke one or more loops in certain cases (for example a multi-step division operation):

```
fun f(x,y) = if x=0 then y else f(x-1, y')
fun main(a,b,c) = if a=0 then b
                  else if ... then main(a', f(k,b), c')
                  else main(a',b',c')
```

Loop collapsing converts the two nested `while`-loops into a single loop which tests a flag each iteration to determine whether the outer, or an inner, loop body is to be executed. This test is almost free in hardware terms:

```
main(inner,x,a,b,c) =
  if inner then                     (* an f() iteration  *)
    if x=0 then main(0,x,a,b,c)     (* exit f()          *)
           else main(1,x-1,a,y',c)  (* another iteration *)
  else
     if a=0 then b
     else if ... then main(1,k,a',b, c')  (* start f()   *)
     else main(0,x,a',b',c')        (* normal main step  *)
```

This optimisation is useful because it allows us to collapse a number of function definitions, $(f_1, \ldots, f_n)$, into a single definition, F. At the hardware-level each

function-block has a certain overhead associated with it (logic to remember who called it, latches to store arguments etc.—see Section 2). Hence this transformation allows us to save area by reducing the number of function-blocks in the final circuit. Note that the reduction in area comes at the cost of an increase in time: whereas previously $f_1, \ldots, f_n$ could be invoked in parallel, now only one invocation of F can be active at once.

Now consider applying this transformation to definitions $(f_1, \ldots, f_n)$ which enjoy some degree of commonality. Once we have combined these definitions into a single definition, F, we can save area further by applying a form of *Common Sub-expression Elimination* within the body of F. This amounts to exploiting `let` declarations to compute an expression once and read it many times (e.g. `f(x)+f(x)` would be transformed into `let y=f(x) in y+y`.)

Despite having a significant impact on the generated hardware, the transformations presented so far have been relatively simple. We have investigated more complicated transformations for exploring hardware/software co-design. Our method takes a SAFL specification, and a user-specified partition into hardware and software parts, and generates a specialised architecture (consisting of a network of heterogenous processors) to execute the software part. The basic idea involves representing processors and instruction memories (containing software) in SAFL itself and using a software compiler from SAFL to generate the code contained in the instruction memories. The details are beyond the scope of this paper; we refer the reader to [20] for more information.

4 Pipelines and Superscalar Expression in SAFL

As an example of how transformations work, consider the simple processor, resembling DLX [7] or MIPS, given in Fig. 1 (we have taken the liberty of removing most type/width information to concentrate on essentials and also omitted '`in`' when it occurs before another '`let`'). The processor has seven representative instructions, defined by enumeration

```
enum { OPhalt, OPj, OPbz, OPst, OPld, OPadd, OPxor };
```

and has two instruction formats (reg-reg-imm) and (reg-reg-reg) determined by a mode bit `m`. The processor is externally invoked by a call to `cpu` providing initial values of `pc` and registers; it returns the value in register zero when the `OPhalt` instruction is executed. There are two memories: `imem` which could be specified by a simple SAFL function expressing instruction ROM and `dmem` representing data RAM. The function `dmem` cannot be expressed directly in SAFL (although it can in SAFL+, our extended version [25]). It is declared by a native language interface and defined directly in Verilog: calls to `dmem` are serialised (just like calls to user-functions); if they could be concurrent a warning is generated. In this case it is clear that at most one call to `dmem` occurs per cycle of `cpu`. The intended behaviour of `dmem(a,d,w)` is to read from location `a` if `w=0` and to write value `d` to location `a` if `w=1`. In the latter case the value of `d` is also returned. The

```
fun cpu(pc, regs) =
  (let I = imem(pc)
   let (op,m,rd,ra) = (I[31:27], I[26], I[21:25], I[16:20])
   let (rb,imm) = (I[0:4], sext32(I[0:15]))
   let A = regs[ra]
   let B = m==0 ? imm : regs[rb]
   let C = alu(op, A, B)
   let D = case op of OPld => dmem(C,0,0)
                    | OPst => dmem(C,regs[rd],1)
                    | _    => C
   let regs' = case op of OPld => regs[D @ rd]
                        | OPadd => regs[D @ rd]
                        | OPxor => regs[D @ rd]
                        | _    => regs
   let pc' = case op of OPj => B
                      | OPbz => pc+4 + (A==0 ? imm : 0)
                      | _ => pc+4
   in (op==OPhalt ? regs'[0] : cpu(pc', regs')));
```

Fig. 1. Simple processor

use of functional arrays for `regs` and `regs'` is also to be noted: the definition `let regs' = regs[v @ i]` yields another array such that

$$\begin{aligned} \texttt{regs'[i]} &= \texttt{v} \\ \texttt{regs'[j]} &= \texttt{regs[j]} \quad \text{if } \texttt{j} \neq \texttt{i} \end{aligned}$$

This can be seen as shorthand: the array `regs` corresponds to a tuple of simple variables, say (`r0`, `r1`, `r2`, `r3`)) and the value `regs[i]` is shorthand for the expression

```
(i==0 ? r0 : (i==1 ? r1 : (i==2 ? r2 : r3)))
```

and the array value `regs[v @ i]` is shorthand for the expression

```
((i==0 ? v : r0), (i==1 ? v : r1), (i==2 ? v : r2), (i==3 ? v : r3)).
```

Note that the SAFL restrictions mean that no dynamic storage is required, even when using array values as first-class objects. There is one advantage of the array notation in that it allows alternative implementation techniques; here we *may* infer that `regs` and `regs'` are never both live and share their storage as a single register file (when the conditionals above become multiplexors) but equally we may choose to use a rather less physically localised implementation, for example the rotary pipelines of Moore at al. [16].

Now let us turn to performance. We start by making three assumptions: first, that both `imem` and `dmem` take one clock tick; second, that the compiler also inserts a clocked register file at the head of `cpu` to handle the recursive loop; and

that the `alu()` function is implemented without clocked registers which we will here count as just one delta[4] delay. We can now count clock and delta cycles, here just in terms of high-level SAFL data flow. Writing $n.m$ to mean n cycles and m delta cycles (relative to an external or recursive call to `cpu` being the 0.0 event), we can derive:

variable	cycle count
entry to body of `cpu()`	0.0
`I`	1.0
`(op, m, rd, ra), (rb, imm)`	1.1
`A, B`	1.2
`C`	1.3
`D`	2.0 or 1.4[5]
`regs'`	2.1 or 1.5
`pc'`	1.3
recursive call to `cpu()`	2.2 or 1.6
next entry to body of `cpu()`	3.0 or 2.0

Note that we have counted SAFL-level data dependencies instead of true gate-delays; this is entirely analogous to counting the number of high-level statements in C to execute program speed instead of looking at the assembler output of a C compiler to count the exact number of instructions. The argument is that justifying many optimisations only needs this approximate count. A tool could easily annotate declarations with this information.

The result of this is that we have built a CPU which takes three clock cycles per memory reference instruction, two clock cycles for other instructions and with a critical path of length 6 delta cycles (which acts as a limit on the maximum clock rate). Actually, by a simple adjustment to the compiler, we could arrange that that `cpu()` is clocked at the same time as `imem()` and therefore achieve a one- or two-cycle instruction rate.

Now suppose we wish to make our simple CPU go faster; two textbook methods are adding a pipeline or some form of superscalar processing. We wish to reduce the number of clock cycles per instruction cycle and also to reduce the critical path length to increase the clock frequency.

The simplest form of pipelining occurs when we wish to enable the `dmem` and `imem` accesses to happen concurrently. The problem in the above design is that the memory address argument to `dmem` is only produced from the `imem` result. Hence we transform `cpu()` to `cpu1a()` as shown in Fig. 2; the suffix '1' on an identifier refers to a value which was logically produced one instruction ago. This transformation is always valid (as a transformation on a recursive program schema and thus computes the same SAFL function) but unfortunately the conditional test for `OPhalt` requires the calls to `imem` and `dmem` still to be serialised. To produce `cpu2()`, as shown in Fig. 3, we need to make a conscious adjustment to pre-fetch the instruction after the `OPhalt` by interchanging the

[4] A delta cycle corresponds to a gate-propagation delay rather than a clock delay.

[5] Depending on which path is taken.

```
fun cpu1a(pc, op1, regs1, C1, rd1) =
  (let D = case op1 of OPld => dmem(C1,0,0)
                     | OPst => dmem(C1,regs1[rd1],1)
                     | _    => C1
   let regs = case op1 of OPld => regs1[D @ rd1]
                        | OPadd => regs1[D @ rd1]
                        | OPxor => regs1[D @ rd1]
                        | _    => regs1
   in (op1==OPhalt ? regs[0] :        (* note this line *)
      let I = imem(pc)                (* note this line *)
      let (op,m,rd,ra) = (I[31:27], I[26], I[21:25], I[16:20])
      let (rb,imm) = (I[0:4], sext32(I[0:15]))
      let A = regs[ra]
      let B = m==0 ? imm : regs[rb]
      let C = alu(op, A, B)
      let pc' = case op of OPj => B
                         | OPbz => pc+4 + (A==0 ? imm : 0)
                         | _ => pc+4
      in cpu1a(pc', op, regs, C, rd)));
```

Fig. 2. CPU after simple transformation

```
fun cpu2(pc, op1, regs1, C1, rd1) =
  (let D = case op1 of OPld => dmem(C1,0,0)
                     | OPst => dmem(C1,regs1[rd1],1)
                     | _    => C1
   let regs = case op1 of OPld => regs1[D @ rd1]
                        | OPadd => regs1[D @ rd1]
                        | OPxor => regs1[D @ rd1]
                        | _    => regs1
   let I = imem(pc)                   (* note this line *)
   in (op1==OPhalt ? regs[0] :        (* note this line *)
      let (op,m,rd,ra) = (I[31:27], I[26], I[21:25], I[16:20])
      let (rb,imm) = (I[0:4], sext32(I[0:15]))
      let A = regs[ra]
      let B = m==0 ? imm : regs[rb]
      let C = alu(op, A, B)
      let pc' = case op of OPj => B
                         | OPbz => pc+4 + (A==0 ? imm : 0)
                         | _ => pc+4
      in cpu2(pc', op, regs, C, rd)));
```

Fig. 3. CPU with pipelined memory access

`if-then-else` and the `imem()` call. This is now in contrast to `cpu()` and `cpu1a()` where instructions are only fetched when needed to execute. Now letting `NOP` stand for `(OPbz<<27)+0` we see that the call `cpu(pc,regs)` is equivalent to the call `cpu2(pc,NOP,regs,0,0)`, save that the latter requires only one clock for every instruction and that the instruction after an `OPhalt` instruction will now be fetched (but not executed). Note that the calls to `dmem` and `imem` in `cpu2()` are now concurrent and hence will happen on the same clock. It is pleasant to see such subtleties expressible in the high-level source instead of as hidden details.

We can now turn to exploring further the memory interface; in particular suppose we wish to retain separate `imem` (ROM) and `dmem` (RAM), each accessible in a single cycle, but wish both instruction- and data-fetches to occur from either source. This is form of a memory controller. In order to avoid concurrent access on every memory access instruction we wish it to be dual-ported, thus it will take two addresses and return two data values. When two concurrent accesses occur, either to `dmem` or to `imem` (e.g. because a `OPld` in one instruction refers to the memory bank which contains the following instruction), a *stall* will occur. A good memory controller will cause a stall only in this circumstance. Fig. 4 shows how this can be implemented in SAFL; `memctrl` is dual ported

```
fun memctrl(pc,a,d,r,w) =
  (let iv = is_dmem(pc) ? dmem(pc,0,0) : imem(pc)
   let dv = (r or w) ? (is_dmem(a) ? dmem(a,d,w) : imem(a)) : a
   in (iv,dv))
fun cpu3(pc, op0, regs0, C0, rd0) =
  (let (I,D) = memctrl(pc, C0, regs[rd0], op==OPld, op==OPst)
   in ...)
```

Fig. 4. CPU with memory controller

(two arguments and results) each memory access is directed (according to the, simple and presumably one delta cycle, function `is_dmem`) to the appropriate form of memory. The SAFL compiler detects the possible concurrent access to `imem` (and to `dmem`) and protects them both with an arbiter. The effect is as desired, a stall occurs only when the two accesses are to the same memory bank.

Another useful transformation is to reduce the length of the critical path in order to increase the clock rate. In `cpu2` this is likely to be the path through the `alu` function. Fig. 5 shows how the access to `alu` can be pipelined along with memory access to create a three-stage pipeline; here the the suffix '1' (resp. '2') on an identifier refers to a value which was logically produced one (resp. two) instructions ago. The processor `cpu4` works by concurrently fetching from `imem` the current instruction, doing the `alu` for the previous instruction `op1` and doing memory access (and register write-back) for the second previous instruction `op2`. It is not quite equivalent to `cpu2` in that it exposes a *delay slot*; the result of an ALU or load instruction is not written back to `regs` until two instructions later,

```
fun cpu4(pc, op1, A1, B1, rd1, op2, regs2, C2, rd2) =
  (let C = alu(op1, A1, B1)
   let D = case op2 of OPld => dmem(C2,0,0)
                     | OPst => dmem(C2,regs2[rd2],1)
                     | _    => C2
   let regs = case op2 of OPld => regs2[D @ rd2]
                        | OPadd => regs2[D @ rd2]
                        | OPxor => regs2[D @ rd2]
                        | _    => regs2
   let I = imem(pc)
   in (op2==OPhalt ? regs[0] :
      let (op,m,rd,ra) = (I[31:27], I[26], I[21:25], I[16:20])
      let (rb,imm) = (I[0:4], sext32(I[0:15]))
      (* forwarding (a.k.a. by-passing) would go here *)
      let A = regs[ra]
      let B = m==0 ? imm : regs[rb]
      let pc' = case op of OPj => B
                         | OPbz => pc+4 + (A==0 ? imm : 0)
                         | _ => pc+4
      in cpu4(pc', op, A, B, rd,  op1, regs, C, rd1)));
```

Fig. 5. CPU with pipelined ALU and memory access

and thus the following instruction will still 'see' the old value. This is typically avoided by adding *forwarding* or *by-passing* hardware. In our terms this means comparing `rd1` with `ra` and `rb` where indicated and using `C` instead of the value from `regs` on equality.

Returning to the original `cpu()` form for simplicity of expression, we can simply convert it to the superscalar processor shown in Fig. 6; since we have dropped the pipeline we just use the '1' and '2' suffices for the 'left' and 'right' instruction of a pair. As a processor this leaves quite a few things to be desired—for example while the left (`I1`) and right (`I2`) instructions are serialised if `I2` reads from a register written by `I1`, there is no such interlock on memory access for concurrent writes. Similarly there is a *branch delay slot* in that `I2` is carried out even if `I1` is a taken-branch. Further, to gain actual performance improvement one would need a single double-width `imem64` function instead of the two `imem` accesses; perhaps one can manage with a single-width `dmem` and require the assembly code to be *scheduled* to pair memory access and non-memory-access instructions. However, all structural hazards will be removed by the SAFL compiler by its insertion of arbiters around concurrently accessible resources. The SAFL form explicitly lays out various options in the design space. For example, as presented in Fig. 6 a single ALU is shared between the two separate instructions; duplicating this is clearly a good idea; however, less frequently used components (perhaps a multiplier called by the ALU) could be provided in a single form accessed by arbiter. A stall then only happens when the `I1` and `I2` instructions both use such a

```
fun cpu5(pc, regs) =
  (let (I1,I2) = (imem(pc), imem(pc+4))
   let (op1,m1,rd1,ra1) = (I1[31:27], I1[26], I1[21:25], I1[16:20])
   let (rb1,imm1) = (I1[0:4], sext32(I1[0:15]))
   let (op2,m2,rd2,ra2) = (I2[31:27], I2[26], I2[21:25], I2[16:20])
   let (rb2,imm2) = (I2[0:4], sext32(I2[0:15]))
   if ((op1 == OPld or op1 == OPadd or op1 == OPxor) and
       (rd1 == ra2 or (m1==1 and rd1 == rb2)
                   or (op2 == OPst and rd1 == rd2))) then
     ...
     <I2 reads from a register written by I1 -- serialise>
     ...
   else
     let (A1,A2) = (regs[ra1], regs[ra2])
     let (B1,B2) = ((m1==0 ? imm1 : regs[rb1]),
                    (m2==0 ? imm2 : regs[rb2]))
     let (C1,C2) = (alu(op1, A1, B1), alu(op2, A2, B2))
     let D1 = case op1 of OPld => dmem(C1,0,0)
                        | OPst => dmem(C1,regs[rd1],1)
                        | _    => C1
     let D2 = case op2 of OPld => dmem(C2,0,0)
                        | OPst => dmem(C2,regs[rd2],1)
                        | _    => C2
     let regs' = case op1 of OPld => regs[D1 @ rd1]
                           | OPadd => regs[D1 @ rd1]
                           | OPxor => regs[D1 @ rd1]
                           | _    => regs
     let regs'' = case op2 of OPld => regs'[D2 @ rd2]
                            | OPadd => regs'[D2 @ rd2]
                            | OPxor => regs'[D2 @ rd2]
                            | _    => regs'
     let pc' = case op1 of OPj => B1
                         | OPbz => pc+8 + (A1==0 ? imm1 : 0)
                         | _ =>
                   case op of OPj => B2
                            | OPbz => pc+8 + (A2==0 ? imm2 : 0)
                            | _ => pc+8
     in (op1==OPhalt ? regs'[0]
         op2==OPhalt ? regs''[0] : cpu5(pc', regs'')));
```

Fig. 6. Simple superscalar processor

resource; we might choose to accept this point on the speed/cost spectrum and again simply requiring compilers to schedule code to avoid such stalls.

5 Compile-Time and Run-Time Types: Unifying SAFL and Lava

Based on the observation that Lava uses Haskell recursion to specify a circuit on the structural level (such as repetitive or nested circuits) whereas SAFL uses recursion to specify behavioural aspects, we now turn to a two-level language which can express both concepts in a single framework.

The idea is analogous to the distinction between static (compile-time) and (dynamic) run-time data in partial evaluation [8]; we consider partial evaluation for an extended SAFL in Section 5.1.

SAFL's type system (not explicitly spelt out previously, although always present in the implementation) is very simple, being of the form where values each have an associated size (n bits say) and therefore are ascribed type bit_n. Our implementation of SAFL currently requires that each constant, function argument and function result[6] is given an explicit width type in the style of the simply typed lambda-calculus. Function types, corresponding to hardware blocks, are then of type $bit_m \mapsto bit_n$. As in ML, functions can always be considered to have a single input and a single output as above; the addition of product forms to SAFL is then all that is required to model multiple arguments, results and `let` definitions. These can be seen as a family of bundling and unbundling operations:

$$join_{ij} : bit_i * bit_j \mapsto bit_{i+j} \tag{1}$$

$$split_{ij} : bit_{i+j} \mapsto bit_i * bit_j. \tag{2}$$

Similarly the family of sum forms $bit_i + bit_j$ can be represented as as type $bit_{\max(i,j)+1}$ in the usual manner.

SAFL functions are restricted to first order; allowing curried functions whose arguments and result are simple values poses few problems as the closure is of known size to the caller, but the gain in expressiveness does not seem worth the implementation effort. However allowing function values as arguments and results breaks the static allocatability requirement (counter-examples can be constructed based of the idea that any program can be expressed in continuation form using only tail recursion, see [19] for more details). Hence, given the syntactical separation between values and functions, the SAFL type system consists essentially of:

$$\begin{array}{ll} \text{(values)} & bit_n \\ \text{(functions)} & bit_m \mapsto bit_n. \end{array}$$

Now let us consider the framework used in Lava. There, unwrapping the Haskell class treatment, circuits are essentially represented as graphs encoded

[6] Function results only need to be typed to ensure that all non-terminating recursive functions have a well-defined result type.

as a datatype, say *circuit*. Thus, disregarding polymorphism at the moment and including $list(t)$ as representative of user-defined datatypes, the type system is essentially

$$\tau ::= int \mid circuit \mid \tau \to \tau \mid \tau \times \tau \mid list(\tau)$$

Circuit composition operations correspond to functions whose arguments or results are functions.

We can now combine these type systems as follows:

$$\begin{aligned} \sigma &::= bit_n \mid bit_m \mapsto bit_n \\ \tau &::= int \mid \sigma \mid \tau \to \tau \mid \tau \times \tau \mid list(\tau). \end{aligned}$$

The interesting effect is that this type system has now become a two-level type system (first studied by Nielson and Nielson [21]) which has two function constructors: '$\to$' representing compile-time, or structural, composition (cf. Verilog `module` instantiation) and '$\mapsto$' representing run-time, or behavioural, computation (i.e. data movement in silicon).

Let us therefore invent a language, 2-SAFL, based on this type system, which: allows use of general recursion to define compile-time functions (of types $\tau \to \tau$) representing the structural design; and run-time functions (of types $bit_m \mapsto bit_n$ and respecting the SAFL static allocatability rules) representing the behavioural core.[7] At its most primitive (using a $\lambda\vec{x}.e$ form to give function values instead of having separate `fun` declarations) it has expressions, e, given by:

$$\begin{array}{lll} e ::= x \mid c \mid e\,e' & & \\ \quad \mid \lambda^c\,\vec{x}.e & \mid \lambda^r\,\vec{x}.e \\ \quad \mid if^c\ e\ then\ e\ else\ e & \mid if^r\ e\ then\ e\ else\ e \\ \quad \mid let^c\ \vec{x} = e\ in\ e & \mid let^r\ \vec{x} = e\ in\ e \end{array}$$

Here the alternative left-hand-side forms λ^c, let^c etc., correspond to compile-time constructions and those on the right λ^r, let^r etc., correspond to run-time, i.e. SAFL, ones. We have constants (including *fix* to express recursion) and variables of both forms and an overloaded application on both forms. Valid programs have a set of well-formedness rules which express static allocatability restrictions on the SAFL $\lambda^r\vec{x}.e$ form together with type and level constraints on the σ and τ as in [21]; for example that compile-time computation (e.g. application of a value defined by λ^c) cannot occur in the body of a $\lambda^r\vec{x}.e$ form. In examples below we revert to the use of the $\texttt{fun}^{\texttt{c}} f(x) = e$ (and $\texttt{fun}^{\texttt{r}}$) form instead of the $\lambda^c\vec{x}.e$ (and λ^r) used above.

This provides various interesting features, which we have not investigated in detail. For example, it can be used to define SAFL functions which differ only in type:

[7] There is a little surprise here: the structural level is outside the the behavioural level. This is consonant with Lava in which compile-time, i.e. Haskell, functions allow one to manipulate the primitive hardware cells which move values at run-time; all we have done is to provide a richer, behavioural, run-time level in the form of SAFL.

```
func multiplier(n) =
  local funr f(x:bit(n), y:bit(n), acc:bit(n)) : bit(n) =
    if y=0 then acc
    else f(x<<1, y>>1, if y[0:0] then acc+x else acc)
  in f
  end;
funr m1 = multiplier(16);
funr m2 = m1;
funr m3 = multiplier(16);
funr m4 = multiplier(24);
funr main(...) = ... m1 ... m2 ... m3 ... m4 ...
```

Here `m1`, `m2` and `m3` represent 16-bit multipliers and `m4` a 24-bit multiplier. Note that resource-awareness is manifested here by `m1` and `m2` being synonyms for the same multiplier while `m3` is a distinct multiplier (as is `m4`, but this is clear because it differs in width). The type of `multiplier` is then

$$(n \in int) \rightarrow (bit_n * bit_n * bit_n \mapsto bit_n).$$

Similarly, consider the way in which Lava circuits can be wired together by means of functions which operate on values of type *circuit*. We can capture this notion by means of a higher-order function taking arguments in the form of SAFL functions. In SAFL we can already define functional units `f` and `g` and then wire them together to make `h` as follows:

```
funr f(x,y) = e1;
funr g(z) = e2;
funr h(x,y) = f(g(x+1),g(y+1));
```

The 2-SAFL language now allows, as in Lava, the particular combinator here used to define `h` to be abstracted and expressed as a higher-order value (suitable for re-use elsewhere in the system where similar combinators are required):

```
funr f(x,y) = e1;
funr g(z) = e2;
func combine(p,q,a) =
  local funr t(x,y) = p(g(x+1),q(y+a)) in t end;
funr h = combine(f,g,1);
```

This example, although contrived, does show how compile-time functions such as `combine` can be used as in Lava. Supposing f has type $(bit_m * bit_n \mapsto bit_r)$ and g has type $(bit_i \mapsto bit_m)$, then the type of `combine` is:

$$(bit_m * bit_n \mapsto bit_r) \times (bit_j \mapsto bit_n) \times int \rightarrow (bit_i * bit_j \mapsto bit_r).$$

We could summarise the 2-SAFL extension as follows: in the Lava view, the only run-time computations arise from built-in primitives (AND, OR, flip-flops) whereas 2-SAFL has SAFL function definitions as primitive; iterative structure

in Lava is represented via re-entrant graphs whereas in 2-SAFL it is represented via SAFL recursion.

We have not properly investigated type inference on this system—in many ways the compile-time types inherit from Lava, including widths, but the availability of SAFL functions at the run-time level instead of just hardware primitives may pose interesting questions of type inference. (For example, as above, run-time types, e.g. bit_n, can depend on compile-time values, e.g. n.) Instead of addressing this issue, we wish to turn attention to the possibility of exploring transformations which interchange the two forms of function (compile-time and run-time), i.e. partial evaluation.

5.1 Partial Evaluation

Although there is only space for a preliminary discussion of these ideas here, the 2-SAFL language provides a useful base for partial evaluation for hardware. In traditional partial evaluation [8], sometimes known as *program specialisation*, a generic program taking n inputs is specialised, with $k < n$ of its inputs being known values, to result in a specialised program taking n–k inputs. The resultant program is generally more efficient that the original program. Few applications to hardware seem to exist; McKay and Singh [13] consider the problem of run-time specialisation of hardware implemented in FPGAs to increase speed.

Standard partial evaluation seems to fit in well with 2-SAFL; we show how a SAFL program of say three inputs can be reduced to a more efficient program of two arguments in the 2-SAFL framework. For example, consider a multi-cycle multiply-and-add defined by

```
fun mult(x, y, acc) =
  if y=0 then acc
  else mult(x<<1, y>>1, if y[0:0] then acc+x else acc)
```

We can trivially convert this to a multiply-by-13-and-add by defining

```
fun mult13(x,acc) = mult(x,13,acc).
```

Indeed, if `mult13` compiles into a zero-clock function (one which does not latch its inputs) then a call `mult13(`x`,`a`)` will compile into hardware identical to that of a call `mult(`x`,13,`a`)`.

However, the two-level type system can be used to create a specialised version of `mult13`. Writing `mult` first as

```
fun mult = λʳ(x, y, acc).
  if y=0 then acc
  else mult(x<<1, y>>1, if y[0:0] then acc+x else acc)
```

and then, curried on the two forms of λ, gives:

```
fun mult' = λᶜy. λʳ(x, acc).
  if y=0 then acc
  else mult'(y>>1)(x<<1, if y[0:0] then acc+x else acc).
```

This last form is not a valid 2-SAFL program since it contains a compile-time application `mult'(y>>1)` within a λ^r. However `mult'(13)` *can* be unfolded as:

```
fun mult13 = λʳ(x, acc). if 13=0 then acc
  else mult6(x<<1, if 13[0:0] then acc+x else acc)
fun mult6 = λʳ(x, acc). if 6=0 then acc
  else mult3(x<<1, if 6[0:0] then acc+x else acc)
fun mult3 = λʳ(x, acc). if 3=0 then acc
  else mult1(x<<1, if 3[0:0] then acc+x else acc)
fun mult1 = λʳ(x, acc). if 1=0 then acc
  else mult0(x<<1, if 1[0:0] then acc+x else acc)
fun mult0 = λʳ(x, acc). if 0=0 then acc
  else mult0(x<<1, if 0[0:0] then acc+x else acc).
```

which simplies to:

```
fun mult13 = λʳ(x, acc). mult6(x<<1, acc+x)
fun mult6 = λʳ(x, acc). mult3(x<<1, acc)
fun mult3 = λʳ(x, acc). mult1(x<<1, acc+x)
fun mult1 = λʳ(x, acc). mult0(x<<1, acc+x)
fun mult0 = λʳ(x, acc). acc
```

and hence (by unfolding, or just by compiling the used-once functions `mul6` to `mul0` into zero-clock functions) to:

```
fun mult13 = λʳ(x, acc).
  acc + x + ((x<<1)<<1) + (((x<<1)<<1)<<1)
```

or equivalently

```
funʳ mult13(x,acc) = acc + x + ((x<<1)<<1) + (((x<<1)<<1)<<1)
```

which now again adheres to the SAFL rules.

In this example at least, once the idea of using `y` as a static (λ^c) parameter had been mooted, the manipulation was forced by the type system.

Finally, let us observe that partial evaluation techniques applied to processors can produce interesting effects. A sequence of frequently occurring software instructions for a processor can be specialised into a single new instruction and the hardware necessary to execute this instruction (hopefully faster) automatically generated.

6 Conclusions and Further Work

We have found SAFL to be surprisingly powerful at describing hardware at a high-level in spite of its meagre features. In particular it seems to be very effective at describing processor design and transformations to adjust their area-time consumption as discussed in Section 4. An important aspect of this is resource-awareness. We can trust the SAFL compiler to optimise code without

altering its gross structure; hence transformations on hardware structure can be seen as SAFL source-to-source transformations.

Another aspect of the same coin is that we would expect to be able to verify mechanically the transformations we make, and indeed to hope that semi-automatic tools can help the user to choose an area-time tradeoff.

At the moment SAFL is a self-contained language which compiles to Verilog. However we could also embed SAFL within VHDL or Verilog as a higher-level behavioural form—the main problem would be restricting access to lower-level details so that SAFL compiler optimisations and SAFL transformations which we have discussed remain valid.

The aspect about which we feel most exposed is that the pure functional call-and-wait-for-result interface provided by SAFL is sometimes too restrictive. Recent work [25] on SAFL+ suggests one possible way forward.

Acknowledgement

This work was supported by (UK) EPSRC grant GR/N64256 "A Resource-Aware Functional Language for Hardware Synthesis"; the second author was also sponsored by AT&T Research Laboratories Cambridge. Phil Endecott, David Greaves and Paul Webster provided helpful comments.

References

1. Van Berkel, K. Handshake Circuits: an Asynchronous Architecture for VLSI Programming. International Series on Parallel Computation, vol. 5. Cambridge University Press, 1993.
2. Bjesse, P., Claessen, K., Sheeran, M. and Singh, S. Lava: Hardware Description in Haskell. Proc. 3rd ACM SIGPLAN International Conference on Functional Programming, 1998.
3. Burstall, R.M. and Darlington, J. A Transformation System for Developing Recursive Programs, JACM 24(1), 1979.
4. Cartwright, R. and Fagan, M. Soft Typing. Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, 1991.
5. Halbwachs, N., Caspi, P., Raymond, P. and Pilaud, D. The Synchronous Dataflow Programming Language Lustre. Proc. IEEE, vol. 79(9). September 1991.
6. Harel, D. and Pnueli, A. On the Development of Reactive Systems. Springer-Verlag NATO ASI Series, Series F, Computer and Systems Sciences, vol. 13, 1985.
7. Hennessy, J.L. and Patterson, D.A. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 1990.
8. Jones, N.D., Gomard, C.K. and Sestoft, P. Partial Evaluation and Automatic Program Generation. Prentice-Hall International Series in Computer Science, 1993.
9. The Jazz Synthesis System: `http://www.exentis.com/jazz`
10. Ku, D. and De Micheli, G. HardwareC—a Language for Hardware Design (version 2.0), Stanford University Technical Report: CSL-TR-90-419, 1990.
11. Li, Y. and Leeser, M. HML, a Novel Hardware Description Language and its Translation to VHDL. IEEE Transactions on VLSI Systems, vol. 8. no. 1. February 2000.

12. Matthews, J., Cook, B. and Launchbury, J. Microprocessor Specification in Hawk. Proc. IEEE International Conference on Computer Languages, 1998.
13. McKay, N and Singh, S. Dynamic Specialisation of XC6200 FPGAs by Parial Evaluation. Lecture Notes in Computer Science: Proc. FPL 1998, vol. 1482, Springer-Verlag, 1998.
14. De Micheli, G. Synthesis and Optimization of Digital Circuits. McGraw-Hill, 1994.
15. Milner, R., Tofte, M., Harper, R. and MacQueen, D. The Definition of Standard ML (Revised). MIT Press, 1997.
16. Moore, S.W., Robinson, P. and Wilcox, S.P. Rotary Pipeline Processors. IEE Part–E, Computers and Digital Techniques, Special Issue on Asynchronous Architectures, 143(5), September 1996.
17. Morison, J.D. and Clarke, A.S. ELLA 2000: A Language for Electronic System Design. Cambridge University Press 1994.
18. Mycroft, A. and Sharp, R.W. The FLaSH Project: Resource-aware Synthesis of Declarative Specifications. Proc. International Workshop on Logic Synthesis 2000.
19. Mycroft, A. and Sharp, R. A Statically Allocated Parallel Functional Language. Lecture Notes in Computer Science: Proc. 27th ICALP, vol. 1853, Springer-Verlag, 2000.
20. Mycroft, A. and Sharp, R. Hardware/Software Co-Design Using Functional Languages. Lecture Notes in Computer Science: Proc. TACAS'01, vol. 2031, Springer-Verlag, March 2001.
21. Nielson, F. and Nielson, H.R. Two Level Semantics and Code Generation. Theoretical Computer Science, 56(1):59-133, 1988.
22. O'Donnell, J.T. Hydra: Hardware Description in a Functional Language using Recursion Equations and High Order Combining Forms, The Fusion of Hardware Design and Verification, G. J. Milne (ed.), North-Holland, 1988.
23. Sharp, R. and Mycroft, A. The FLaSH Compiler: Efficient Circuits from Functional Specifications. AT&T Research Laboratories Cambridge Technical Report tr.2000.3, June 2000.
Available from `http://www.uk.research.att.com`
24. Sharp, R. and Mycroft, A. Soft Scheduling for Hardware. Lecture Notes in Computer Science: Proc. SAS'01, vol. 2126, Springer-Verlag, July 2001.
25. Sharp, R. and Mycroft, A. A Higher-Level Language for Hardware Synthesis. Lecture Notes in Computer Science: Proc. CHARME'01, vol. 2144 (this volume), Springer-Verlag, September 2001.
26. Sheeran, M. muFP, a Language for VLSI Design. Proc. ACM Symp. on LISP and Functional Programming, 1984.