Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Higher-Level Techniques for Hardware Description and Synthesis

Alan Mycroft, Richard Sharp

University of Cambridge
Computer Laboratory
JJ Thomson Avenue
Cambridge CB3 0FD, UK
e-mail: `am@cl.cam.ac.uk`
e-mail: `rws26@cl.cam.ac.uk`

The date of receipt and acceptance will be inserted by the editor

Abstract. The FLaSH (Functional Languages for Synthesising Hardware) system allows a designer to map a high level functional language, SAFL, and its more expressive extension, SAFL+, into hardware. The system has two phases: first we perform architectural exploration by applying a series of semantics-preserving *transformations* to SAFL specifications; then the resulting specification is *compiled* into hardware in a *resource-aware* manner—that is, we map separate functions to separate hardware functional units (functions which are called multiple times become shared functional units).

This article introduces the SAFL language and shows how program transformations on it can explore area-time trade-offs. We then show how the FLaSH compiler compiles SAFL to synchronous hardware and how SAFL transformations can also express hardware/software co-design. As a case study we demonstrate how SAFL transformations allow us to refine a simple specification of a MIPS-style processor into pipelined and superscalar implementations. The superset language SAFL+ (adding process calculi features but retaining many of the design aims) is then described and given semantics both as hardware and as a programming language.

1 Introduction and Background

In 1975 a single Integrated Circuit contained several hundred transistors; by 1980 the number had increased to several thousand. Today, designs fabricated with state-of-the-art VLSI technology often contain several million transistors.

The exponential increase in circuit complexity has forced engineers to adopt higher-level tools. Whereas in the 1970s transistor and gate-level design was the norm, during the 1980s Register Transfer Level (RTL) Hardware Description Languages (HDLs) started to achieve widespread acceptance. Using such languages, designers were able to express circuits as hierarchies of components (such as registers and multiplexers) connected with wires and buses. The advent of this methodology led to a dramatic increase in productivity since, for some classes of design, time consuming place-and-route details could now be automated

More recently, *high-level synthesis* (also referred to as *behavioural synthesis*) has started to have an impact on the hardware design industry. In the last few years commercial tools have appeared enabling high-level, imperative languages (referred to as *behavioural languages* within the hardware community) to be compiled directly to hardware. Although these techniques undoubtedly offer increased levels of abstraction over RTL specification there is still room for even higher-level HDLs, particularly when it comes to specifying interfaces between separate components (see Section 1.1). Since current trends predict that the exponential increase in transistor density will continue throughout the next decade, investigating higher-level tools for hardware description and synthesis will remain an important research area.

The FLaSH project (Functional Languages for Synthesising Hardware) started in 1999 at AT&T Laboratories Cambridge with the aim of addressing such issues. The primary intent was to adopt an aggressively high-level stance with respect to hardware specification. Our aim was to design a system in which:

– the programming language is clean, simple and formally defined;
– low-level implementation details do not become fixed early in the design process;
– programs are susceptible to compiler analysis and optimisation facilitating the automatic synthesis of efficient circuits;
– specifications can be mapped to radically different design styles (e.g. either synchronous hardware, asyn-

chronous hardware, or perhaps even a mixture of both).

We argue that the results of the following sections (covering compilation, co-design and stepwise transformation) vindicate this approach.

1.1 Motivation for Higher Level HDLs

Although existing behavioural HDLs provide high-level primitives for algorithmic description, their support for structuring large designs is often lacking. Many such HDLs (including Behavioural VHDL and Verilog) use *blocks* parameterised over input and output ports as a structuring mechanism. For example, at the top level, a Behavioural Verilog [1] program still consists of `module` declarations and instantiations albeit that the modules themselves contain higher-level constructs such as assignment, sequencing and while-loops.

Experience has shown that the notion of a block is a useful syntactic abstraction, supporting a "define-once, use-many" methodology. However, as a *semantic abstraction* it buys one very little; in particular: (*i*) any part of a block's internals can be exported to its external interface; and (*ii*) inter-block control- and data-flow mechanisms must be coded explicitly on an ad hoc basis.

Point (*i*) results in it being difficult to reason about the global (inter-module) effects of local (intra-module) transformations. For example applying small changes to the local structure of a block (e.g. delaying a value's computation by one cycle) may have dramatic effects on the global behaviour of the program as a whole. We believe point (*ii*) to be particularly serious. Firstly, it leads to low-level implementation details scattered throughout a program—e.g. the definition of explicit control signals used to sequence operations in separate modules, or (arguably even worse) reliance on unwritten inter-module timing assumptions. Secondly, it inhibits compiler analysis: since inter-block control- and data-flow mechanisms are coded explicitly it is difficult for a compiler to infer how separate blocks relate to each other. Thus, scope for global analysis and optimisation is very limited. Based on these arguments, we argue that structural blocks are not a high-level abstraction mechanism.

1.2 The FLaSH Project

The core of the FLaSH project concerns a behavioural HDL called SAFL (Statically Allocated Functional Language) and its associated tool chain.

SAFL is designed around the observation that many of the problems associated with structural blocks (see above) can be alleviated by structuring code as a series of function definitions. Firstly, the properties of functions make it easier to reason about the effects of program transformations; as a result, source-level program transformation becomes a viable technique for architectural exploration. Secondly, the "invoke and wait for result" interface provided by functions removes the programmer from the burden of explicitly specifying ad hoc inter-module control- and data-flow mechanisms. As well as making the code shorter and easier to read, the implicit control-flow information encapsulated in the function-abstraction increases the scope for global compiler analysis and optimisation.

We have implemented an optimising compiler which translates SAFL into synchronous hardware (via hierarchical RTL Verilog) and the system has been tested on realistic designs, including a commercial processor[1] and a DES encrypter/decrypter [31]. Ultimately, the compiler will be accompanied by a SAFL *transformer*: a tool which helps engineers perform architectural exploration by applying source-to-source transformations to SAFL specifications. Although the transformer tool is still in development, we have demonstrated various SAFL-level transformations which embody a wide range of implementation trade-offs (e.g. functional unit duplication versus sharing [27] and hardware/software co-design [28]). 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.

Whilst SAFL is an excellent vehicle for high-level synthesis research we recognise that it is not expressive enough for industrial hardware description. In particular the facility for I/O is lacking and, in some circumstances, the "call and wait for result" interface provided by the function model is too restrictive. To address these issues we have developed a language, SAFL+, which extends SAFL with process-calculus features including synchronous channels and channel-passing in the style of the π-calculus [22]. The incorporation of channel-passing allows a style of programming which strikes a well-chosen balance between the flexibility of structural blocks and the analysability of functions. We have extended our SAFL tools to deal with SAFL+ demonstrating that our analysis and compilation techniques for SAFL are applicable to realistic hardware description languages.

1.3 Structure of the Article

This article draws together various strands from our conference papers: examining the theoretical basis of static allocation with parallelism in the SAFL language [27] (Section 2), describing the FLaSH compiler [32] (Section 3), examining hardware/software co-design[28] (Section 4) and demonstrating how the expressivity of SAFL

[1] We implemented the XAP processor designed by Cambridge Consultants: `http://www.camcon.co.uk`; we did not implement the `SIF` instruction. The SAFL specification of the XAP compiles into hardware which runs as fast as the hand-designed version and contains less than 4000 (2-input equivalent) gates—the same order of magnitude of the hand-designed version.

can be increased by extending the language with process-calculus features [34] (Section 6). A case study is presented (Section 5) in which a simple MIPS-style CPU is defined in SAFL. Program transformations are applied to refine the specification into pipelined and superscalar implementations.

Each section of the article summarises a different aspect of our work and hence (after Section 2, which defines terminology used in the remainder of the paper) they can largely be read independently of one another. To maintain a sense of coherence throughout the document we include summaries and conclusions at the end of each section. Finally, Section 7 considers related work and Section 8 outlines directions for future research.

2 The SAFL Language

SAFL has syntactic categories e (term) and p (program). Let v range over a set of integer 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. We can now define the abstract-syntax of SAFL programs, p, as follows:

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

It is sometimes convenient to extend this syntax slightly. In later examples 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). Programs have a distinguished function `main` (normally f_n) which represents an external world interface—at the hardware level it accepts values on an input port and may later produce a value on an output port.

SAFL is a call-by-value language in which independent sub-expressions are evaluated in parallel. Thus, in the form:

$$\texttt{let } \vec{x} = (e_1, \ldots, e_k) \texttt{ in } e_0$$

and in calls:

$$f(e_1, \ldots, e_k) \quad \text{or} \quad a(e_1, \ldots, e_k)$$

all the e_i $(1 \le i \le k)$ are to evaluated concurrently. In the conditional:

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

we first evaluate e_1; depending on the result of e_1, one of e_2 or e_3 is subsequently evaluated.

Note that parallel execution is essential for efficient hardware implementation where, in contrast to software, fine-grained concurrency is available for free. The functional properties of SAFL are particularly useful in this respect: referential transparency allows our compiler to generate highly parallel hardware.

2.1 Static Allocation

We say that SAFL is *statically allocatable*, meaning that we can allocate all storage required for a SAFL program at compile-time. Our desire for static allocation is motivated by an observation that dynamically-allocated storage does not map well onto silicon: an addressable global store leads to a von Neumann bottleneck which inhibits the natural parallelism of a circuit.

In order to achieve static allocability we impose the restriction that all recursive calls must be tail-recursive. This is formalised as a well-formedness check: define the *tailcall contexts*, $\mathcal{TC}$ by

$$\begin{aligned} \mathcal{TC} ::= & \ [\,] \\ & \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } \mathcal{TC} \\ & \mid \texttt{if } e_1 \texttt{ then } \mathcal{TC} \texttt{ else } e_3 \\ & \mid \texttt{let } \vec{x} = \vec{e} \texttt{ in } \mathcal{TC} \end{aligned}$$

The well-formedness condition is then that, for every user-function application $f^i(\vec{e})$ within function definition $f^j(\vec{x}) = e_j$, we have that:

$$i < j \vee (i = j \wedge \exists (C \in \mathcal{TC}).e_j = C[f^i(\vec{e})])$$

The first part $(i < j)$ is merely static scoping (definitions are in scope if previously declared) while the second part says that if the call is recursive $(i = j)$ then it must be in tailcall context. (In previous work we have shown how the language can be extended to allow mutual recursion whilst still maintaining static allocation [27].)

To emphasise the hardware connection we define the *area* of a SAFL program to be the total space required for its execution. Due to static allocation we see that *area* is $O(\textit{length of program})$.

2.2 Integrating with External Hardware Components

Although the SAFL language in itself is powerful enough to specify many hardware designs in full there are often circumstances where a SAFL hardware design must interface with external hardware components. In the SAFL language, facility is provided to access externally defined hardware via a function call interface. The signature of external functions is given using the `extern` keyword. For example, one may declare an interface to an external RAM block as follows:

```
extern mem(addr:16, data:32, wr:1) : 32;
```

where the $:\langle n \rangle$ annotations are used to specify the bit-widths of the argument and result types (see Section 2.4).

Whilst the details of the interfacing mechanism are not described in this article, it is worth noting that external calls may be side-effecting. We take an ML-style view of side-effects: since SAFL is a strict language it is easy to reason about where side-effects will occur in a program's execution. Of course, we intend that programmers write as much as possible of their specification in SAFL, only relying on external calls when absolutely necessary.

2.3 Semantics

At an abstract level, the semantics of SAFL are straightforward. Indeed, a so-called big-step Structural Operational Semantics of the form $\langle S, e \rangle \Downarrow v$ (where S gives values to free variables of e) for SAFL can be given by only seven inference rules. However, although this semantics is able to determine the result of a SAFL program's execution, it is does not model many of the important (intensional) properties of the language such as concurrency or calls to external (possibly side-effecting) functions. These issues are dealt with formally in Section 6 where a more fine-grained semantics based on the *Chemical Abstract Machine* [3] is presented; this covers both SAFL and SAFL+.

2.4 Types in SAFL

For most of the examples in this article, we will omit type information. However, the types of all variables must clearly be known to the compiler in order that the appropriate widths of buses and registers can be generated. In practice, all SAFL variables are explicitly annotated with a type bit_n. SAFL's concrete syntax uses the form `x`$:\langle n \rangle$ to specify the bit-width, $\langle n \rangle$, of a variable, `x`, at the point of `x`'s definition. All constants, v, also have a type (either explicitly given or otherwise inferred from their value). Built-in functions, such as addition and concatenation, may have a family of types (e.g. $bit_n * bit_n \mapsto bit_n$ and $bit_m * bit_n \mapsto bit_{m+n}$). User-defined functions also require the type of result to be provided when it cannot be inferred from the type of result.

The value, `()`, of width 0 plays a special rôle and has type *unit*. We adopt the convention that all functions return a single result; side-effecting (`extern`) functions which do not need to return a value can return `()` instead.

2.5 Resource Awareness

The static allocation properties of SAFL allow our compiler to enforce a direct mapping between a function definition, $f(\vec{x}) = e$, and a hardware block, H_f, with output port, P_f, consisting of: (i) a fixed amount of storage (registers holding values of the arguments $\vec{x}$); and (ii) a circuit to compute e to P_f.

We say that our SAFL compiler is *resource aware* since each function declaration at the source-level is translated into a *single* resource at the hardware level. In this framework, multiple calls to a function f corresponds directly to sharing the resource H_f at the hardware level. We believe that resource awareness offers a number of benefits:

1. The function-resource correspondence allows SAFL to express *both* logical specification *and* system-level hardware structure without requiring any extra language features.
2. A designer is able to visualise how a SAFL program will be structured at the hardware-level. This intuitive understanding prevents one from having to "second guess" the compiler.
3. Source-to-source program transformation becomes a powerful technique. For example, Burstall and Darlington's (source-level) fold/unfold transformation [4] allows exploration of (implementation-level) tradeoffs between resource sharing and duplication—see Section 2.6.

At first sight there appears to be contention between our goal to develop a truly high-level specification language on the one hand, and the decision to enforce a 1-1 correspondence between function definitions and hardware resources on the other. Recall that one of our initial requirements (see Section 1) was that our high-level hardware specification language should not fix low-level implementation details. Thus, at least on the surface, it appears that by advocating resource-awareness we have violated our own design criteria.

These apparently conflicting views are resolved via an extra program-transformation step in the SAFL-to-silicon design-flow. We intend that designers initially write SAFL specifications as clearly as possible, without considering any implementation-level issues. A source-level program transformation phase is then used to refine a given specification towards a suitable implementation whilst preserving its logical semantics. Hence SAFL frees a designer from low-level concerns in the initial phases of development but is still expressive enough to encode important implementation-level choices later in the design process.

Hardware designers frequently complain that it is difficult to visualise the circuits that will be generated by black-box high-level synthesis tools. However, the desire for an intuitive understanding of the compilation process must be traded off against the benefits of automatic analysis and optimisation: a compiler which performs significant optimisation is necessarily harder to predict than one which performs only syntax-directed translation. A strong argument in favour of resource awareness is that it explicitly defines the boundary between human specification and compiler optimisation—a SAFL program fixes the top-level circuit structure but leaves a compiler free to optimise function-internals and inter-resource communication structures.

It is important to observe the interaction between parallel execution and resource-awareness. Since our compiler generates concurrent hardware we have to be careful to ensure that multiple accesses to a shared hardware resource will not occur simultaneously. Section 3.1 describes how shared resources can be protected with arbiters. We have developed a whole-program analysis which allows redundant arbiters to be optimised away (see Section 3.3).

2.6 Transformations in SAFL

Source-to-source transformation of SAFL provides a powerful technique for design-space exploration. We believe SAFL has two major advantages over conventional HDLs when it comes to source-level transformation:

1. 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[2]).
2. The resource-aware properties of SAFL give program 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).

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 [4] 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.

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.

[2] Assuming of course that we are not dealing with `extern` functions.

```
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')
```

Fig. 1. Unfolded `mult` function

Note that fold/unfold allows us to do more than resource/duplication sharing trade-offs; 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 the code shown in Fig. 1. When synthesised, this version uses almost twice as much area and takes half as many clock cycles. (Note that the clock speed slows down since the critical path becomes longer).

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:

```
fun f(x,y) = if x=0 then y else f(x-5, y+1)
fun main(a,b,c) =
  if a=0 then b
  else if ... then main(a', f(k,b), c')
  else main(a',b',c')
```

In imperative programming terminology loop collapsing converts 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. Our functional equivalent involves combining two (possibly tail recursive) function definitions, f and g, into a single function definition which takes the disjoint union of f and g's formal parameters as well as an extra argument which specifies whether f or g's body should be executed. Applying this transformation yields the code shown in Fig. 2. 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

```
fun main(inner,x,a,b,c) =
  if inner then
    if x=0 then main(0,x,a,b,c)
                  (* exit f() *)
           else main(1,x-1,a,y',c)
                  (* another f 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  *)
```

Fig. 2. Loop collapsing example

store arguments etc.—see Section 3). 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`.)

We note one final transformation: Partial Evaluation [15] techniques can also be used to specialise SAFL programs (and hence hardware) when some of the inputs to the program are known. This is explored further in our CHARME'01 invited paper [26].

2.7 The SAFL Language: Discussion and Conclusions

We are often asked why SAFL uses eager evaluation, especially given that Haskell-based tools (e.g. Lava and Hawk) are lazy. Firstly, the applications are different: SAFL describes computations whereas Lava describes hardware structure. Even given this, there remains a choice: lazy evaluation can be seen as a form of compute-on-demand hardware implementation whereas eager evaluation corresponds to a data-flow-like input-driven implementation. We chose eager evaluation since classical lazy evaluation would be 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[3]. 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'.

Another frequently asked question is why we chose a functional style, rather than (say) an imperative languages with assignment `while`-loops, and the like. After all, a single tail-recursive function can be seen as a non-recursive function with a `while`-loop and assignment to parameters. 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 resource-awareness is compromised; and (*iii*) concurrent function calls require a `par` construct to implement them as concurrent `while`-loops thus adding complexity.

In summary the SAFL language provides a simple, clean and powerful framework in which to investigate concepts in hardware description and synthesis. The functional nature of the language and its notion of resource awareness make source-level program transformation a powerful technique for architectural exploration. In contrast to HDLs such as VHDL/Verilog, where low-level design details become fixed early in the specification process, the high-level properties of SAFL allow us to make sweeping architectural changes late in the design flow. We have already seen that program transformation can be used to explore various area-time trade-offs including resource sharing/duplication. In Section 4 we take these techniques a stage further, describing a SAFL-level program transformation which allows an engineer to explore complex hardware/software partitionings.

3 The FLaSH Compiler

The implementation of the FLaSH compiler, which translates SAFL to hardware, has been a central area of our research. From a synthesis perspective, a key advantage of the SAFL language is that its high-level properties facilitate *global* analysis and optimisation. In contrast, synthesis systems for languages such as Verilog are usually restricted to local optimisation at the behavioural level—it is too difficult to perform complex static analysis across `module` boundaries.

We have implemented two global analyses as part of the FLaSH compiler:

– *Parallel Conflict Analysis* (see Section 3.3.1) detects whether multiple calls to the same function may occur simultaneously at run-time. The analysis is used to drive the FLaSH compiler's novel scheduling technique.

[3] While this can sometimes be detected and optimised (strictness analysis) the formal problem is undecidable and so poses a difficulty for language design.

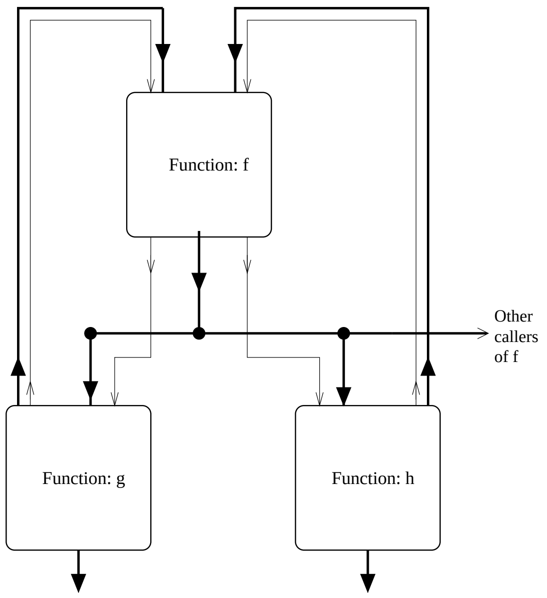


Fig. 3. Function Connectivity

– *Data Validity Analysis* (see Section 3.3.2) determines how long the output of a shared resource will remain valid. This information is used to minimise the number of temporary registers required to latch the results of function calls.

Note that although the SAFL language is designed to be architecture neutral (see Section 3.4) our compiler can currently only generate synchronous hardware, targeting structural Verilog as its object language. (We use standard industrial RTL compilers to map the generated Verilog to silicon.)

Having lexed and parsed SAFL source, the FLaSH compiler performs a number of analyses (such as type checking and parallel conflict analysis) at the abstract-syntax level. The abstract syntax tree is then translated into graph-based intermediate code where further analyses (including data validity analysis) and optimising transformations are applied. The full gory details of the compiler's internals are available as a technical report [32]. For space constraints we can only provide a summary here. We start by describing a naïve translation of SAFL to synchronous hardware (Section 3.1). The optimisations employed in the FLaSH compiler are outlined in Section 3.3.

3.1 Naïve Translation to Synchronous Hardware

Since we only consider synchronous hardware here we assume the availability global clock, *clk*. 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*-tick. We adopt the protocol/convention that signal wires (e.g. *request* and *ready*) are held high for exactly one cycle.

The SAFL compiler translates each source-level function definition

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

into 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_f (output wires which reflect the value of e);
– a *ready* signal which is asserted when P_f is valid.

Fig. 3 shows graphically how functional units are connected together. In this section, and throughout the remainder of this document, we adopt the convention that thin lines represent control signals and thick lines represent data wires. The figure shows a functional-unit H_f which is shared between H_g and H_h. Notice how H_f's data output is shared, but the control structure is duplicated on a per call basis.

To perform a call to resource H_f the caller places the argument values on its data input into H_f before triggering a call event on the corresponding control input. Some point later, when H_f has finished computing, the result of the call is placed on H_f's shared data-output and an event is generated on the corresponding control output.

The internals of a functional-unit are shown in Fig. 4. For each (non-recursive) call to a function, f, a separate control/data input-pair is fed into the functional-unit H_f at the circuit-level. Let us first consider the control path. If the function is shared between multiple call-sites then control wires are passed through an arbiter (priority-encoder) which ensures that only one call is dealt with at a time[4]. Note that the use of dynamic arbitration to protect shared resources is a novel aspect of our research; other high-level synthesis generate a fixed static schedule at compile-time. (We describe some of the benefits of this scheduling methodology in Section 3.3.1.)

Having been through the arbiter, the control inputs are fed into the *External Call Control Unit* (ECCU—see Fig. 5), which:

1. remembers which of the control inputs triggered the call;
2. invokes the function body expression by generating an event on the `FB_invoke` wire (we consider the internals of the function's body circuit in Section 3.2);
3. waits for completion (signalled by the `FB_finished` wire); and finally
4. generates an event on the corresponding control output, signalling to the caller that the result is waiting on H_f's shared data output.

[4] We use a static analysis to optimise away redundant arbiters. See Section 3.3.1.

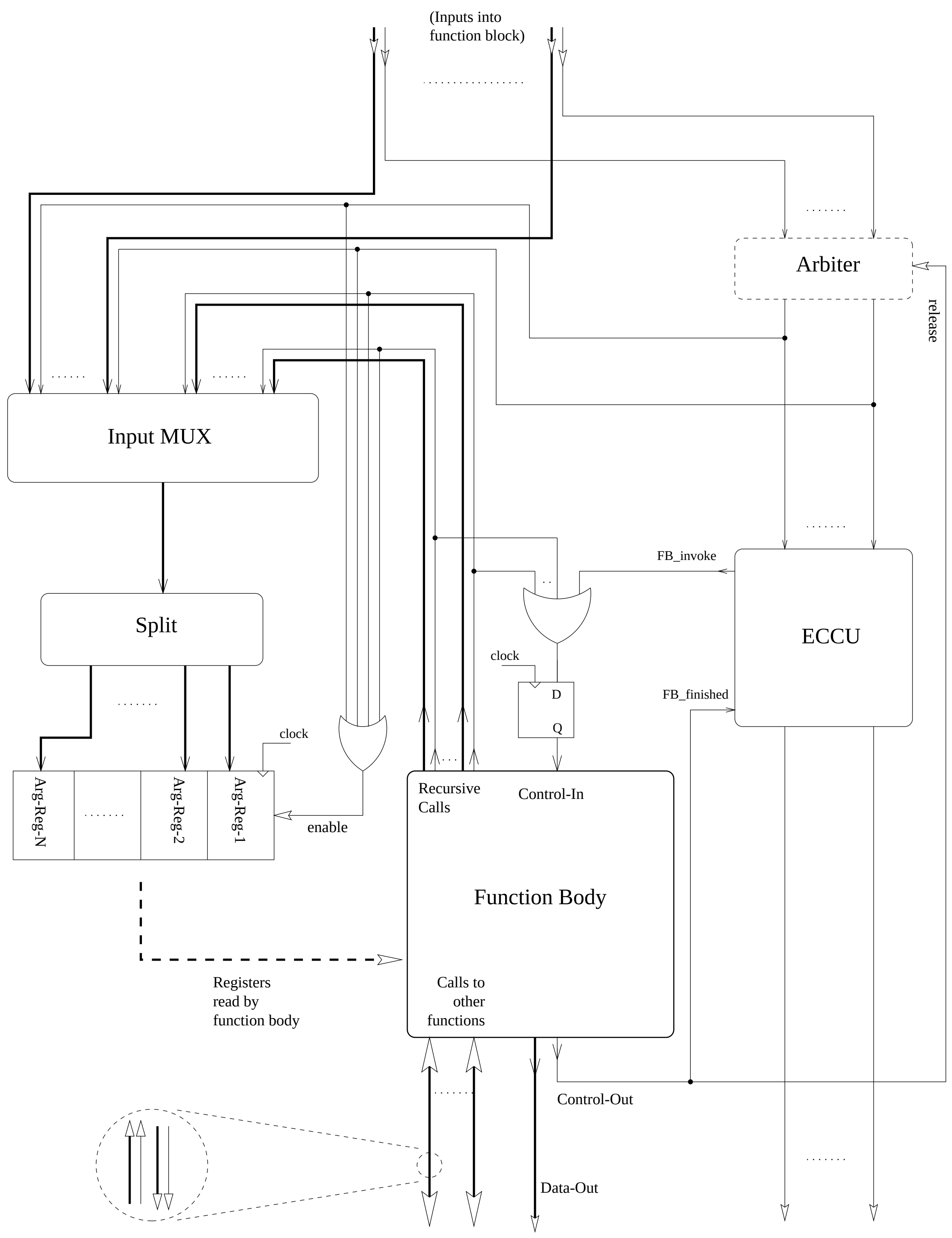


Fig. 4. A Block Diagram of a Hardware Functional-Unit

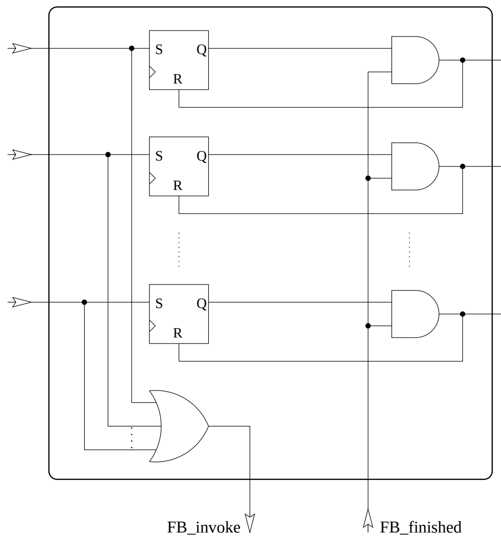


Fig. 5. The Design of the External Call Control Unit (ECCU)

Now let us consider the data-path. The data inputs are fed into a multiplexor which uses the corresponding control inputs as select lines. The selected data input is latched into the argument registers. (Obviously, the *splitter* is somewhat arbitrary; it is included in the diagram to emphasise that multiple arguments are all placed on the single data-input). Recursive calls feed back into the multiplexor to create loops, re-triggering the body expression as they do so. (Recall that SAFL enforces the restriction that all recursive calls are tail-recursive.) The D-type flip-flop is used to insert a 1-cycle delay onto the control path to match the 1-cycle delay of latching the arguments into the registers on the data-path.

The function body expression contains connections to the other functional-units that it calls. These connections are the ones marked "calls to other functions" in Fig. 4 and are seen in context in Fig. 3.

3.2 Translating SAFL expressions

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

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

variables x: the treatment of variables is similar to constants save that the output port is connected to wires that reflect the value of the variable (e.g. in the case of a formal parameter the output port would be connected to one of the enclosing function's argument registers).

if e_1 then e_2 else e_3: first e_1 is computed; when it signals *ready* its boolean output is used to route a *request* signal either to e_2 or to e_3 and also to route 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 (see below).

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 $g(\vec{e})$: First the $\vec{e}$ are computed concurrently. If the call is self- (tail-) recursive then it is implemented as an internal feedback loop (see Fig. 4). Otherwise the outputs of the argument expressions are connected to the hardware block corresponding to g; when all arguments are ready the *request* for g is activated.

One subtlety in this compilation scheme concerns the values returned by shared functional units 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 $P_{\mathtt{h}}$ 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 $P_{\mathtt{k}}$ will be lost on the subsequent computation of `k(x+1)`. Therefore we insert a clocked *permanisor* register within the hardware functional unit, $H_{\mathtt{f}}$ 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.

The process of adding permanisor registers at the output of resources, like `k` above, allows us to avoid latching variables defined by `let` declarations—permanisors have already ensured that signals of `let`-bound variables remain valid as long as is needed.

3.3 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.

3.3.1 Removing Arbiters (Soft Scheduling)

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 (*Parallel Conflict Analysis* [33]) remove redundant scheduling logic. We call the SAFL approach *soft scheduling* to highlight the analogy with Soft Typing [6]: the aim is to retain the flexibility of dynamic scheduling whilst using static analysis to remove as many dynamic checks as possible. In [33] 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 it provides a convenient compromise between static and dynamic scheduling, allowing the programmer to choose which to adopt. For example, compiling `f(g(4))+f(k(5))` will generate an arbiter to serialise access to the shared resource $H_{\mathtt{f}}$ *dynamically.* Alternatively we can use a `let`-declaration to specify an ordering *statically.* The circuit corresponding to `let x=f(g(4)) in x+f(k(5))` does not require dynamic arbitration; we have specified a static order of access to $H_{\mathtt{f}}$. Note that program transformation can be used to explore static vs. dynamic scheduling trade-offs. In fact, we can represent the different static scheduling algorithms applied by conventional high-level synthesis systems (e.g. ASAP, List Scheduling etc. [21]) as source-level transformations in SAFL.

3.3.2 Register Placement

In our naïve translation to hardware (previous section) we noted that a caller latches the result of a call into a permanising register. 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. We have implemented a parallel data-flow analysis (called *Data Validity Analysis*) which allows us to eliminate unnecessary permanising registers [32].

3.3.3 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`.

3.3.4 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 version of the SAFL compiler [32] 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[5], 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).

3.4 Translation to Other Design Styles

At the moment the FLaSH compiler only generates synchronous hardware. We have not yet implemented back-ends to target different design-styles. However, since we intend the translation of SAFL to other styles of hardware (particularly asynchronous hardware) to become a core area of the FLaSH project in the near future we felt it was appropriate to include a brief discussion of the topic here.

In the design of the SAFL language we were careful not to favour the description of any particular circuit design paradigm. We say that SAFL is *architecture neutral* to mean that it abstracts a number of implementation styles. The reason why SAFL offers architecture neutrality (whereas languages such as Verilog/VHDL do not) is simple: the SAFL language prevents a designer from

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

making *any* assumptions about the underlying hardware.

3.4.1 Generating Asynchronous Hardware

Note that the design philosophy outlined in Section 3.1 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.

3.4.2 Generating "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*) 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.

3.5 Compiling SAFL: Discussion and Conclusions

The FLaSH compiler has demonstrated that it is possible to automatically generate efficient hardware from very high-level specifications, suggesting that automatic synthesis of a behavioural functional HDL is a viable technique for hardware construction. Furthermore, our "soft scheduling" technique, which arose as a direct result of implementing the FLaSH compiler, offers the potential to improve synthesis tools for existing HDLs (such as HardwareC and Tangram) both in terms of expressivity and efficiency [33].

A common opinion amongst the hardware community is "the higher-level the HDL, the less efficient the generated circuit". Contrary to this popular belief we have demonstrated that the high-level features of a well chosen HDL can actually *increase* the efficiency of generated hardware: it is precisely the high-level properties of SAFL which allow our compiler to perform effective global analysis and optimisation. Extending this argument, we observe that other high-level properties of SAFL also contribute to low-level circuit efficiency. For example, the functional nature of the language lends itself to naturally parallel implementation and our use of immutable (`let`) declarations maps well onto the dataflow-nature of circuits (i.e. not every variable requires a register).

4 Hardware/Software Co-Design

In this section we show that source-level transformation of a SAFL specification allows one to make *radical*[6] changes to a circuit's implementation in a systematic way. The example we choose involves representing hardware/software as a source-to-source transformation at the SAFL level and summarises one of our previous publications [28]. In fact we go one step further than traditional co-design since as well as partitioning a specification into hardware and software parts our transformation procedure can also synthesise an architecture tailored specifically for executing the software part. This architecture consists of any number of interconnected heterogeneous processors. The technique offers engineers and designers a number of potential benefits:

– Synthesising an architecture specifically to execute a known piece of software can offer significant advantages over a fixed architecture [30].
– The ability to synthesise multiple processors allows a wide range of area-time trade-offs to be explored. Not only does hardware/software partitioning affect the area-time position of the final design, but the number of processors synthesised to execute the software part is also significant: increasing the number of processors pushes the area up whilst potentially reducing execution time (as the processors can operate in parallel).
– Resource-awareness allows a SAFL specification to represent shared resources. This increases the power of our partitioning transformation since, for example, multiple processors can access the same hardware resource (see Fig. 6 for an example).

The key insight exploited by our approach is that a processor is just another SAFL function definition, as is an

[6] i.e. well beyond the scope of those described in Section 2.6

instruction memory ROM (for various concrete examples of describing processors in SAFL see Section 5). Hence our paradigm allows processors, hardware functions and software functions to be intermixed freely.

4.1 Technical Details

The first step in the partitioning transformation is to define a partitioning function, π, specifying which SAFL functions are to be implemented directly in hardware and which are to be mapped to a processor for software execution. We do not deal with automated partitioning here; we assume that π is supplied by the user.

Let $\mathcal{M}$ be the set of *processor instances* (processors encoded as SAFL functions) used in the final design. We define a (partial) partitioning function

$$\pi : \textit{SAFL function name} \rightharpoonup \mathcal{M}$$

mapping the function definitions in our SAFL specification onto processors in $\mathcal{M}$. $\pi(f)$ is the processor on which function f is to be implemented. If $f \notin \mathit{Dom}(\pi)$ then we realise f in hardware, otherwise we say that f is *located* on machine $\pi(f)$. Note that multiple functions can be mapped to the same processor.

We extend π to a transformation function

$$\hat{\pi} : \textit{SAFL Program} \rightarrow \textit{SAFL Program}$$

such that given a SAFL program, p, $\hat{\pi}(p)$ is another SAFL program which respects the partitioning function π. Fig. 6 shows the effect of a partitioning transformation, $\hat{\pi}$, where

$$\mathcal{M} = \{M_1, M_2\}; \text{ and}$$
$$\pi = \{(f, M_1),\ (h, M_1),\ (i, M_2),\ (j, M_2)\}$$

In this example we see that g and k are implemented in hardware since $g, k \notin \mathit{Dom}(\pi)$. $\hat{\pi}(p)$ contains function definitions: M_1, M_2, IM_1, IM_2, g and k where M_1 and M_2 are processor instances and IM_1 and IM_2 are instruction memories (see Section 4.2).

4.2 Processors: Templates and Instances

The processor *template*, PT, is an abstract model of a processor parameterised on the code it will have to execute. Given a machine code program, p, $PT\langle p\rangle$ is a processor *instance*: a SAFL function encoding a version of the processor specialised for executing p. (Our notion of a template is similar to a VHDL *generic*.)

A processor instance, $PI_i \in \mathcal{M}$, is a SAFL function of the form:

```
fun $PI_i$(a$_1$, ..., a$_{n\_args(PI_i)}$, PC, ...) = ...
     where $n\_args(m) = max(\{arity(f) \mid \pi(f) = m\})$
```

Arguments $\mathtt{a}_1$, ..., $\mathtt{a}_{n\_args(PI_i)}$ are used to receive arguments of functions located on PI_i. `PC` holds the program counter. Other arguments depend on the type of the processor: for example, in the case of a stack machine an argument may be used to hold the stack pointer; in the case of a register machine arguments may be used to hold the values of registers.

Each processor instance is associated with an instruction memory function, IM_i of the form:

```
fun $IM_i$(address) =
   case address of 0 => instruction_0
                 | 1 => instruction_1
                      ... etc.
```

PI_i calls IM_i(`PC`) to load instructions for execution.

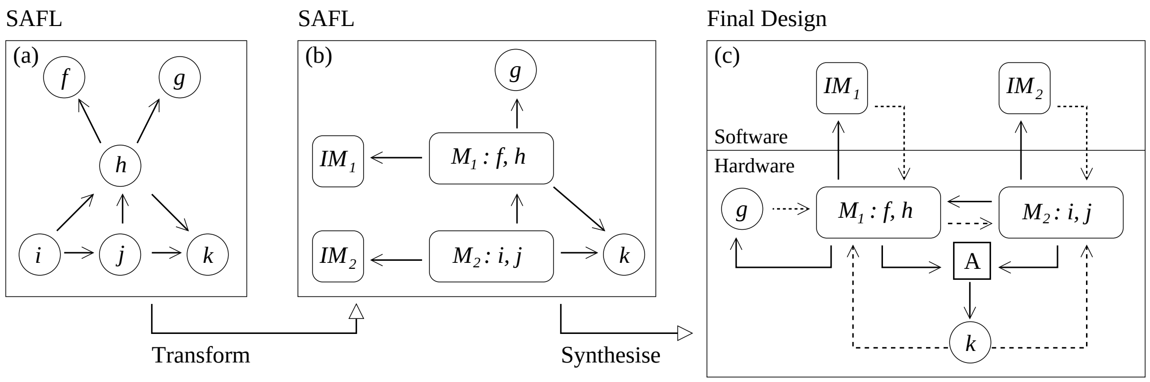
As well as the usual instructions for arithmetic computation and control-flow, our processors have a family of instructions, $\mathtt{Call\_Ext}_f$, which are used to invoke hardware blocks defined by SAFL function f, one instruction per (used) function. The co-design of hardware and software means that instructions and ALU operations are only added to PI_i if they appear in IM_i. Parameterising the template in this way can considerably reduce the area of the final design since we remove redundant logic in each processor instance. We can consider many other areas of parameterisation. For example we can adjust the op-code width and assign op-codes to minimise instruction-decoding delay [30].

4.3 Compilation to Machine Code

The details of the compilation of SAFL to machine code depends on the chosen style of processor. However, irrespective of the processor architecture in question, special care must be taken when compiling function definitions. Suppose function, g, is in software ($g \in \mathit{Dom}(\pi)$) and calls function f. The code generated for the call depends on the location of f relative to g. There are three cases to consider:

1. If f and g are both implemented in software on the same machine ($f \in \mathit{Dom}(\pi) \wedge \pi(f) = \pi(g)$) the generated code must set up f's arguments (e.g. load them into registers or push them onto the stack etc.) and execute a local branch to f's entry point.
2. If f is implemented in hardware ($f \notin \mathit{Dom}(\pi)$) then we set up f's arguments and invoke the hardware resource corresponding to f by means of a $\mathtt{Call\_Ext}_f$ instruction.
3. If f and g are both implemented in software but on different machines ($f, g \in \mathit{Dom}(\pi) \wedge \pi(f) \neq \pi(g)$) then g needs to invoke $\pi(f)$ (the machine on which f is located). This is effected by a $\mathtt{Call\_Ext}_{\pi(f)}$ instruction.

There are a number of other subtleties which, due to space constraints, are not discussed here. In [28] we present the low-level details of a compilation function



Partitioning: (a) shows the call-graph of a SAFL specification, p; (b) shows the call-graph of $\hat{\pi}(p)$, where $\pi = \{(f, M_1), (h, M_1), (i, M_2), (j, M_2)\}$. IM_1 and IM_2 are instruction memory functions (see Section 4.2); (c) shows the structure of the final circuit after compilation. The box marked 'A' represents an arbiter (inserted automatically by the SAFL compiler) protecting shared resource k; the bold arrows represent calls, the dotted arrows represent return values.

Fig. 6. A diagrammatic view of the partitioning transformation

from SAFL to stack machine code. In order for the translation to work correctly we require two distinct entry points for each compiled function: the *internal* entry point is used when f is invoked internally (i.e. by means of a local branch). The *external* entry point is used when f is invoked externally (i.e. via a call to $\pi(f)$, the machine on which f is implemented). In this latter case, we simply execute k `PushA` instructions to push f's arguments onto the stack before jumping to f's internal entry point.

4.4 The Partitioning Transformation

For expository purposes let us assume for the moment that all processors are the same type. This allows us to use a single compilation function, $[\![\cdot]\!]$, which translates SAFL function definitions into machine code. (Section 4.6 shows how we can extend the transformation to cope with a network of heterogenous processors.) The details of the partitioning transformation, $\hat{\pi}$, are as follows:

Let p be the SAFL program we wish to transform using π. Let f be a SAFL function in p with definition d_f of the form

$$\texttt{fun}\ f(x_1, \ldots, x_k) = e$$

We construct a partitioned program $\hat{\pi}(p)$ from p as follows:

1. For each function definition $d_f \in p$ to be mapped to hardware (i.e. $f \notin Dom(\pi)$) create a variant in $\hat{\pi}(p)$ which is as d_f but for each call, $g(e_1, \ldots, e_k)$:

   If $g \in Dom(\pi)$ then replace the call $g(\vec{e})$ with a call:

$$m(e_1, \ldots, e_k,\ \underbrace{0, \ldots, 0,}_{n\_args(m)-k}\ g_{entry}, \ldots)$$

   where $m = \pi(g)$, the processor instance on which g is located. The 0's are used to pad out the number of arguments to the appropriate length. The program counter is set to g's entry point (the offset of g's code in IM_m). Other arguments depend on the style of processor being used—for example, in the case of a stack machine [28] we initialise the stack pointer to 0.
2. For each $m \in \mathcal{M}$:
   (a) Compile instruction sequences for functions located on m:

$$Code_m = \{[\![d_f]\!] \mid \pi(f) = m\}$$

   (b) Generate *machine code* for m, $MCode_m$, by resolving symbols in $Code_m$, assigning opcodes and converting into binary representation.
   (c) Generate an *instruction memory* for m by adding a function definition, IM_m, to $\hat{\pi}(p)$ of the form:
```
fun IMm(address) =
   case address of 0 => instruction_0
                 | 1 => instruction_1
                        ... etc.
```
   where each $\texttt{instruction}\_i$ is taken from $MCode_m$.
   (d) Generate a processor machine instance, $PI\langle Code_m\rangle$ and append it to $\hat{\pi}(p)$.

For each $m \in \mathcal{M}$, $\hat{\pi}(p)$ contains a corresponding processor instance and instruction memory function. When $\hat{\pi}(p)$ is compiled to hardware resource-awareness ensures that each processor definition function becomes a single processor and each instruction memory function becomes a single instruction memory. The remaining functions in $\hat{\pi}(p)$ are mapped to hardware resources as required. Function calls are synthesised into optimised communication paths between the hardware resources (see Fig. 6c).

4.5 Validity of Partitioning Functions

The problem we address in this section is that the SAFL rule requiring all (mutual) recursion to be tail-recursion fails to be preserved under partitioning. This is a technical, rather than central, issue.

In general, a partitioning function, π, may transform a valid SAFL program, p, into an invalid SAFL program, $\hat{\pi}(p)$, which does not satisfy the recursion restrictions for static allocation (see Section 2). For example consider the following program, p_{bad}:

```
fun f(x) = x+1
fun g(x) = f(x)+2
fun h(x) = g(x+3)
```

Partitioning p_{bad} with $\pi = \{(\texttt{f},\texttt{PI}),\ (\texttt{h},\texttt{PI})\}$ yields a new program, $\hat{\pi}(p_{bad})$, of the form:

```
fun IM(PC) = ...
fun PI(x,PC,...) = ... g(t) ...
fun g(x) = PI(x, ⟨f_entry⟩, 0) + 2;
```

$\hat{\pi}(p_{bad})$ has invalid recursion between `g` and `PI`. The problem is that the call to `PI` in the body of `g` is part of a mutually-recursive cycle and is not in tail-context.

We therefore require a restriction on partitions π to ensure that if p is a valid SAFL program then $\hat{\pi}(p)$ will also be a valid SAFL program. For the purposes of this article we give the following sufficient condition:

π is a valid partition with respect to SAFL program, p, iff all cycles occurring the call graph of $\hat{\pi}(p)$ already exist in the call graph of p, with the exception of self-cycles generated by direct tail-recursion.

Thus, in particular, new functions in $\hat{\pi}(p)$—i.e. processor instances and their instruction memories—must not have mutual recursion with any other functions.

4.6 Dealing with Heterogeneous Processors

In general, designs often consist of multiple communicating processors chosen to reflect various cost and performance constraints. Our framework can be extended to handle heterogeneous processors as follows:

Let *Templates* be a set of processor templates.

Let *Compilers* be a set of compilers from SAFL to machine code for processor templates.

As part of the transformation process, the user now specifies two extra functions:

$$\delta : \mathcal{M} \rightarrow \mathit{Templates}$$
$$\tau : \mathcal{M} \rightarrow \mathit{Compilers}$$

δ maps each processor instance, $m \in \mathcal{M}$, onto a SAFL processor template and τ maps each $m \in \mathcal{M}$ onto an associated compiler. We then modify the transformation procedure described in Section 4.4 to generate a partitioned program, $\hat{\pi}_{\delta,\tau}(p)$ as follows: for each $m \in \mathcal{M}$ we generate machine code, MCode_m, using compiler $\tau(m)$; we then use processor template, $\mathit{PI} = \delta(m)$, to generate processor instance $\mathit{MT}\langle \mathit{MCode}_m \rangle$ and append this to $\hat{\pi}_{\delta,\tau}(p)$. To simplify the presentation we assume a uniform calling convention between separate processors; however, individual processors are free to use their own calling conventions internally.

4.7 Co-Design: Discussion and Conclusions

Source-level program transformation of a high-level HDL is a powerful technique for exploring a wide range of architectural trade-offs from an initial specification. The partitioning transformation outlined here is potentially applicable to other hardware description language (e.g. HardwareC or Tangram) given suitable compilation functions and associated processor templates. However, for the reasons given in Section 2.6 we believe that our methods are particularly powerful in the SAFL domain. To recap:

- The functional properties of SAFL allow equational reasoning and hence make a wide range of transformations applicable.
- The resource-aware properties of SAFL give our transformations precise meaning at the design-level.

We have tested our hardware/software co-design technique on various SAFL specifications, using a network of stack machines to execute the software part of our partition. Our paper on this subject includes the SAFL code for a stack machine and provides examples of applying the co-design transformation. When synthesised, the stack machine processor (without any `Call_Ext` instructions) has an area of approximately 2000 2-input equivalent gates [28].

5 Pipelines and Superscalar Expression in SAFL

In this section we provide concrete examples of both SAFL programs and SAFL-level transformations. We start by defining a simple CPU in SAFL and then demonstrate how source-level transformations can be used to introduce pipeline and superscalar features.

Consider the simple processor, resembling DLX [11] or MIPS, given in Fig. 7 (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

```
fun cpu(pc, regs) =
  (let I = imem(pc)
   let (op,m,rd,ra) =
          (I[27:31], I[26],I[21:25], I[16:20])
   let (rb,imm) = (I[0:4], sext32(I[0:15]))
   let A = regs[ra]
   let B = if m=0 then imm else 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 + (if A=0
                                        then imm
                                        else 0)
                      | _ => pc+4
   in (if op=OPhalt then regs'[0]
                    else cpu(pc', regs')));
```

Fig. 7. Simple processor

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 (see previous Section) and `dmem` representing data RAM. The function `dmem` cannot be expressed directly in SAFL (although it can in SAFL+ using the `array` declaration, see Section 6.2). For the moment it suffices to treat `dmem` as defined by a native language interface to Verilog with signature

```
extern dmem(a:32, d:32, w:1) : 32;
```

Calls to `dmem` are therefore 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 use of functional arrays[7] for `regs` and `regs'` is also to be noted: the definition `let regs' = regs[v @ i]` yields another array such that

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

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

```
(if i=0 then r0
 else if i=1 then r1
 else if i=2 then r2
 else r3)
```

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

```
((if i=0 then v else r0),
 (if i=1 then v else r1),
 (if i=2 then v else r2),
 (if i=3 then v else r3)).
```

Note that the SAFL restrictions mean that no dynamic storage is required, even when using functional array values as first-class objects. A significant advantage of the functional array notation in that it allows alternative implementation techniques and does not require access serialisation. For example 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. [24].

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[8] 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[9]
`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 estimate 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

[7] These are distinct from the mutable arrays introduced in section 6.2.

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

[9] Depending on which path is taken.

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. 8; 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. 9, we need to make a conscious adjustment to pre-fetch the instruction after the `OPhalt` by interchanging the `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. 10 shows how this can be implemented in SAFL; `memctrl` is dual ported (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. 11 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, 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. 12; 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. 12 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 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.

```
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 (if op1=OPhalt then regs[0] else      (* note this line *)
      let I = imem(pc)                      (* note this line *)
      let (op,m,rd,ra) = (I[27:31], I[26], I[21:25], I[16:20])
      let (rb,imm) = (I[0:4], sext32(I[0:15]))
      let A = regs[ra]
      let B = if m=0 then imm else regs[rb]
      let C = alu(op, A, B)
      let pc' = case op of OPj => B
                         | OPbz => pc+4 + (if A=0 then imm else 0)
                         | _ => pc+4
      in cpu1a(pc', op, regs, C, rd)));
```

Fig. 8. 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 (if op1=OPhalt then regs[0] else      (* note this line *)
      let (op,m,rd,ra) = (I[27:31], I[26], I[21:25], I[16:20])
      let (rb,imm) = (I[0:4], sext32(I[0:15]))
      let A = regs[ra]
      let B = if m=0 then imm else regs[rb]
      let C = alu(op, A, B)
      let pc' = case op of OPj => B
                         | OPbz => pc+4 + (if A=0 then imm else 0)
                         | _ => pc+4
      in cpu2(pc', op, regs, C, rd)));
```

Fig. 9. CPU with pipelined memory access

```
fun memctrl(pc,a,d,r,w) =
  (let iv = if is_dmem(pc) then dmem(pc,0,0) else imem(pc)
   let dv = if (r or w) then (if is_dmem(a) then dmem(a,d,w) else imem(a))
                        else 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. 10. CPU with memory controller

```
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 (if op2=OPhalt then regs[0] else
      let (op,m,rd,ra) = (I[27:31], 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 = if m=0 then imm else regs[rb]
      let pc' = case op of OPj => B
                         | OPbz => pc+4 + (if A=0 then imm else 0)
                         | _ => pc+4
      in cpu4(pc', op, A, B, rd,  op1, regs, C, rd1)));
```

Fig. 11. CPU with pipelined ALU and memory access

6 SAFL+: Extending the SAFL Language

The research methodology that underlies the FLaSH project involves using a small and elegant language to explore new possibilities in the field of high-level hardware description and synthesis. The simplicity of SAFL has provided us with two key benefits: firstly it allows us to be more productive by minimising implementation and development times; secondly it enables us to present our ideas clearly and concisely without becoming entangled in the complexity of a fully-featured programming language. Of course, an obvious concern with this methodology is that using a simple language to investigate analysis and compilation methods may result in the development of techniques which are *only* applicable to such simple languages. To rebut this argument we dedicate this section to extending SAFL with the capabilities one would expect from an industrial-strength HDL and demonstrate that the analysis and compilation techniques described in previous sections scale accordingly.

We introduce an extended language, SAFL+. Our motivation is to increase the expressivity of SAFL without sacrificing analysability:

- We extend SAFL with synchronous channels and assignment and argue that the resulting combination of functional, concurrent and imperative styles is a powerful framework in which to describe a wide range of hardware designs.
- Channel passing in the style of the π-calculus [22] is introduced. By parameterising functions over both data and channels the SAFL+ `fun` declaration becomes a powerful abstraction mechanism unifying a range of structuring techniques treated separately by existing HDLs (Section 6.1.3).
- We show how SAFL+ is implemented at the circuit-level (Section 6.2) and define the language formally by means of an operational semantics (Section 6.3).

6.1 SAFL+ Language Description

We start by presenting the syntax of the SAFL+ language and informally describing its semantics. A formal treatment of the language semantics is presented in Section 6.3.

SAFL+ extends SAFL by supporting a combination of imperative, concurrent and functional programming in the style of Standard ML [23]. Its abstract syntax of SAFL+ programs, p, declarations d and expressions e is presented in Fig. 13. The differences are additional forms of declaration (channels and mutable arrays) and additional forms of expression to express concurrency and imperative operations. We use r for array variables, c for channel variables, x for other variables; for clarity the letter m will be used for the constant size of an array instead of the usual letter v ranging over constants. Function declarations now take the form:

$$\texttt{fun}\ f(x_1,\ldots,x_k)\ [c_1,\ldots,c_j] = e \qquad (\text{where } k,j \geq 0)$$

We make a syntactic distinction between arguments used to pass data, $x_1,\ldots,x_k$, and arguments used to pass channels $c_1,\ldots,c_j$; array variables and function names are similarly disjoint and these may not be passed as arguments. While these rules may be relaxed somewhat, we have kept them to provide an efficient, and largely linear, mapping from SAFL resource definitions and their references to generated hardware; in particular, we use a form of *control flow analysis* (CFA) to determine possi-

```
fun cpu5(pc, regs) =
  (let (I1,I2) = (imem(pc), imem(pc+4))
   let (op1,m1,rd1,ra1) = (I1[27:31], I1[26], I1[21:25], I1[16:20])
   let (rb1,imm1) = (I1[0:4], sext32(I1[0:15]))
   let (op2,m2,rd2,ra2) = (I2[27:31], 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
     ...
     ⟨ Here I2 reads from a register written by I1---serialise them⟩
     ...
   else
     let (A1,A2) = (regs[ra1], regs[ra2])
     let (B1,B2) = ((if m1=0 then imm1 else regs[rb1]),
                    (if m2=0 then imm2 else 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 + (if A=0 then imm else 0)
                         | _ =>
                    case op of OPj => B2
                             | OPbz => pc+8 + (if A=0 then imm else 0)
                             | _ => pc+8
     in (if op1=OPhalt then regs'[0]
         else if op2=OPhalt then regs''[0]
         else cpu5(pc', regs'')));
```

Fig. 12. Simple superscalar processor

ble values at each use of a channel name which has been passed as an argument.[10]

As in SAFL, iteration is provided in the form of self-tail-recursive calls and general recursion is forbidden to permit static allocation of storage. Again the distinguished function, `main`, represents an external world interface—at the hardware level it accepts values on an input port and may later produce a value on an output port. (Although, in the current version of the language, the `main` function does not have channel parameters, note that it can read and write on globally defined channels). To ensure that recursion with channel parameters can be efficiently implemented, we require that the actual parameter list appearing in a recursive call *exactly* matches the formal parameter list, for example

```
fun f(x)[c,d] =
  if x=0 then d!x else (c!x; f(x-1)[c,d]).
```

The `static` construct, used to introduce local definitions, is provided purely for syntactic convenience. It is borrowed from the C language as a way of providing top-level definitions which are only accessible locally. It is not to be confused with the kind of *dynamic* channel-creation present in the π-calculus. The `static` construct will not be considered further in this article since it can be eliminated by bringing local `static` declarations to top level, renaming if necessary[11]. It is retained because

[10] The current compiler is rather more strict; only channel constants (those defined in a `channel` declaration) may be passed to functions; channel formal parameters may not be passed to further functions with the exception of tail-recursive calls.

[11] Note that bringing a locally defined function to the top level may require extra arguments to be added to the function in order to pass in values for its free variables.

$$\begin{array}{lcll}
e & ::= & x \mid v & \text{(Variable, Constant)} \\
 & \mid & r[e] \mid r[e] := e & \text{(Array read/write)} \\
 & \mid & c? \mid c\,!\,e & \text{(Channel read/write)} \\
 & \mid & a(e_1, \ldots, e_k) & \text{(Call to primitive function)} \\
 & \mid & f(e_1, \ldots, e_k)[c_1, \ldots, c_j] & \text{(Call to user-defined function)} \\
 & \mid & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \text{(Conditional)} \\
 & \mid & \texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0 & \text{(Parallel let)} \\
 & \mid & \texttt{static } p \texttt{ in } e & \text{(Local declarations)} \\
 & \mid & e \parallel e \mid e;\, e & \text{(Parallel/sequential composition)} \\
 & & & \\
d & ::= & \texttt{fun } f(x_1, \ldots, x_k)[c_1, \ldots, c_j] = e & \text{(Function declaration)} \\
 & \mid & \texttt{channel } c & \text{(Channel declaration)} \\
 & \mid & \texttt{channel external } c & \text{(I/O Channel declaration)} \\
 & \mid & \texttt{array } [v]\ r & \text{(Array declaration)} \\
 & & & \\
p & ::= & d_1 \ldots d_n &
\end{array}$$

Fig. 13. The abstract syntax of SAFL+ programs, p

it helps program structuring and because we compile it specially to avoid the cost of the above treatment of free variables.

Our existing compiler provides a number of simple syntactic sugarings: the declaration `array [1] r` can be written `reg r`—when accessing such arrays one writes `r` instead of `r[0]` and functions without channel parameters can omit their square brackets completely (both in definition and calls). Since on a (tail) recursive call channel parameters must exactly match those in the function's definition, they are usually not written.

6.1.1 Resource Awareness Revisited

We extend the SAFL view which models hardware as a fixed set of communicating and (possibly) shared resources with additional resource forms and operations thereon. As can be seen from Fig. 13, a program consists of a series of resource declarations. There are three different types of resource, each of which addresses a key element of hardware design:

Function: used for *Computation*
implemented by *General Purpose Logic*;
Channel: used for *Communication*
implemented by *Buses, Wires and Control Logic*;
Array: used for *Mutable Storage*
implemented by *Memories or Registers.*

SAFL+ preserves the SAFL property of resource-awareness; each declaration, d, (be it a function, channel or array declaration) is compiled to a *single* hardware block, H_d. Multiple references to d at the source-level (e.g. multiple calls to a function or multiple assignments to an array) correspond to the sharing of H_d at the circuit-level.

A call, $f(\vec{x})[\vec{c}]$, corresponds to: (i) acquiring mutually exclusive access to resource, H_f; (ii) passing data $\vec{x}$ and channel-parameters $\vec{c}$ into H_f; (iii) waiting for H_f to terminate; and (iv) latching the result from H_f's shared output.

Just as in SAFL, sharing issues, such as ensuring mutually exclusive access to resources, are dealt with by the automatic generation of synchronous arbiters. Our SAFL+ compiler uses the Soft Scheduling techniques outlined in Section 3.3.1 to optimise away redundant arbiters.

6.1.2 Channels and Channel Passing

SAFL+ provides synchronous channels to allow parallel threads to synchronize and transfer information. Channels can be used to transfer data locally within a function, or globally, between concurrently executing functions.

Our channels generalise Occam [13] and Handel-C [7] channels in a number of ways: SAFL+ channels can have any number of readers and writers, are bidirectional and can connect any number of parallel processes. As in the π-calculus, if there are multiple readers and multiple writers all wanting to communicate on the same channel then a single reader and a single writer are chosen non-deterministically.

At the hardware level a channel is implemented as a many-to-many communications bus supporting atomic transfer of single values between readers and writers (see Section 6.2). No language-support is provided for *bus-transactions* (e.g. lock the bus for 20 cycles and write the following sequence of data values onto it). In Section 6.1.3 a SAFL+ code fragment is presented which shows how such transactions can be implemented by using explicit locking.

Channels declared as `external` are used for I/O: writing to an external channel corresponds to an output action; reading an external channel corresponds to reading an input. There is no synchronisation on external channels although writes are guaranteed to occur under mutual exclusion. For example, for an external channel `c`, the only two possible output sequences occurring as a result of evaluating expression `(c!2 || c!3)` are $\langle 2, 3\rangle$ or $\langle 3, 2\rangle$. (See Section 6.3).

The code in Fig. 14 illustrates channel-passing in SAFL+ by defining two resources parameterised over channel parameters: `Accumulate` reads integers from a channel, returning their total when a 0 is read; and `GenNumbers` writes a decreasing stream of integers to a channel, terminating when 0 is reached. The function `sum(x)` calculates the sum of the first `x` integers by composing the two resources in parallel and linking them with a common channel, `connect`. (Note that the parallel composition operator, `||`, waits for both its components to terminate before returning the value of the rightmost one.)

6.1.3 The Motivation for Channel Passing

By parameterising functions over both data and channel parameters, the SAFL+ `fun` definition becomes a powerful abstraction mechanism, encapsulating a wide range of structuring primitives treated separately in existing HDLs:

– Pure functions can be expressed by omitting channel parameters:
```
    fun f(x,y) = ...
```
– Structural-level blocks (e.g. the `module` construct of Verilog) can be expressed as non-terminating `fun` declarations parameterised over channels:
```
    fun m() [in1,in2,out] = (...; m())
```
– HardwareC `process` declarations can be expressed as non-terminating `fun` definitions (possibly without channel or data parameters):
```
    fun p() = (...; p())
```
– HardwareC `procedure`s can be expressed as `fun` declarations that return a result of zero width:
```
    fun s(x,y) = (...; ())
```
The function call-return protocol ensures that a caller to `s` may not continue until `s` has completed, even though no data is returned.

As well as unifying a number of common abstraction primitives, SAFL+ also supports a style of programming not exploited by existing HDLs. Recall the definition of `Accumulate` in Section 6.1.2. The `Accumulate` function can be seen as a hybrid between a structural-level block (since it is parameterised over a port, `c`) and a function (since it terminates, returning a result). More generally, by passing in locally defined channels, a caller, f, is able to synchronise and communicate with its callee, g, during g's execution. For example, consider the SAFL+ code in Fig. 15 which declares a lock shared between functions `f1` and `f2` to implement mutual exclusion from the critical regions. The `lock` function is parameterised over two channels: `acquired` is signalled as soon as `lock` starts executing, indicating to the caller that the lock has been acquired; `release` is used by the caller to signal that it has finished with the lock (at which point `lock` terminates). Recall that resource-awareness means that `lock` represents a single resource shared by functions `f1` and `f2`: the compiler ensures that only one caller can acquire it at a time. By passing in locally defined channels, functions `f1` and `f2` are able to communicate with `lock` during its execution.

6.2 Translating SAFL+ to Hardware

We have already described how SAFL is translated to synchronous hardware (see Section 3.1). Here we show how the SAFL+ extensions (i.e. channels, channel passing and arrays) can be integrated into this existing framework. As before, we adopt the graphical convention that thick lines represent data-wires and thin lines represent control signals.

A channel is translated into a shared bus surrounded by control logic to arbitrate between waiting readers and writers. Fig. 16 shows channel control circuitry in a case where there are two readers and three writers. Since we are primarily targeting FPGAs we choose to multiplex data onto the bus rather than using tri-state buffers. To perform a read operation the reader signals its read-request and blocks until the corresponding read-acknowledge is signalled. The read-acknowledge line remains high for one cycle during which time the reader samples the data from the channel. To perform a write operation the writer places the data to be written onto a channel's data-input and signals the corresponding write-request line; the writer blocks until the corresponding write-acknowledge is signalled. Our current compiler synthesises static fixed-priority arbiters to resolve multiple simultaneous read requests or multiple simultaneous write requests. However, since the SAFL+ semantics do not specify an arbitration policy, future compilers are free to exploit other selection mechanisms.

Our SAFL+ compiler performs a static flow-analysis to determine which *actual channels* (those bound directly by the `channel` construct) a given formal-channel-parameter may range over. This information enables the compiler to statically connect each channel operation (read or write) to every possible actual channel that it may need to access dynamically. At the circuit level channel values are represented as small integers which are passed as additional parameters on a function call.

Our intermediate code [32] is augmented with READ and WRITE nodes representing channel operations. In cases where our flow-analysis detects that a channel operation may refer to a number of possible actual channels, multiplexers and demultiplexers are used to dynamically route to the appropriate channel. READ nodes have a control-input (used to signal the start of the operation), a control-output (used to signal the completion of the operation), a channel-select-input (used to select which actual channel to read from) and a data-output

```
fun Accumulate(state)[c] = let val read_value = c?
                           in if read_value=0 then state
                              else Accumulate(state+read_value)  end

fun GenNumbers(state)[c] = (c!state; if c=0 then () else GenNumbers(state-1))

fun sum(x) = static channel connect
             in GenNumbers(x)[connect] || Accumulate(0)[connect]   end
```

Fig. 14. Example showing SAFL+ channel-passing

```
fun lock()[acquired, release] = acquired!(); release?

fun f1() = static channel go   channel done
           in (lock()[go,done] ||  (go?;  (* f1's critical region *)  done!()) )   end

fun f2() = static channel go   channel done
           in (lock()[go,done] ||  (go?;  (* f2's critical region *)  done!()) )   end
```

Fig. 15. Mutual exclusion implemented by `lock`

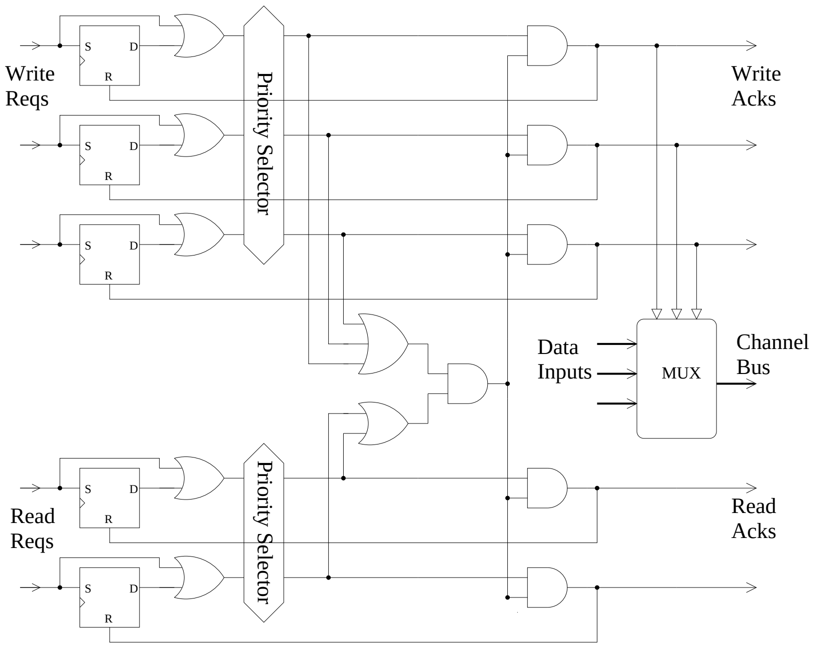


Fig. 16. A Channel Controller. The synchronous RS flip-flops (R-dominant) are used to latch pending requests (represented as 1-cycle pulses). Static fixed priority selectors are used to arbitrate between multiple requests. The 3 data-inputs are used by the three writers to put data onto the bus.

(the result of the read operation). Similarly WRITE nodes have a control-input, a control-output, a channel-select-input and a data-output. Fig. 17 shows READ and WRITE nodes connected to multiple channels.

We extend the translation of `fun` declarations to include extra registers to latch channel-parameters. At the circuit-level channel-parameters are fed into the select lines of the multiplexers and demultiplexers seen in Fig. 17. In this example 'ChSel' would be read directly from the registers storing the enclosing function's channel-parameters.

Arrays are represented as RAMs wrapped up in the necessary logic to arbitrate between multiple concurrent accesses. Our compiler translates array declarations, `array [m] r:n`, into SAFL+ function definitions with signature:

```
fun r (addr:k, data:n, wr_select:1) : n
```

where `k` is $\lceil \log_2 \mathtt{m} \rceil$. Calling `r` always returns the value stored at memory location `addr`. If `wr_select` is 1 then location `addr` is updated to contain `data`. Hence array assignments, `r[e1] := e2`, are translated into function calls of the form `r(e1,e2,1)` and array accesses, `r[e]`, are translated into calls of the form `r(e,0,0)`. Treat-

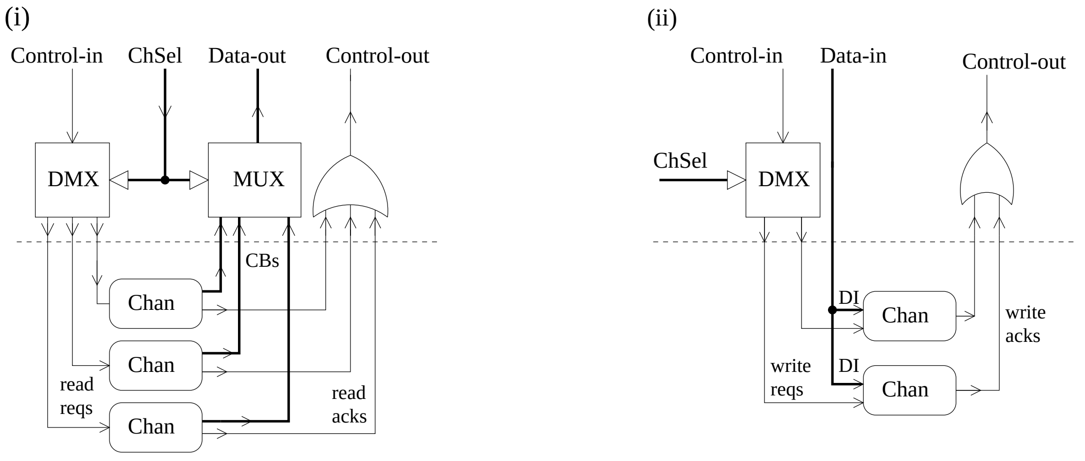


Fig. 17. (*i*) A Read node connected to 3 channels; (*ii*) A Write node connected to 2 channels. Each of the boxes labelled 'Chan' is a channel (as in Fig. 16). Although each such channel may well have other readers/writers these are not shown in the figure. The data-wires labelled 'CB' are the channel buses, those labelled 'DI' are channels' data-inputs (multiplexed onto the channel buses—see Fig. 16). 'ChSel' is the channel-select-input. Note that (although not shown in this figure) channel buses may be shared among many readers. The dotted line represents the boundary between the resource performing the channel operation and the channels themselves.

ing arrays as SAFL+ functions in this way allows us to use the compiler's existing machinery to synthesise the necessary logic to serialise concurrent accesses to the array and latch address lines. The compiler automatically generates the body of `r`, which consists solely of RAM.

6.3 Operational Semantics for SAFL+

In this section we define the meaning of the SAFL+ language formally through an operational semantics. Although, at first sight, the semantics may seem theoretical and far-removed from hardware-implementation we argue that this is not the case. We note that many of the symbols in Fig. 20 have a direct correspondence to circuit-level components. For example, *channel resources*, $\langle v \rangle_c$ (see below), represent channel controller circuits (as shown in Fig. 16) and the (*Call*) rule (see Fig. 20) corresponds directly to transferring data into the callee's argument registers (see Section 3.1 for circuits which achieve this effect).

Due to the static nature of SAFL+, we can simplify the semantics by assuming that: programs have been α-converted to make all variable names distinct. Additionally, we will assume the form of a SAFL+ function is $\texttt{fun}\ f(\vec{x})[\vec{c}] = b_f$ so that we can access the body of function f as b_f.

A program *resource state*, ranged over by σ, represents the state of a single resource; a *program state*, ranged over by Σ, is a multiset of *resource states*; this is often written $\sigma_1 \mid \ldots \mid \sigma_n$. *Resources* consist of *function resources*, *channel resources* and *array resources* given below; they are defined formally in Fig. 18. As resources execute concurrently in the style of the Chemical Abstract Machine [3] program states can be viewed as a "solution" of reacting resource states. Our presentation is inspired by that of Marlow *et al* [19] but, because of static allocatability of resources, our transition rules will only change the activation statuses of resource states, and not the number of them.

We give the semantics by describing how one program state, Σ, evolves into another, say Σ', by means of a *transition*: $\Sigma \xrightarrow{\alpha} \Sigma'$, where α represents an optional I/O action taking one of the following forms:

$\bar{\mathbf{c}}\langle v \rangle$	output v on external channel $\mathbf{c}$;
$\mathbf{c}(v)$	read a value v from external channel $\mathbf{c}$;
$go(\vec{v})$	pass parameters $\vec{v}$ into the `main` function;
$done(v)$	read result v from the `main` function.

Note that we use a bold-face $\mathbf{c}$ to range over external channels (in contrast to c, which ranges over non-external channels).

Each non-external channel declaration, `channel` c, corresponds to a channel resource. When an empty channel resource (written $\langle\rangle_c$) reacts with a waiting writer a value, v, is transferred and c becomes full (written $\langle v \rangle_c$). On reacting with a waiting reader, the value is consumed and the c enters an acknowledge state (written $\langle \text{Ack} \rangle_c$). The Ack interacts with the writer, notifying it that communication has taken place and returning c to the empty state, $\langle\rangle_c$. The explicit use of Ack models the synchronous nature of SAFL+ channels ensuring that a writer is blocked until its data has been consumed by a reader.

Array resources, $[\mathcal{S}]_r$, correspond to array declarations, `array [m]` r. The content of the array, $\mathcal{S}$ is is a function representing the content of the array (i.e. mapping indexes $0 \ldots (\mathtt{m}-1)$ onto values) and where $upb(\mathcal{S})$ is the upper bound of $\mathcal{S}$ (here `m`). We write $\mathcal{S}\{j \mapsto v\}$ to denote the function which is as $\mathcal{S}$ but maps index j onto value v. Accessing elements outside the bounds of an array leads to undefined behaviour. To reflect this we define $\mathcal{S}(j)$ to be an undefined value if $j \geq upb(\mathcal{S})$.

Furthermore if $j \geq upb(\mathcal{S})$ then $\mathcal{S}\{j \mapsto v\}$ represents an undefined state mapping indexes $0 \dots (upb(\mathcal{S}) - 1)$ onto undefined values.

Each SAFL+ function declaration, `fun` $f\ (\vec{x})\ [\vec{c}] = b_f$, is represented by a function resource written $(\!| \cdot |\!)_f$. At any given time this may be *free*, written $\mathbb{0}_f$ or *busy* performing a computation e, written $(\!|e|\!)_f$; e is called an *evaluation state*. The syntax of evaluation states (see Fig. 18) is that of SAFL+ expressions augmented with an additional $\mathcal{W}_g$ construct which signifies awaiting the result of an initiated call to function resource g. One can see $\mathcal{W}_g$ as a form of channel-read, $c?$, construct which reads from the (private) channel written by g to signal function return.

Using the Chemical Abstract Machine metaphor, resource elaboration or interaction takes place independently of the rest of the program state; the fact that program states are multisets means that we understand reactions (see Fig. 20) of the form

$$\sigma_1 \mid \dots \mid \sigma_n \quad\longrightarrow\quad \sigma'_1 \mid \dots \mid \sigma'_n$$

as including those of the form

$$\sigma_1 \mid \dots \mid \sigma_n \mid \Sigma \quad\longrightarrow\quad \sigma'_1 \mid \dots \mid \sigma'_n \mid \Sigma$$

and even variants permuting the σ_i and Σ.

SAFL+ is an implicitly parallel language—an expression may contain a number of sub-expressions which can be evaluated concurrently. To formalise this notion we use a *context*, $\mathbb{E}$, to highlight the parts of an evaluation state which can be evaluated concurrently (see Fig. 19). Intuitively a context is an evaluation state $\mathbb{E}[\cdot]$ with a hole $[\cdot]$ into which we can insert an evaluation state, e, to derive a new evaluation state $\mathbb{E}[e]$.

A useful mental model is to consider a frontier of evaluation which is defined by $\mathbb{E}$ and advanced by applying the transition rules (see Section 6.3.1 and Fig. 20).

6.3.1 Transition Rules

This section presents the transition rules for SAFL+; SAFL is included as a special case.

Given an evaluation state, e, then values $v_1 \dots v_k$ (respectively channel names $c'_1 \dots c'_j$) may be substituted for variables $x_1 \dots x_k$ (respectively channel parameter names $c_1 \dots c_j$) using the notation $\{\vec{c'}/\vec{c}, \vec{v}/\vec{x}\}e$.

The rules in Fig. 20 are divided into six categories:

- (*Call*) and (*Return*) deal with interaction between functional resources.
- (*Ch-Write*), (*Ch-Read*) and (*Ch-Ack*) model communication over channels.
- (*Input*) and (*Output*) deal with I/O through external channels.
- (*Ar-Write*) and (*Ar-Read*) handle access to array resources.
- (*Start*) and (*End*) correspond to external call/return of `main()`.
- The remainder of the rules represent local computation within a function.

Note that the left hand side of the (*Tail-Rec*) rule is not enclosed in a context. This reflects the syntactic restriction that recursive calls are forbidden to occur as a proper sub-expression of a function body (with the exception of `if-then-else`, `let-in` and sequencing '`;`'). We remark that channel parameters cause no additional problems over those of ordinary value parameters.

6.4 SAFL+: Discussion and Conclusions

We have described the SAFL+ language, outlining its use for hardware description and synthesis. We argue that the major advantages of SAFL+ over most conventional high-level description languages are:

- The combination of resource-awareness and channel-passing makes SAFL+ `fun` declarations a very powerful abstraction mechanism. Both structural blocks and functions can be seen as special cases of `fun` declarations[12].
- By structuring programs as a series of function definitions (as opposed to a collection of structural blocks), SAFL+ supports a wide range of analyses and transformations which are not applicable to conventional HDLs.
- SAFL+ has a formally defined semantics.

7 Comparison with Other Work

It is important to draw a distinction between this work and the framework used in Hydra [29], Lava [5], HML [17], μFP [35] and Hawk [20]. SAFL is a *behavioural* hardware description language which, just like a normal language designed for software description, allows a programmer to describe algorithms. SAFL specifications can either be translated into structural circuit descriptions by our optimising compiler, or can be executed as ML-style programs for simulation purposes[13]. In contrast, Hydra, Lava, HML, μFP and Hawk use the power of existing functional languages to describe the interconnections between low-level components. Thus they are *structural* hardware design systems in our taxonomy. In Lava or Hydra a hardware description is a Haskell program which can either generate a netlist or simulate its behaviour (depending on the interpretation in which it is executed). The Hawk system is similar in many respects, but performs only hardware simulation.

[12] Of course, we prefer designers not to code structural blocks as non-terminating SAFL+ `fun` definitions unless absolutely necessary, since we consider it a "low-level" programming style.

[13] A cycle-accurate SAFL interpreter has been implemented. We use this tool to model SAFL designs before they are mapped to synchronous hardware.

$$
\begin{array}{lll}
\sigma & ::= \ \cdot_f \mid (\!| e |\!)_f & \text{(function resource } f\text{: free or busy)}\\
 & \mid \ \langle\rangle_c \mid \langle v\rangle_c \mid \langle \text{Ack}\rangle_c & \text{(channel resource } c\text{: free, awaiting read, in acknowledge state)}\\
 & \mid \ [\mathcal{S}]_r & \text{(array resource } r\text{: with content S)}\\
\\
e & ::= \ \mathcal{W}_g & \text{(awaiting result from } g)\\
 & \mid \ x \mid v \mid f(e,\ldots,e)[c_1,\ldots,c_n] \mid a(e,\ldots,e) & \ldots\\
 & \mid \ \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ (x_1,\ldots,x_k) = (e,\ldots,e)\ \texttt{in}\ e & \ldots\\
 & \mid \ r[e] \mid r[e] := e \mid c? \mid c\,!\,e \mid e \,\|\, e \mid e; e & \text{(as in Fig. 13)}
\end{array}
$$

Fig. 18. The Syntax of Resource States, σ, and Evaluation States, e

$$
\begin{array}{l}
\cdot \ ::= [\cdot]\\
\mid \ f(\cdot, e_2,\ldots,e_k) \mid \ldots \mid f(e_1,\ldots,e_{k-1},\cdot) \mid a(\cdot, e_2,\ldots,e_k) \mid \ldots \mid a(e_1,\ldots,e_{k-1},\cdot)\\
\mid \ \texttt{if}\ \cdot\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{let}\ \vec{x} = (\cdot, e_2,\ldots,e_k)\ \texttt{in}\ e \mid \ldots \mid \texttt{let}\ \vec{x} = (e_1,\ldots,e_{k-1},\cdot)\ \texttt{in}\ e\\
\mid \ r[\cdot] \mid r[\cdot] := e \mid r[e] := \cdot \mid c\,!\,\cdot \mid \cdot \,\|\, e \mid e \,\|\, \cdot \mid \cdot\,; e
\end{array}
$$

Fig. 19. A context, $\cdot$, defining which sub-expressions admit (concurrent) evaluation

$$
\begin{array}{rcll}
(\!| \cdot[g(v_1,\ldots,v_k)[d_1,\ldots,d_j]] |\!)_f \mid \cdot_g & \longrightarrow & (\!| \cdot[\mathcal{W}_g] |\!)_f \mid (\!| \{\vec{d}/\vec{c}, \vec{v}/\vec{x}\} b_g |\!)_g \quad f \neq g & (\mathit{Call})\\
(\!| v |\!)_f \mid (\!| \cdot[\mathcal{W}_f] |\!)_g & \longrightarrow & \cdot_f \mid (\!| \cdot[v] |\!)_g & (\mathit{Return})\\
(\!| \cdot[c\,!\,v] |\!)_f \mid \langle\rangle_c & \longrightarrow & (\!| \cdot[\mathcal{W}_c] |\!)_f \mid \langle v\rangle_c & (\mathit{Ch\text{-}Write})\\
(\!| \cdot[c?] |\!)_f \mid \langle v\rangle_c & \longrightarrow & (\!| \cdot[v] |\!)_f \mid \langle \text{Ack}\rangle_c & (\mathit{Ch\text{-}Read})\\
(\!| \cdot[\mathcal{W}_c] |\!)_f \mid \langle \text{Ack}\rangle_c & \longrightarrow & (\!| \cdot[()] |\!)_f \mid \langle\rangle_c & (\mathit{Ch\text{-}Ack})\\
(\!| \cdot[\mathbf{c}\,!\,v] |\!)_f & \xrightarrow{\bar{\mathbf{c}}\langle v\rangle} & (\!| \cdot[()] |\!)_f & (\mathit{Output})\\
(\!| \cdot[\mathbf{c}?] |\!)_f & \xrightarrow{\mathbf{c}(v)} & (\!| \cdot[v] |\!)_f & (\mathit{Input})\\
(\!| \cdot[r[v_1] := v_2] |\!)_f \mid [\mathcal{S}]_r & \longrightarrow & (\!| \cdot[()] |\!)_f \mid [\mathcal{S}\{v_1 \mapsto v_2\}]_r & (\mathit{Ar\text{-}Write})\\
(\!| \cdot[r[v]] |\!)_f \mid [\mathcal{S}]_r & \longrightarrow & (\!| \cdot[\mathcal{S}(v)] |\!)_f \mid [\mathcal{S}]_r & (\mathit{Ar\text{-}Read})\\
\cdot_{\texttt{main}} & \xrightarrow{go(\vec{v})} & (\!| \{\vec{v}/\vec{x}\} b_{\texttt{main}} |\!)_{\texttt{main}} & (\mathit{Start})\\
(\!| v |\!)_{\texttt{main}} & \xrightarrow{done(v)} & \cdot_{\texttt{main}} & (\mathit{End})\\
\\
(\!| \cdot[a(v_1,\ldots,v_k)] |\!)_f & \longrightarrow & (\!| \cdot[v] |\!)_f \ \text{ where } v = a(v_1,\ldots,v_k) & (\mathit{PrimOp})\\
(\!| \cdot[\texttt{if}\ 0\ \texttt{then}\ e_2\ \texttt{else}\ e_3] |\!)_f & \longrightarrow & (\!| \cdot[e_3] |\!)_f & (\mathit{CFalse})\\
(\!| \cdot[\texttt{if}\ v\ \texttt{then}\ e_2\ \texttt{else}\ e_3] |\!)_f & \longrightarrow & (\!| \cdot[e_2] |\!)_f \quad v \neq 0 & (\mathit{CTrue})\\
(\!| \cdot[\texttt{let}\ \vec{x} = \vec{v}\ \texttt{in}\ e] |\!)_f & \longrightarrow & (\!| \cdot[\{\vec{v}/\vec{x}\}e] |\!)_f & (\mathit{Let})\\
(\!| \cdot[v; e] |\!)_f & \longrightarrow & (\!| \cdot[e] |\!)_f & (\mathit{Seq})\\
(\!| \cdot[v_1 \,\|\, v_2] |\!)_f & \longrightarrow & (\!| \cdot[v_2] |\!)_f & (\mathit{Par})\\
(\!| f(v_1,\ldots,v_k)[d_1,\ldots,d_j] |\!)_f & \longrightarrow & (\!| \{\vec{d}/\vec{c},\ \vec{v}/\vec{x}\} b_f |\!)_f & (\mathit{Tail\text{-}Rec})
\end{array}
$$

Fig. 20. Transition Rules for SAFL+

One area we particularly wish to highlight is Singh's work [8] on extending Lava with combinators which express geometrical (and hence layout) information. This again distinguishes SAFL and Lava; whereas we *compile* functional algorithms to structural HDLs, Lava *is* a structural HDL. It is theoretically possible to use SAFL as a front-end for Lava or indeed to code the behavioural ideas of SAFL in terms of Lava's gate-level primitives. Although outside the scope of this article, we are currently developing a framework which integrates Lava-style structural expansion into the SAFL language [31].

Some other researchers have also considered using functional languages for *behavioural* hardware description. A notable example is Johnson's Digital Design Derivation (DDD) system [14] which uses a scheme-like language to describe circuit behaviour. A series of semantics preserving transformations are presented which can be used to refine a behavioural specification into a circuit structure; the transformations are applied manually by an engineer. This is a different approach to hardware design using SAFL [27] where, although semantics preserving transformations are used to explore architectural trade-offs (including allocation, binding and scheduling [21]) at the source-level, the resulting SAFL specification is fed into an optimising compiler which generates a structural hardware design automatically. Also, the DDD system does not have SAFL's notions of either static allocation (see Section 2.1) or resource awareness (see Section 2.5): two of the key points of our research.

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 [25]: *"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."*

A number of synchronous dataflow languages, the most notable being LUSTRE [9], 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 [10]). Furthermore LUSTRE is inherently synchronous: the whole system proceeds in lock-step through a series of discrete time steps. This is in contrast to SAFL which can easily be compiled into both synchronous and asynchronous implementations.

There are various parallels between SAFL+ and the HardwareC [16] language: both provide synchronous channels and allow function definitions to be treated as shared resources. The major differences are: (*i*) whereas HardwareC offers purely imperative features, SAFL+ also supports a functional style; (*ii*) the expressivity of SAFL+ is greater due to our less restrictive scheduling policy [33]; and (*iii*) HardwareC provides a `block` primitive for structural-level declarations whereas SAFL+ only allows function declarations. Note, however, that the SAFL+ expression forms are sufficiently simple and powerful that we can see all four of HardwareC's structuring primitives (`block`, `process`, `procedure` and `function`) as special cases of SAFL+ `fun` declarations (see Section 6.1.3). Since SAFL+ only requires a single structuring primitive it yields a simpler semantics.

Although languages such as HardwareC [16] and Tangram [2] allow function definitions to be treated as shared resources we feel that these projects have not gone as far as us in exploiting the potential benefits of this design style. In particular:

– We have developed a number of global analyses and optimisations which are only made possible by structuring hardware as a series of function definitions [33, 32].
– We have investigated the impact of source-to-source transformations on SAFL and shown that it is a powerful tool for design-space exploration. It is the functional properties of the language that make program transformation so powerful.

Hoe and Arvind's TRAC system [12] generates synchronous hardware from a high-level specification expressed in a term-rewriting system. Broadly speaking, terms correspond to states and rules correspond to combinatorial logic which calculates the next state of the system. Restrictions imposed on the structure of rewrite rules facilitate the static allocation of storage. This closely corresponds to the tail-recursion restriction imposed on SAFL programs to achieve static allocation.

8 Conclusions and Directions for Future Work

This article has introduced and formally defined SAFL and SAFL+, exemplifying their use for processor design and hardware/software co-design. We argue that the major advantages over conventional high-level synthesis languages are:

– Programs are compiled with a *resource-aware* compiler so that the generated hardware largely reflects source structure.
– Source-to-source transformation is a powerful technique for design-space exploration.
– By structuring programs as a series of function definitions (as opposed to a collection of structural blocks), SAFL and SAFL+ support a wide range of analyses and transformations which are not applicable to conventional HDLs.
– SAFL and SAFL+ have a formally defined semantics; this can be used to justify the correctness of analyses and transformations.
– The SAFL+ combination of resource-awareness and channel-passing makes `fun` declarations a very powerful abstraction mechanism. Both structural blocks and functions can be seen as special cases of `fun` declarations.

Although we have implemented silicon compilers for SAFL and SAFL+ and tested them on realistic examples, we have yet to use the system to build a large system-on-a-chip design. Using SAFL+ to construct a large hardware design will provide useful results regarding both the expressivity of the language and the efficiency of our compiler.

The translation of SAFL and SAFL+ to hardware given in this paper (Sections 3.1 and 6.2) outlines one of many possible implementation techniques. We are currently implementing silicon compilers targeting asynchronous and globally-asynchronous-locally-synchronous (GALS) hardware. The latter involves allocating SAFL function resources to separate clock domains and extending our compiler to generate the necessary inter-clock-domain interfaces—we argue that this is another example of a transformation with pervasive effects on silicon. Note that when specifying hardware in languages such as Verilog/VHDL the chosen design style quickly becomes fixed: even at the behavioural level one is forced to encode low-level protocols explicitly across module boundaries. SAFL and SAFL+ do not suffer from this problem. Since the use of functions abstracts inter-block control- and data-flow we are free to map the high-level specifications to any design style we choose.

Resource awareness allows SAFL to describe the *system-level* structure of a design by mapping `fun` declarations to circuit-level functional units. In contrast, systems such as μFP and Lava offer much finer-grained control over circuit structure, taking logic-gates (rather than function definitions) as their structural primitives. In practice both approaches are appropriate depending on the type of hardware that is being designed. Motivated by this observation, we have recently developed a framework which integrates Lava-style structural expansion with SAFL [31]. In future work we wish to explore these possibilities further.

Applying SAFL-level transformations entirely by hand is a tedious and error-prone business. To address this issue we are working on a semi-automated tool which allows a designer to select transformations from a predefined library and apply them to SAFL fragments. There is a great deal of potential for future work in this area. In particular we would like to investigate methods for automatically choosing which transformations to apply in accordance with user specified time-area targets.

Acknowledgement

This work was supported by (UK) EPSRC grant, reference GR/N64256: "A Resource-Aware Functional Language for Hardware Synthesis"; AT&T Research Laboratories Cambridge provided additional support (including sponsoring the second author).

References

1. Verilog HDL Language Reference Manual. IEEE Draft Standard 1364, October 1995.
2. Van Berkel, K. Handshake Circuits: an Asynchronous Architecture for VLSI Programming. International Series on Parallel Computation, vol. 5. Cambridge University Press, 1993.
3. Berry, G., and Boudol, G. The Chemical Abstract Machine. *Theoretical Computer Science 96* (1992), 217–248.
4. Burstall, R.M. and Darlington, J. A Transformation System for Developing Recursive Programs, JACM 24(1), 1979.
5. Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. Lava: Hardware Description in Haskell. In *Proceedings of the 3rd International Conference on Functional Programming* (1998), SIGPLAN, ACM.
6. Cartwright, R. and Fagan, M. Soft Typing. Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, 1991.
7. Celoxica (Ltd.). Handel-C Language Datasheet. Available from Celoxica: `http://www.celoxica.com`.
8. Claessen, K., Sheeran M. and Singh, S. Design and Verification of a Sorter Core. Lecture Notes in Computer Science: Proc. CHARME'01, vol. 2144, Springer-Verlag, September 2001.
9. Halbwachs, N., Caspi, P., Raymond, P. and Pilaud, D. The Synchronous Dataflow Programming Language LUSTRE. Proc. IEEE, vol. 79(9). September 1991.
10. 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.
11. Hennessy, J.L. and Patterson, D.A. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 1990.
12. Hoe, J., and Arvind. Hardware Synthesis from Term Rewriting Systems. In *Proceedings of X IFIP International Conference on VLSI* (1999).
13. Inmos (Ltd.). *Occam 2 Reference Manual.* Prentice Hall, 1998.
14. Johnson, S., and Bose, B. DDD: A System for Mechanized Digital Design Derivation. Tech. Rep. 323, Indiana University, 1990.
15. Jones, N., Gomard, C. and Sestoft, P. Partial Evaluation and Automatic Program Generation. Published by Prentice Hall (1993); ISBN 0-13-020249-5.
16. Ku, D., and De Micheli, G. HardwareC—A Language for Hardware Design (version 2.0). Tech. Rep. CSL-TR-90-419, Stanford University, 1990.
17. Li, Y., and Leeser, M. HML, A Novel Hardware Description Language and its Translation to VHDL. *IEEE Transactions on VLSI Systems*, 1 (February 2000).
18. McKay, N and Singh, S. Dynamic Specialisation of XC6200 FPGAs by Partial Evaluation. Lecture Notes in Computer Science: Proc. FPL 1998, vol. 1482, Springer-Verlag, 1998.
19. Marlow, S., Peyton Jones, S., Moran, A., and Reppy, J. Asynchronous Exceptions in Haskell. To appear. *Proceedings of ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)* 2001.
20. Matthews, J., Cook, B., and Launchbury, J. Microprocessor specification in Hawk. In *Proceedings of the IEEE International Conference on Computer Languages* (1998).
21. De Micheli, G. Synthesis and Optimization of Digital Circuits. McGraw-Hill, 1994.
22. Milner, R. The Polyadic π-calculus: A Tutorial. Tech. Rep. ECS-LFCS-91-180, University of Edinburgh, October 1991.
23. Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML (Revised).* MIT Press, 1997.
24. 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.
25. Morison, J.D. and Clarke, A.S. ELLA 2000: A Language for Electronic System Design. Cambridge University Press 1994.

26. Mycroft, A. and Sharp, R.W. Hardware Synthesis using SAFL and Application to Processor Design. Lecture Notes in Computer Science: Proc. CHARME'01, vol. 2144, Springer-Verlag, September 2001.
27. Mycroft, A., and Sharp, R. A Statically Allocated Parallel Functional Language. In *Proceedings of the International Conference on Automata, Languages and Programming* (2000), vol. 1853 of *LNCS*, Springer-Verlag.
28. Mycroft, A., and Sharp, R. Hardware/software Codesign Using Functional Languages. In *Proceedings of TACAS* (2001), vol. 2031 of *LNCS*, Springer-Verlag.
29. 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.
30. Page, I. Parameterised Processor Generation. In Moore and Luk (eds.), More FPGAs, pages 225-237. Abingdon EE&CS Books, 1993.
31. Sharp, R. Functional Design using Behavioural and Structural Components. Proceedings of the Workshop on Designing Correct Circuits (DCC) 2002. To appear.
32. Sharp, R., and Mycroft, A. The FLaSH compiler: Efficient Circuits from Functional Specifications. Tech. Rep. tr.2000.3, AT&T Laboratories Cambridge, 2000.
33. Sharp, R., and Mycroft, A. Soft Scheduling for Hardware, 2001. In *Proceedings of the 8th International Static Analysis Symposium* (2001), vol. 2126 of *LNCS*, Springer-Verlag.
34. Sharp, R. and Mycroft, A. A Higher-Level Language for Hardware Synthesis. Lecture Notes in Computer Science: Proc. CHARME'01, vol. 2144, Springer-Verlag, September 2001.
35. Sheeran, M. μFP, A Language for VLSI design. In *Proceedings of the ACM Symposium on LISP and Functional Programming* (1984).