Hardware/Software Co-Design using Functional Languages

Alan Mycroft* **and Richard Sharp***

Computer Laboratory, Cambridge University

http://www.cl.cam.ac.uk/users/am/

3 April 2001

*Work done at/in collaboration with AT&T Laboratories Cambridge.

SAFL

UNIVERSITY OF CAMBRIDGE

- Language of the FLaSH project (Functional Languages for Synthesising Hardware) we have developed at AT&T Laboratories Cambridge.
- First-order call-by-value functional language
- Syntactically restricted to be statically allocatable (recursive loops must only involve tail-calls)—well-suited to hardware synthesis
- Parallel evaluation of tupled expressions (arguments)
- Shared resources protected by semaphores (arbiters); recursion restrictions imply no deadlock.

Why good for hardware?

UNIVERSITY OF
CAMBRIDGE

- Static allocation is exactly what hardware does (any form of dynamic addressing is hard—"von Neumann bottleneck").
- Parallelism (from call-by-value) is what hardware provides lots of.
- Resource aware: the *compiler* treats functions as hardware units; two calls to one function represent a shared unit (protected by semaphore); we duplicate a function for two unshared units.
- Separate *transformer* to map within area-time spectrum.
- We have built a hardware compiler which translates SAFL to circuit designs (hierarchical RTL Verilog).

  Promising results: 3-4 page spec. of a commercial processor.

Transformations

UNIVERSITY OF CAMBRIDGE

Transformations move in the area-time spectrum:

- Fold-Unfold (Burstall & Darlington).
- Instructions ROMs for specialised processors (this paper).
- Specialised processors are themselves SAFL programs.
- Choice of mapping of (e.g. parallel) tasks to one processor (serialised or interleaved) or to multiple processors.
- RAM access allows 'unbounded' memory use at the cost of von Neumann bottleneck.
- All within a high-level language: SAFL expresses gross structure, but fine details filled in by compiler.

Language

- c constants,
- x variables (`let` and formal parameters)
- a primitive functions
- f user-defined functions

Terms e and programs p given by:

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \mid \\
 & & \texttt{let } \vec{x} = (e_1, \ldots, e_k) \texttt{ in } e_0 \mid \\
 & & a(e_1, \ldots, e_{arity(a)}) \mid f(e_1, \ldots, e_{arity(f)}) \\
p & ::= & \texttt{fun } f^1(\vec{x}) = e_1; \ \ldots; \ \texttt{fun } f^n(\vec{x}) = e_n;
\end{array}
$$

Calls in recursive cycles must be tail-calls (can relax in software).

Hardware-Software Partitioning

- Partitioning function π, a *partial function* in

$$\text{set-of-function-names} \to \text{set-of-processors}$$

  Names not mapped remain as hardware.

- A processor is a SAFL function M_i which refers to an instruction ROM IM_i (another SAFL function);
- IM_i contains binary code for $\pi^{-1}(M_i)$;
- Each M_i has an associated compiler from SAFL to code for M_i;
- The M_i can be heterogeneous.
- No automatic partitioning—the game is to be able to explore the area-time spectrum (including software/hardware implementation choices).

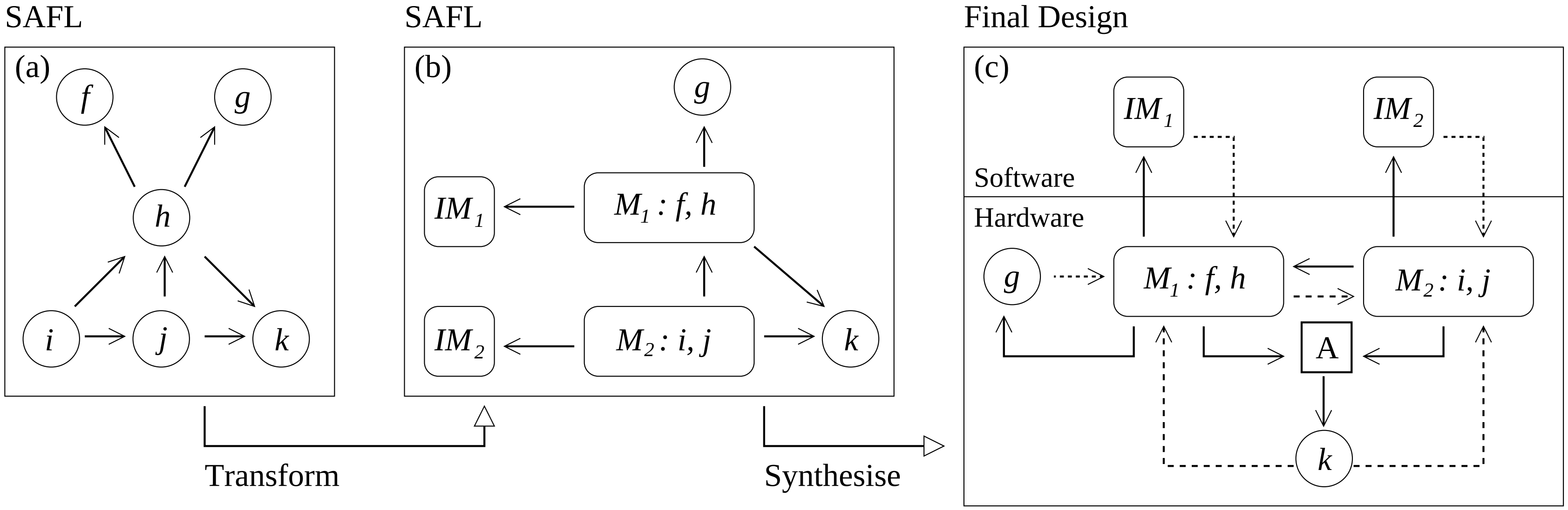
The Partitioning Transformation (1)

If P is a SAFL program we write $\hat{\pi}(P)$ for the transformed SAFL program.

- If P has function names S then $\hat{\pi}(P)$ has names $S \cup \{M_i\} \cup \{IM_i\} \setminus \mathrm{dom}(P)$

The Partitioning Transformation (2)

Consider $\pi = \{(f, M_1), (h, M_1), (i, M_2), (j, M_2)\}$.



(a) call-graph of a SAFL specification, P;

(b) call-graph of $\hat{\pi}(P)$ (new processor functions M_1 and M_2 and their instruction ROMs IM_1 and IM_2);

(c) circuit structure after compilation (box 'A' is an arbiter automatically inserted by the SAFL compiler to protect k).

Hardware Synthesis using SAFL (1)

UNIVERSITY OF CAMBRIDGE

The SAFL compiler compiles each function definition:

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

into a hardware block, H_f, with output port P_f, consisting of:

- a fixed amount of storage (registers holding values of the arguments $\vec{x}$) and
- a circuit to compute e to P_f (maybe completion signal too).

Hardware Synthesis using SAFL (2)

UNIVERSITY OF CAMBRIDGE

- Shared functional units (two or more callers) have multiplexers to route values from multiple call sites (and arbiters too, unless analysis can show there are no concurrent calls).
- Returned values are just distributed to all callers (if the implementation uses a completion signal then this is returned to the actual caller).

Our current implementation:

- "Activate and wait for result" or "Clocked asynchronous" model: global clock with completion signal after possibly computation-dependent delay.

Processor synthesis

UNIVERSITY OF CAMBRIDGE

- Works for functions representing processors (say $CPU()$) too;
- An external call invokes $CPU()$ with initial PC, SP and argument values;
- A `halt` instruction returns a value (e.g. top of SP) onto P_{CPU}.

A processor like $CPU()$ will tend to use instruction memory ROM. This is a also SAFL function $\mathtt{IM}_i$:

```
fun IMi(address) =
   case address of 0 => instruction0
                 | 1 => instruction1
                      ...  etc.
```

Invocation forms

UNIVERSITY OF CAMBRIDGE

Suppose f calls g and (f,g) are in:

- (hardware,hardware): as previous slides
- (hardware,software): as previous slides
- (software,hardware): compile a special instruction which copies the actual parameters to a port which activates H_g
- (software,software) same processor: use a typical CALL form (push old PC to SP and branch to new code address)
- (software,software) different processor:
  treat as (software, hardware) but invoke CPU holding g with initial register values (PC = entry-address-for g)

Return values work similarly.

Validity of partitioning

Not all partitionings are valid.

- Suppose $f()$ calls $g()$ which calls $h()$;
- Suppose $\pi(f) = \pi(h) = M_1$ and $\pi(g) = M_2$;
- Then M_1 and M_2 are recursive, but not in general tail recursive.
- Violates SAFL rules (recursive re-invocation of M_1)

Validity:

- π is a valid partition with respect to SAFL program, P, iff all cycles occurring in 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.

A stack-machine processor in SAFL

See attached slide.

Compiling SAFL into Stack Code (1)

$$[\![c]\!]\sigma \stackrel{\text{def}}{=} [\texttt{PushC}(c)]$$

$$[\![x]\!]\sigma \stackrel{\text{def}}{=} [\texttt{PushV}(\sigma(x))]$$

$$[\![f(e_1,\ldots,e_k)]\!]\sigma \stackrel{\text{def}}{=} \begin{cases} [\![e_1]\!]\sigma \ @\ [\![e_2]\!]\sigma^{+1} \ @\ \ldots\ @\ [\![e_k]\!]\sigma^{+(k-1)} \\ \quad @\ [\texttt{Call\_Ext}_f] \\ \qquad\qquad\qquad \text{if } f \notin \mathrm{dom}(\pi) \\ [\![e_1]\!]\sigma \ @\ [\![e_2]\!]\sigma^{+1} \ @\ \ldots\ @\ [\![e_k]\!]\sigma^{+(k-1)} \\ \quad @\ [\texttt{Call\_Int}(f_{entry})] \\ \qquad\qquad\qquad \text{if } f \in \mathrm{dom}(\pi) \wedge \pi(f) = \pi(g) \end{cases}$$

$$[\![a(e_1,e_2)]\!]\sigma \stackrel{\text{def}}{=} [\![e_1]\!]\sigma \ @\ [\![e_2]\!]\sigma^{+1} \ @\ [\texttt{Alu2}(a)]$$

Compiling SAFL into Stack Code (2)

UNIVERSITY OF CAMBRIDGE

$$
\begin{aligned}
[\![\texttt{fun}\ g(x_1, \ldots, x_k) = e]\!] \quad \stackrel{\text{def}}{=} \quad & [\mathit{label}\text{: } g_{entry}] \\
& @\ [\![e]\!]^g \emptyset \{x_k \mapsto 1, x_{k-1} \mapsto 2, \ldots, x_1 \mapsto k\} \\
& @\ [\texttt{Return}(k)] \\
& @\ [\mathit{label}\text{: } \mathit{ext\_g}\ _{entry}] \\
& @\ [\texttt{PushA}(1),\ \ldots,\ \texttt{PushA}(k), \\
& \quad \texttt{Call\_Int}(g_{entry}),\ \texttt{Halt}]
\end{aligned}
$$

Fine Grained Partitioning

How can we map *part* of a function definition to hardware and the rest to software:

```
f(x,y) = if x=0 then y
                 else f(x-1, x*y - 7 + 5*x)
```

Mapping `f` to software will use instructions:

multiply `x` and `y`, subtract 7, add 5 times `x`.

However, use fold/unfold first to get

```
i(x,y) = x*y-7 + 5*x
f(x,y) = if x=0 then y else f(x-1, i(x,y))
```

Now mapping `f` to software and `i` to hardware leads to a software representation for `f` containing fewer instructions and a specialised processor with a `x*y-7 + 5*x` instruction.

Removing tail recursion rules

Consider a new language, SAFL+ constructed by removing the recursion restrictions from SAFL.

We cannot compile this to hardware directly. However, provided we ensure that

- some processors M_i^+ can access an auxiliary stack held in RAM and
- (for each SAFL+ function f containing recursion in a non-tail-call context) we require π to map f to one of the M_i^+

then SAFL+ can be compiled to hardware and software.

Conclusions/More recent work

We can express algorithms at a high-level and later choose to implement then as hardware or software for a specialised CPU.

Transformations all happen within a single framework.

SAFL as is stands has an "invoke and wait for return" model. Current work extends SAFL to include functions which can interact using channels (so functions can return results or take additional arguments during a computation).