Hardware/Software Co-Design using Functional Languages

Alan Mycroft[1,2]
[1]Computer Laboratory
New Museums Site, Pembroke Street
Cambridge CB2 3QG, UK
am@cl.cam.ac.uk

Richard Sharp[2]
[2]AT&T Laboratories Cambridge
24a Trumpington Street
Cambridge CB2 1QA, UK
rws@uk.research.att.com
(Correspondence author)
Fax: +44 1223 313542

ABSTRACT

In previous work we have developed and prototyped a silicon compiler which translates a functional language (SAFL) into hardware. Here we present a SAFL-level program transformation which: (*i*) partitions a specification into hardware and software parts and (*ii*) generates a specialised architecture to execute the software part. The architecture consists of a number of interconnected heterogeneous processors. Our method allows a large design space to be explored by systematically transforming a single SAFL specification to investigate different points on the area-time spectrum.

1. INTRODUCTION

In [8] we introduced a hardware description language, SAFL (Statically Allocated Functional Language), and sketched its translation to hardware. An optimising silicon compiler for SAFL targetting hierarchical RTL Verilog has been implemented [13] and tested on a number of designs, including a small commercial processor[1]. SAFL is a first-order functional language with an ML [6] style syntax. We argue the case for functional languages over (say) Occam on the grounds of easier and more powerful program analysis, transformation and manipulation techniques. The essential features of SAFL can briefly be summarised as follows:

- programs are a sequence of function definitions;
- functions can call other defined functions but recursive calls must be tail-recursive[2]. (Section 2.5 addresses the exact technical restrictions.)

This allows our SAFL silicon compiler to:

- compile SAFL in a *resource-aware* manner. That is we map each function definition into a single hardware-level *resource*; functions which are called more than once become shared resources[3].
- synthesise highly parallel hardware—referential transparency allows one to evaluate all subexpressions in parallel.
- statically allocate the storage (e.g. registers and memories) required by a SAFL program.

The SAFL language is designed to facilitate source-to-source transformation. Whereas traditional "black-box" synthesis systems synthesise hardware according to user-supplied constraints, our approach is to select a particular implementation by applying transformation rules to the SAFL source as a pre-compilation phase. We have shown that applying fold/unfold transformations [2] to SAFL specifications allows one to explore various time-area tradeoffs at the hardware level [8; 9]. The purpose of this paper is to demonstrate how hardware/software partitioning can be seen as a source-to-source transformation at the SAFL level thus providing a formal framework in which to investigate hardware/software co-design. 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. There are a number of advantages to our approach:

- Synthesising an architecture specifically to execute a known piece of software can offer significant advantages over a fixed architecture [12].
- The ability to synthesise multiple processors allows a wide range of area-time tradeoffs 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

[1]The instruction set of Cambridge Consultants XAP processor (see `www.camcon.co.uk`) was implemented. We did not include the SIF instruction.

[2]Section 2.6.3 shows how this restriction can be removed by mapping general recursive functions into software.

[3]All sharing issues are handled automatically by our silicon compiler: arbiters are inserted where necessary to protect shared resources and data-validity analysis is performed facilitating the generation of efficient inter-resource interface logic [13].

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 Figure 1 for an example).

1.1 A brief overview of the SAFL language

SAFL is a language of first order recurrence equations; a user program consists of a sequence of function definitions:

$$\texttt{fun}\ f_1(\vec{x}) = e_1;\ \ \ldots\ ;\ \texttt{fun}\ f_n(\vec{x}) = e_n$$

Programs have a distinguished function, `main`, (usually 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. The abstract syntax of SAFL expressions, e, is as follows (we abbreviate tuples $(e_1, \ldots, e_k)$ as $\vec{e}$ and similarly $(x_1, \ldots, x_k)$ as $\vec{x}$):

- variables: x; constants: c;
- user function calls: $f(\vec{e})$;
- primitive function calls: $a(\vec{e})$—where a ranges over primitive operators (e.g. `+`, `-`, `<=`, `&&` etc.);
- conditionals: e_1 `?` e_2 `:` e_3; and
- let bindings: `let` $\vec{x} = \vec{e}$ `in` e_0 `end`

See Figures 3 and 4 for concrete examples of SAFL code.

1.2 Comparison with other work

Although there has been a lot of work on using functional languages to *describe* circuits concisely, e.g. muFP [14] and Lava [1] by Sheeran and Hydra [10] by O'Donnell, it seems as though the direct *compilation* of functional languages to hardware has been less well-studied. Our approach takes SAFL constructs (rather than gates) as primitives allowing our SAFL compiler to perform a number of high-level analyses and optimisations. (A more detailed comparison of SAFL with other hardware description languages can be found in [9]).

The idea of converting a program into a parameterised processor and corresponding instruction memory is not new; Page described a similar transformation [12] within the framework of Handel [11] (a subset of Occam for which a silicon compiler was written). However, the extra transformational power provided by our functional specification language allows us to generalise Page's work in a number of ways. Rather than synthesising a single parameterised processor our method allows one to generate a much more general architecture consisting of multiple communicating processors accessing a set of (potentially shared) hardware resources.

2. 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. Automated partitioning is not the subject of this paper; we assume that π is supplied by the user. For expository purposes we initially describe a transformation where all processors are variants of a stack machine: Section 2.1 describes the operation of the stack machine and Section 2.2 shows how it can be encoded as a SAFL function; a compiler from SAFL to stack code is presented in Section 2.3. In Section 2.6.2 we generalise our partitioning transformation to a network of heterogenous processors.

Let $\mathcal{M}$ be the set of processor instances 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 $f \notin \textit{Dom}(\pi)$ 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. Figure 1 shows the effect of a partitioning transformation, $\hat{\pi}$, where

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

In this example we see that g and k are implemented in hardware since $g, k \notin \textit{Dom}(\pi)$. $\hat{\pi}(P)$ contains function definitions: $\texttt{M}_1$, $\texttt{M}_2$, $\texttt{IM}_1$, $\texttt{IM}_2$, `g` and `k` where $\texttt{M}_1$ and $\texttt{M}_2$ are processor instances and $\texttt{IM}_1$ and $\texttt{IM}_2$ are instruction memories (see Section 2.2).

2.1 The Stack Machine Template

Our stack machine can be seen as a cut-down version of both Landin's SECD machine [5] and Cardelli's Functional Abstract Machine [3]. Each instruction has an op-code field and an operand field n. The following instructions are defined:

`PushC`(n) push constant n onto the stack;

`PushV`(n) push variable (from offset n into the current stack);

`PushA`(n) push the value of the stack machine's argument $\texttt{a}_n$ (see Section 2.2) to the stack;

`Squeeze`(n) pop top value; pop next n values; re-push top value;

`Return`(n) pop result; pop link; pop n arguments; re-push result; branch to link;

`Call_Int`(n) push address of next instruction onto stack and branch to address n;

`Jz`(n) pop a value; if it is zero branch to address n;

`Jmp`(n) jump to address n;

`Alu2`(n) pop two values; do 2-operand builtin operation n on them and push the result;

`Halt` terminate the stack machine returning the value on top of the stack.

We define a family of instructions to allow the stack machine to call external functions:

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 2.2); (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.

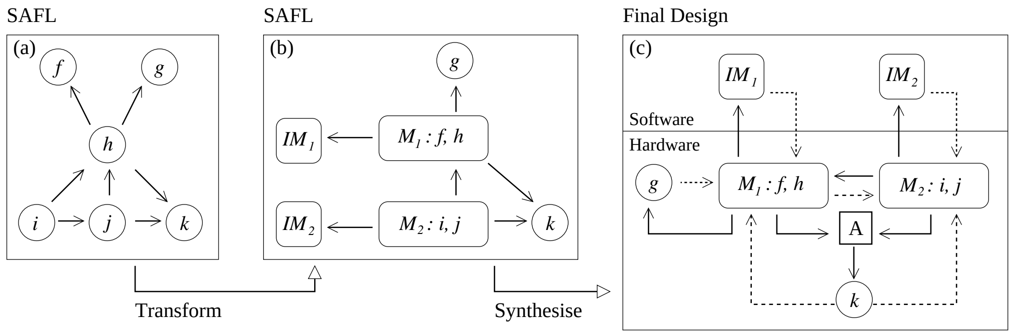

Figure 1: A diagrammatic view of the partitioning transformation

$\texttt{Call\_Ext}_f$ pops each of f's arguments from the stack; invokes the external function f and pushes the result to the top of the stack.

The stack machine *template*, SMT, is an abstract model of the stack machine parameterised on the code it will have to execute. Given a stack machine program, s, (i.e. a list of stack machine instructions as outlined above) $SMT\langle s\rangle$ is a stack machine *instance*: a SAFL function encoding a stack machine specialised for executing s. Our notion of a template is similar to a VHDL *generic*.

2.2 Stack Machine Instances

A stack machine instance, $\texttt{SM}_i \in \mathcal{M}$, is a SAFL function of the form:

$$\texttt{fun SM}_i\texttt{(a}_1\texttt{, ... , a}_{n_i}\texttt{, PC, SP) = ...}$$
$$\text{where } n_i = max(\{arity(f) \mid \pi(f) = \texttt{SM}_i\})$$

Arguments PC and SP are used to store the program counter and stack pointer respectively; $\texttt{a}_1, \ldots, \texttt{a}_{n_i}$ are used to receive arguments of functions located on $\texttt{SM}_i$. Each stack machine instance is associated with an instruction memory function, $\texttt{IM}_i$ of the form:

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

$\texttt{SM}_i$ calls $\texttt{IM}_i\texttt{(PC)}$ to load instructions for execution.
For example, consider a stack machine instance, $\texttt{SM}_{f,h}$, where we choose to locate functions f (of arity 2) and h (of arity 3). Then $n_{f,h} = 3$ yielding signature: $\texttt{SM}_{f,h}(\texttt{a}_1, \texttt{a}_2, \texttt{a}_3, \texttt{PC}, \texttt{SP})$. $\texttt{IM}_{f,h}$ is an instruction memory containing compiled code for f and h. To compute the value of $h(x, y, z)$ we invoke $\texttt{SM}_{f,h}$ with arguments $\texttt{a}_1 = x$, $\texttt{a}_2 = y$, $\texttt{a}_3 = z$, $\texttt{PC} = ext\_h_{entry}$ (h's external entry point—see Section 2.3) and $\texttt{SP} = 0$. Similarly to compute the value of $f(x, y)$ we invoke $\texttt{SM}_{f,h}$ with arguments $\texttt{a}_1 = x$, $\texttt{a}_2 = y$, $\texttt{a}_3 = 0$, $\texttt{PC} = ext\_f_{entry}$ and $\texttt{SP} = 0$. Note how we pad the a-arguments with 0's since $arity(f) < 3$.

The co-design of hardware and software means that instructions and ALU operations are only added to $\texttt{SM}_i$ if they appear in $\texttt{IM}_i$. Parameterising the stack machine 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 [12]. Figure 4 gives the SAFL code for a 16-bit stack machine instance[4]. An alu2 function, and an example stack machine program which computes triangular numbers is shown in Figure 3.

2.3 Compilation to Stack Code

Figure 2 gives a compilation function from SAFL to stack based code. Although the translation of many SAFL constructs is self-explanatory, the compilation rules for function definition and function call require further explanation:

2.3.1 Compiling Function Definitions

The code generated for function definition

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

requires explanation in that we create 2 distinct entry points for f: f_{entry} and ext_f_{entry}. The *internal* entry point, f_{entry}, is used when f is invoked internally (i.e. with a Call_Int instruction). The *external* entry point, ext_f_{entry}, 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, f_{entry}.

2.3.2 Compiling Function Calls

Suppose function g is in software ($g \in Dom(\pi)$) and calls function f. The code generated for the call depends on the location of f relative to g. There are three possibilities:

[4] Approximately 2000 2-input equivalent gates when compiled using the SAFL silicon compiler. For simplicity we consider a simple stack machine with no Call_Ext instructions.

1. If f and g are both implemented in software on the same machine ($f \in Dom(\pi) \wedge \pi(f) = \pi(g)$) then we simply push each of f's arguments to the stack and branch to f's internal entry point with a `Call_Int` instruction. The `Call_Int` instruction pushes the return address and jumps to f_{entry}; the compiled code for f is responsible for popping the arguments and link leaving the return value on the top of the stack.

2. If f is implemented in hardware ($f \notin Dom(\pi)$) then we push each of f's arguments to the stack and invoke the hardware resource corresponding to f by means of a $\texttt{Call\_Ext}_f$ instruction. The $\texttt{Call\_Ext}_f$ instruction pops each of f's arguments, invokes resource f and pushes f's return value to the stack.

3. If f and g are both implemented in software but on different machines ($f, g \in Dom(\pi) \wedge \pi(f) \neq \pi(g)$) then g needs to invoke $\pi(f)$ (the machine on which f is located). We push $\pi(f)$'s arguments to the stack: the arguments for f possibly padded by 0s (see Section 2.2) followed by the program counter `PC` initialised to ext_f_{entry} and the stack pointer `SP` initialised to 0. We then invoke $\pi(f)$ using a $\texttt{Call\_Ext}_{\pi(f)}$ instruction.

2.4 The Partitioning Transformation

Having introduced the stack machine (Section 2.1) and the associated compilation function (Section 2.3) 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:

   $$m(e_1, \ldots, e_k,\ \underbrace{0, \ldots, 0,}_{arity(m)-2-k}\ ext\_g_{entry}, 0)$$

   where $m = \pi(g)$, the stack machine instance on which g is located.

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, $\texttt{IM}_m$, 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 stack machine instance, $SMT\langle Code_m\rangle$ 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 Figure 1c).

2.5 Validity of Partitioning Functions

This section concerns some fine technical details—it can be skipped on first reading.
We clarify the SAFL restriction on recursion[5] given in the Introduction as follows.

In order for a SAFL program to be valid, all recursive calls, including those calls which form part of mutually-recursive cycle, may only occur in tail-context. Non-recursive calls may appear freely.

This allows storage for SAFL variables to be allocated statically as tail recursion does not require the dynamic allocation of stack frames.
Unfortunately, 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 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 $\pi = \{(f, \texttt{SM}),\ (h, \texttt{SM})\}$ yields a new program, $\hat{\pi}(P_{bad})$, of the form:

```
fun IM(PC) = ...
fun SM(x,PC,SP) = ... let t = <top-of-stack>
                          in g(t) ...
fun g(x) = SM(x, <ext_f_entry>, 0) + 2;
```

$\hat{\pi}(P_{bad})$ has invalid recursion between `g` and `SM`. The problem is that the call to `SM` 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 paper 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. stack machines and their instructions memories—must not have mutual recursion with any other functions.

[5] A more formal presentation can be found in [8].

Let σ, be an environment mapping variable names to stack offsets (offset 0 signifies the top of the stack). Let g be the name of the function we are compiling. Then $[\![\cdot]\!]^g\sigma$ gives an instruction list corresponding to g. (We omit g for readability in the following—it is only used to identify whether a called function is located on the same machine).
We use the notation $\sigma\{x \mapsto n\}$ to represent environment σ extended with x mapping to n. σ^{+n} represents an environment constructed by incrementing all stack offsets in σ by n—i.e. $\sigma^{+n}(x) = \sigma(x) + n$. $\emptyset$ is the empty environment. The infix operator @ appends instruction lists. $Repeat(l, n)$ is l @ ... @ l (n times); (this is used to generate instruction sequences to pad argument lists with 0s).

$$
\begin{array}{rcl}
[\![c]\!]\sigma & \stackrel{\text{def}}{=} & [\mathtt{PushC}(c)] \\
[\![x]\!]\sigma & \stackrel{\text{def}}{=} & [\mathtt{PushV}(\sigma(x))] \\
[\![f(e_1,\ldots,e_k)]\!]\sigma & \stackrel{\text{def}}{=} & \left\{ \begin{array}{ll}
[\![e_1]\!]\sigma \;@\; [\![e_2]\!]\sigma^{+1} \;@\; \ldots \;@\; [\![e_k]\!]\sigma^{+(k-1)} \;@\; [\mathtt{Call\_Ext}_f] & \text{if } f \notin Dom(\pi) \\[6pt]
[\![e_1]\!]\sigma \;@\; [\![e_2]\!]\sigma^{+1} \;@\; \ldots \;@\; [\![e_k]\!]\sigma^{+(k-1)} & \\
\quad @\; [\mathtt{Call\_Int}(f_{entry})] & \text{if } f \in Dom(\pi) \wedge \pi(f) = \pi(g) \\[6pt]
[\![e_1]\!]\sigma \;@\; [\![e_2]\!]\sigma^{+1} \;@\; \ldots \;@\; [\![e_k]\!]\sigma^{+(k-1)} & \\
\quad @\; Repeat([\mathtt{PushC}(0)], arity(\pi(f)) - 2 - k) & \\
\quad @\; [\mathtt{PushC}(ext\_f_{entry}), \mathtt{PushC}(0), \mathtt{Call\_Ext}_{\pi(f)}] & \text{if } f \in Dom(\pi) \wedge \pi(f) \neq \pi(g)
\end{array} \right. \\
[\![a(e_1,e_2)]\!]\sigma & \stackrel{\text{def}}{=} & [\![e_1]\!]\sigma \;@\; [\![e_2]\!]\sigma^{+1} \;@\; [\mathtt{Alu2}(a)] \\
[\![\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2]\!]\sigma & \stackrel{\text{def}}{=} & [\![e_1]\!]\sigma \;@\; [\![e_2]\!]\sigma^{+1}\{x \mapsto 0\} \;@\; [\mathtt{Squeeze}(1)] \\
[\![e_1\ ?\ e_2\ :\ e_3]\!]\sigma & \stackrel{\text{def}}{=} & \text{let } l \text{ and } l' \text{ be new labels in} \\
 & & \quad [\![e_1]\!]\sigma \;@\; [\mathtt{Jz}\ (l)] \;@\; [\![e_2]\!]\sigma \;@\; [\mathtt{Jmp}\ (l'), \textit{label: } l] \;@\; [\![e_3]\!]\sigma \;@\; [\textit{label: } l'] \\[10pt]
[\![\mathtt{fun}\ g(x_1,\ldots,x_k) = e]\!] & \stackrel{\text{def}}{=} & [\textit{label: } g_{entry}] \;@\; [\![e]\!]^g\emptyset\{x_k \mapsto 1, x_{k-1} \mapsto 2, \ldots, x_1 \mapsto k\} \;@\; [\mathtt{Return}(k)] \\
 & & \quad @\; [\textit{label: } ext\_g_{entry},\ \mathtt{PushA}(1),\ \ldots,\ \mathtt{PushA}(k),\ \mathtt{Call\_Int}(g_{entry}),\ \mathtt{Halt}]
\end{array}
$$

Figure 2: Compiling SAFL into Stack Code for Execution on a Stack Machine Instance

2.6 Extensions

2.6.1 Fine Grained Partitioning

We have presented a program transformation to map function definitions to hardware or software, but what if we want to map *part* of a function definition to hardware and the rest to software? This can be achieved by applying fold/unfold transformations before our partitioning transformation. For example, consider the function

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

If we choose to map `f` to software our design will contain a processor and associated machine code consisting of a sequence of instructions representing multiply `x` and `y`, subtract 7, add 5 times `x`. However, consider transforming `f` with a single application of the *fold*-rule [2]:

```
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.

2.6.2 Dealing with Heterogeneous Processors

So far we have only considered executing software on a network of stack machines. Our framework can be extended to handle a network of heterogeneous processors as follows:
Let *Templates* be a set of processor templates (c.f. the stack machine template, *SMT*, in section 2.1).
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:

$$
\begin{array}{rcl}
\delta & : & \mathcal{M} \rightarrow \textit{Templates} \\
\tau & : & \mathcal{M} \rightarrow \textit{Compilers}
\end{array}
$$

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 2.4 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, $MT = \delta(m)$, to generate processor instance $MT\langle MCode_m\rangle$ and append this to $\hat{\pi}_{\delta,\tau}(P)$.

2.6.3 Extending the SAFL Language

Recall that the SAFL language specifies that all recursive calls must be in tail-context. Since only tail-recursive calls are permitted, our silicon compiler is able to statically allocate all the storage needed for a SAFL program.
As an example of these restrictions consider the following definitions of the factorial function:

```
rfact(x) = if x=0 then 1 else x*rfact(x-1)
ifact(x,a) = if x=0 then a else ifact(x-1,x*a)
```

`rfact` is not a valid SAFL program since the recursive call is not in a tail-context. However the equivalent tail-recursive factorial function, `ifact` which uses a second argument to accumulate partial results is a valid SAFL program.

Although one can sometimes transform a non-tail recursive program into an equivalent tail-recursive one, this is not always easy or natural. The transformation of factorial into its tail-recursive equivalent is only possible because multiplication is an associative operator. Thus, in general we require a way of extending SAFL to handle general unrestricted recursion. Our partitioning transformation provides us with one way to do this:

Consider a new language, SAFL+ constructed by removing the recursion restrictions from SAFL. We can use our partitioning transformation to transform SAFL+ to SAFL simply by ensuring that each function definition containing recursion other than in a tail-call context is mapped to software. Note that our compilation function (Figure 2) is already capable of dealing with general recursion without any modification.

3. CONCLUSIONS AND FURTHER WORK

Source-level program transformation of a high level HDL is a powerful technique for exploring a wide range of architectural tradeoffs from an initial specification. The partitioning transformation outlined here is applicable to any hardware description language (e.g. VHDL or Verilog) given suitable compilation functions and associated processor templates. However, we believe that our methods are particularly powerful in the SAFL domain for two reasons:

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

We are in the process of deploying the techniques outlined here as part of a semi-automated transformation system for SAFL programs. The goal of the project is to develop a framework in which a SAFL program can be systematically transformed to investigate a large number of possible implementations of a single specification. So far we have developed a library of transformations which allow us to represent a wide range of concepts in hardware design including: resource sharing/duplication, static/dynamic scheduling [9] and now hardware/software partitioning. In the future we plan to investigate how partial evaluation techniques [4] can be used to transform a processor definition function and its corresponding instruction memory function into a single unit with hardwired control.

Although initial results have been promising, the project is still in its early stages. We are currently investigating ways of extending the SAFL language to make it more expressive without loosing too many of its mathematical properties. Our current ideas centre around adding synchronous communication and a restricted form of π-calculus [7] style channel passing. We believe that this will allow us to capture the semantics of I/O whilst maintaining the correspondence between high-level function definitions and hardware-level resources.

4. ACKNOWLEDGEMENTS

This work is part of a collaborative project, "Self-Timed Microprocessors", involving Cambridge University Computer Laboratory (EPSRC grant GR/L86326), AT&T Laboratories Cambridge and Cambridge Consultants Ltd.

5. REFERENCES

[1] Bjesse, P., Claessen, K., Sheeran, M. and Singh, S. Lava: Hardware Description in Haskell. Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, 1998.

[2] Burstall, R.M. and Darlington, J. A Transformation System for Developing Recursive Programs, JACM 24(1).

[3] Cardelli, L. The Functional Abstract Machine. Technical Report TR-107, AT&T Bell Laboratories, April 1983.

[4] Jones, N., Gomard, C. and Sestoft, P. Partial Evaluation and Automatic Program Generation. Published by Prentice Hall (1993); ISBN 0-13-020249-5.

[5] Landin, P. The Mechanical Evaluation of Expressions. Computer Journal, Vol. 6, No. 4, 1964, pages 308-320.

[6] Milner, R., Tofte, M., Harper, R. and MacQueen, D. The Definition of Standard ML (Revised). MIT Press, 1997.

[7] Milner, R. The Polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, October 1991.

[8] Mycroft, A. and Sharp, R. A Statically Allocated Parallel Functional Language. Proc. of the International Conference on Automata, Languages and Programming 2000. LNCS Vol. 1853, Springer-Verlag.

[9] Mycroft, A. and Sharp, R. The FLaSH Project: Resource-Aware Synthesis of Declarative Specifications. Proceedings of The International Workshop on Logic Synthesis 2000. Also available as AT&T Technical Report tr.2000.6 via `www.uk.research.att.com`

[10] O'Donnell, J. Generating Netlists from Executable Circuit Specifications in a Pure Functional Language. In *Functional Programming Glasgow*, Springer-Verlag Workshops in Computing, pages 178-194, 1993.

[11] Page, I. and Luk, W. Compiling Occam into Field-Programmable Gate Arrays. In Moore and Luk (eds.) FPGAs, pages 271-283. Abingdon EE&CS Books, 1991.

[12] Page, I. Parameterised Processor Generation. In Moore and Luk (eds.), More FPGAs, pages 225-237. Abingdon EE&CS Books, 1993.

[13] Sharp, R. and Mycroft, A. The FLaSH Compiler: Efficient Circuits from Functional Specifications. AT&T Technical Report tr.2000.3. Available from `www.uk.research.att.com`

[14] Sheeran, M. muFP, a Language for VLSI Design. Proc. ACM Symp. on LISP and Functional Programming, 1984.

```
(* +----------------------------------------------------------------------+
   |                                                                      |
   | SAFL specification of simple stack processor                        |
   |    Richard Sharp and Alan Mycroft, July 2000                         |
   |                                                                      |
   +----------------------------------------------------------------------+ *)


(* -------------------------------- ALU ------------------------------------ *)

fun alu2(op:16, a1:16, a2:16):16 =
   case op of 0 => a1+a2
            ! 1 => a1-a2
            ! 2 => a1&&a2
            ! 3 => a1||a2
            ! 4 => a1^^a2
            ! 16 => a1<a2
            ! 17 => a1>a2
            ! 18 => a1=a2
            ! 19 => a1>=a2
            ! 20 => a1<=a2
            ! 21 => a1<>a2

(* ---------------- Instruction memory here  --------------------- *)

(* The following codes: f(x) = if x then x+f(x-1) else 0;   *)
(* i.e. it computes triangular numbers                      *)
fun load_instruction (address:16):24 = case address of
       0 => %000010010000000000000001 (*     pusha 1    *)
     ! 1 => %000001010000000000000011 (*     call_int f *)
     ! 2 => %000000000000000000000000 (*     halt       *)
     ! 3 => %000000100000000000000001 (* f:  pushv 1    *)
     ! 4 => %000001110000000000001100 (*     jz l1      *)
     ! 5 => %000000100000000000000001 (*     pushv 1    *)
     ! 6 => %000000100000000000000010 (*     pushv 2    *)
     ! 7 => %000000010000000000000001 (*     pushc 1    *)
     ! 8 => %000010000000000000000001 (*     alu2 sub   *)
     ! 9 => %000001010000000000000011 (*     call_int f *)
     ! 10=> %000010000000000000000000 (*     alu2 add   *)
     ! 11=> %000001100000000000001101 (*     jmp l2     *)
     ! 12=> %000000010000000000000000 (* l1: pushc 0    *)
     ! 13=> %000001000000000000000001 (* l2: return 1   *)
 default => %101010101010101010101010 (*     illop      *)

external mem_acc (address:16,data:16,write:1):16

inline fun data_read  (address:16):16 = mem_acc(address,0,0)
inline fun data_write (address:16,data:16):16 = mem_acc(address,data,1)
```

Figure 3: The Stack Machine (Part 1 of 2)

```
(* --------------------- Stack Machine Instance  ------------------------ *)

fun SMachine (a1:16, PC:16, SP:16):16 =

   let var new_PC  : 16  = PC + 1
       var instr   : 24  = load_instruction(PC)
       var op_code : 8   = instr[23,16]
       var op_rand : 16  = instr[15,0]

       var inc_SP : 16  = SP + 1
       var dec_SP : 16  = SP - 1
    in
       case op_code of
            0 => (* halt, returning TOS *)
                    data_read(SP)
          ! 1 => (* push constant operation *)
                    data_write(dec_SP, op_rand);
                    SMachine (a1, new_PC, dec_SP)
          ! 2 => (* push variable operation *)
                    let var data:16 = data_read(SP+op_rand)
                    in  data_write(dec_SP, data);
                        SMachine (a1, new_PC, dec_SP) end
          ! 9 => (* push a-argument operation *)
                    data_write(dec_SP, a1);
                    SMachine (a1, new_PC, dec_SP)
          ! 3 => (* squeeze operation -- op_rand is how many locals to pop *)
                    let var new_SP:16 = SP + op_rand
                        var v:16 = data_read(SP)
                    in  data_write(new_SP, v);
                        SMachine (a1, new_PC, new_SP) end
          ! 4 => (* return operation -- op_rand is how many actuals to pop *)
                    let var new_SP:16 = inc_SP + op_rand
                        var rv:16 = data_read(SP)
                    in  let var rl:16 = data_read(inc_SP)
                        in  data_write(new_SP, rv);
                            SMachine (a1, rl, new_SP) end end
          ! 5 => (* call_int operation *)
                    data_write(dec_SP, new_PC);
                    SMachine (a1, op_rand, dec_SP)
          ! 6 => (* jmp (abs) operation *)
                    SMachine (a1, op_rand, SP)
          ! 7 => (* jz (abs) operation *)
                    let var v:16 = data_read(SP)
                    in  SMachine (a1, if v=0 then op_rand else new_PC, inc_SP) end
          ! 8 => (* alu2: binary alu operation -- specified by immediate field *)
                    let var v2:16 = data_read(SP)
                    in  let var v1:16 = data_read(inc_SP)
                        in  data_write(inc_SP, alu2(op_rand, v1, v2));
                            SMachine (a1, new_PC, inc_SP) end end
         default =>
               (* halt, returning 0xffff -- illegal opcode *)
                    %1111111111111111
   end
```

Figure 4: The Stack Machine (Part 2 of 2)