Soft Scheduling for Hardware

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

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

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

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

Abstract. Hardware designs typically combine parallelism and resource-sharing; a circuit's correctness relies on shared resources being accessed mutually exclusively. Conventional high-level synthesis systems guarantee mutual exclusion by statically serialising access to shared resources during a compile-time process called *scheduling*. This approach suffers from two problems: (*i*) there is a large class of practical designs which cannot be scheduled statically; and (*ii*) a statically fixed schedule removes some opportunities for parallelism leading to less efficient circuits. This paper surveys the expressivity of current scheduling methods and presents a new approach which alleviates the above problems: first scheduling logic is automatically generated to resolve contention for shared resources dynamically; then static analysis techniques remove redundant scheduling logic.
We call our method Soft Scheduling to highlight the analogy with Soft Typing: the aim is to retain the flexibility of dynamic scheduling whilst using static analysis to remove as many dynamic checks as possible.

1 Introduction

At the structural level a hardware design can be seen as a set of interconnected resources. These resources run concurrently and are often shared.

The interaction between parallelism and resource-sharing leads to an obvious problem: how does one ensure that shared resources are accessed mutually exclusively? Existing silicon compilers solve the mutual exclusion problem by statically serialising operations during a compile-time *scheduling* phase (see Section 1.1). This paper describes an alternative approach:

> We automatically generate circuitry to perform scheduling *dynamically* in a manner which avoids deadlock. Efficient circuits are obtained by employing *static* analysis to remove redundant scheduling logic.

Our method is to scheduling as Soft Typing [4] is to type checking (see Figure 1): the aim is to retain the flexibility of dynamic scheduling whilst using

	Typing	Scheduling
Static	No dynamic checks required in object code. Not all valid programs pass type checker.	No scheduling logic required in final circuit. Not all valid programs can be scheduled statically.
Dynamic	Dynamic checking of argument types required each time a function is called. All valid programs can be run.	Scheduling logic required on each shared resource in the final circuit. All valid programs can be scheduled.
Soft	Fewer dynamic checks required (some removed statically). All valid programs can be run.	Less scheduling logic required (some removed statically). All valid programs can be scheduled.

Fig. 1. An Informal Comparison Between Soft Scheduling and Soft Typing

static analysis to remove as many dynamic checks as possible. To highlight this analogy we choose to call our method *Soft Scheduling*.

Although this paper considers the application of Soft Scheduling to hardware synthesis, the technique is also applicable to software compilation. Aldrich *et al.* [1] advocate a similar approach which uses static analysis to remove redundant synchronization from Java programs.

1.1 Conventional High-Level Synthesis

The hardware community refer to high-level, block-structured languages as *behavioural*. At a lower level, *structural* languages describe a circuit as a set of components, such as registers and multiplexers connected with wires and buses (e.g. RTL Verilog [9]). *High-level synthesis* (sometimes referred to as *behavioural synthesis*) is the process of compiling a behavioural specification into a structural hardware description language.

A number of behavioural synthesis systems have been developed for popular high-level languages (e.g. CtoV [20] and Handel [17]). Such systems typically translate high-level specifications into an explicitly parallel flow-graph representation where *allocation*, *binding* and *scheduling* [6] are performed:

- *Allocation* is typically driven by user-supplied directives and involves choosing which resources will appear in the final circuit (e.g. 3 adders, 2 multipliers and an ALU).
- *Binding* is the process of assigning operations in the high-level specification to low-level resources—e.g. the `+` in line 4 of the source program will be computed by `adder_1` whereas the `+` in line 10 will be computed by the `ALU`.
- *Scheduling* involves assigning start times to operations in the flow-graph such that no two operations will attempt to access a shared resource simultaneously. Mutually-exclusive access to shared resources is ensured by statically serialising operations during scheduling.

The Contributions of This Paper

1. In contrast to conventional scheduling, we describe a method which generates logic to schedule shared resources dynamically. We show that (*i*) our approach is more expressive: all valid programs can be scheduled; and (*ii*) in some cases, our approach can generate more efficient designs by exploiting parallelism possibilities removed by static scheduling.
2. We describe a high-level static analysis that enables us to remove redundant scheduling logic and show that this can significantly improve the efficiency of generated circuits.

We have implemented Soft Scheduling as part of the FLaSH Synthesis System (see Section 1.2)—a novel hardware synthesis package being developed in conjunction with Cambridge University and AT&T Laboratories Cambridge. This paper presents Soft Scheduling in the framework of the FLaSH silicon compiler.

1.2 The FLaSH Synthesis System

In previous work we introduced a hardware description language, SAFL [15] (Statically Allocated Functional Language), and sketched its translation to hardware. An optimising silicon compiler (called FLaSH [18]—Functional Languages for Synthesising Hardware) has been implemented to translate SAFL into hierarchical RTL Verilog. The system has been tested on a number of designs, including a small commercial processor[1].

Although, for expository purposes, this paper describes our method in the framework of SAFL, Soft Scheduling techniques are applicable to any high-level Hardware Description Language which allows function definitions to be treated as shared resources (e.g. HardwareC [11], Balsa [7], Tangram [2]). Indeed we have extended SAFL with π-calculus [14] style channels and assignment without modifying the Soft Scheduling phase [19].

Outline of Paper The remainder of this paper is structured as follows: Section 2 surveys existing scheduling techniques and explains the motivation for our research; the SAFL language and its translation to hardware are briefly outlined in Section 3; Section 4 presents the technical details of Soft Scheduling; some practical examples are described in Section 5.

2 Comparison With Other Work

Traditional high-level synthesis packages perform scheduling using a data-structure called a *sequencing graph*—a partial ordering which makes dependencies between

[1] The instruction set of Cambridge Consultants XAP processor was implemented (see `www.camcon.co.uk`). We did not include the SIF instruction (a form of debugging breakpoint which transfers data to or from internal registers via a separately clocked serial interface).

operations explicit. Recall that, in this context, scheduling is performed by assigning a start time to each operation in the graph such that operations which invoke a shared resource do not occur in parallel [6]. There are a number of problems with this approach:

1. The time taken to execute each operation in the sequencing graph must be bounded statically (and in general padded to this length). This restriction means that conventional scheduling techniques are not expressive enough to handle a large class of practical designs. For example, it is impossible to statically schedule an operation to perform a bus transaction of unknown length.
2. Since operations are scheduled statically one must be pessimistic about what *may* be invoked in parallel in order to achieve safety. This can inhibit parallelism in the final design by unnecessarily serialising operations.

Ku and De Micheli have proposed Relative Scheduling [10] which extends the method outlined above to handle operations with statically unbounded computation times. Their technique partitions a flow-graph into statically-schedulable segments separated by *anchor nodes*—nodes which have unbounded execution delays. Each segment is scheduled separately at compile-time. Finally, the compiler connects segments together by generating logic to signal the completion of anchor nodes dynamically.

In [12] Ku and De Micheli show how Relative Scheduling of shared resources is integrated into their Olympus Hardware Synthesis System [5]. Their method permits the scheduling of operations whose execution time is not statically bounded, hence alleviating Problem 1 (above). However, potential contention for shared resources is still resolved by serialising operations at compile time so Problem 2 remains. Furthermore, there is still a class of practical designs which cannot be scheduled by Olympus. Consider the following example.

Using `||` as a parallel composition operator and assuming suitable definitions of procedures `Processor`, `DMA_Controller` and `Memory` we would essentially like to describe the system of Figure 2 as:

```
Processor() || DMA_Controller() || Memory()
```

Since the operations corresponding to the invocation of the `Processor` and `DMA_Controller` both access a shared resource (`Memory`) the Olympus Synthesis System requires that the calls must be serialised. However, if neither the call to `Processor()` nor the call to `DMA_Controller` terminate[2], attempting to sequentialise the operations is futile; the correct operation of the system relies on their parallel interleaving. Soft Scheduling is expressive enough to cope with non-terminating operations: the FLaSH compiler automatically generates an arbiter to ensure mutually exclusive access to the `Memory` whilst allowing the `Processor` and `DMA_Controller` to operate in parallel (see Figure 2.ii). The following table summarises the expressivity of various scheduling methods.

[2] This is not merely a contrived example. In real designs both Processors and DMA Controllers are typically non-terminating processes which constantly update the machine state.

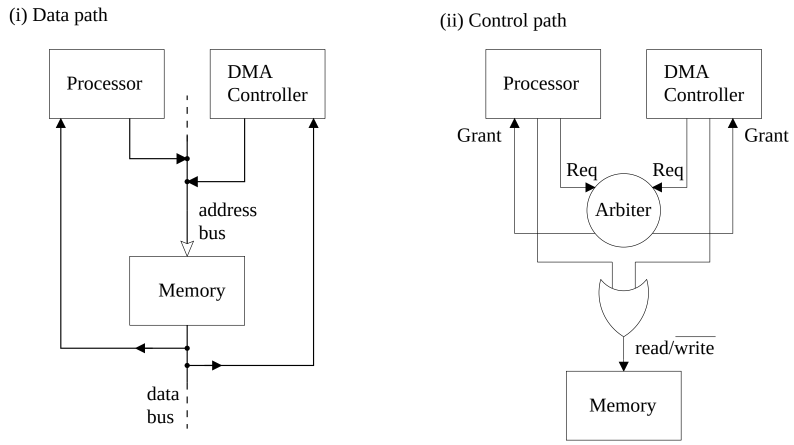

Fig. 2. A hardware design containing a memory device shared between a DMA controller and a processor

	Bounded execution delays	Unbounded execution delays	Non-terminating operations
Static	✓	✗	✗
Relative	✓	✓	✗
Soft	✓	✓	✓

Although the technique of using arbiters to protect shared resources is widely employed, current hardware synthesis packages require arbitration to be coded at the *structural* level on an *ad hoc* basis. Since arbitration can impose an overhead both in terms of chip area and time, programmers often try to eliminate unnecessary locking operations manually. For large designs this is a tedious and error-prone task which often results in badly structured and less reusable code. In contrast the Soft Scheduling approach analyses a *behavioural* specification, automatically inserting arbiters on a where-needed basis. This facilitates readable and maintainable source code without sacrificing efficiency.

This paper does not discuss the SAFL language in depth. A detailed comparison of SAFL with other hardware description languages including Verilog, VHDL, MuFP, Lava, ELLA and Lustre can be found in [18].

3 An Overview of The SAFL Language

SAFL is a language of first order recurrence equations with an ML [13] style syntax. 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`

In order to distinguish distinct call sites we assume that each abstract-syntax node is labelled with a unique identifier, α, writing $f^{\alpha}(e_1, \ldots, e_k)$ to indicate a call to function f at abstract-syntax node α.

Although functions can call other previously defined functions arbitrarily, the only form of recursion allowed is tail-recursion. This allows us to statically allocate the storage (e.g. registers and memories) required by a SAFL program [15]. Tail recursive calls are compiled into feedback loops at the circuit level.

SAFL is a call-by-value language. All function-call arguments and `let`-definiens are evaluated in parallel. Operations can be sequenced using the `let` construct since the language semantics state that all `let`-declarations must terminate before the `let`-body is evaluated.

We compile SAFL to hardware in a *resource aware* manner. That is each function definition is mapped into a single hardware-level *resource*; functions which are called more than once become shared resources. For example, consider the following SAFL code:

```
fun mult(x, y, acc) =
  if (x=0 or y=0) then acc
     else mult(x<<1, y>>1, if y.bit0 then acc+x else acc)

fun square(x) = mult(x, x, 0)
fun cube(x)   = mult(x, mult(x, x, 0), 0)
```

This SAFL specification describes a circuit containing a *single* shift-add multiplier shared between hardware-blocks to compute squares and cubes. Notice how in contrast to traditional high-level synthesis (see Section 1.1) the resource aware interpretation of SAFL specifications explicitly contains allocation and binding information. (Although not of direct relevance to this paper, in [15] we show how *fold/unfold* transformations [3] can be used to explore various allocation and binding constraints.)

3.1 Translating SAFL to Hardware

As in Relative Scheduling [10] the FLaSH compiler generates logic to explicitly signal the completion of operations. More precisely, each SAFL function definition, f, is compiled into a single resource, H_f, consisting of:

– logic to compute its body expression
– multiple control and data inputs: one control/data input-pair for each call site
– multiple control outputs (one to return control to each caller)
– a single data output (which is shared between all callers)

An example of function connectivity is given in Figure 3. In this example resource H_f 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. Full details of the translation to hardware can be found in [18].

4 Soft Scheduling: Technical Details

To protect shared resources the FLaSH compiler automatically generates scheduling logic to resolve conflicts dynamically (see Figure 3). The scheduling circuitry consists of two parts: (i) an arbiter to select which caller to service; and (ii) a locking mechanism to ensure the resource is accessed mutually exclusively. For the sake of brevity, this paper uses the term *arbiter* to refer to both the arbiter and locking structure.

Our approach is the hardware equivalent of using binary semaphores to protect critical regions in multi-threaded software. The analogy between arbiters and semaphores is explored further in [15] where a compilation function from SAFL to software is presented.

4.1 Removing Redundant Arbiters

Just because a resource is shared does not necessarily mean that arbitration is required. For example consider the following SAFL program:

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

In this case, the two calls to `f` cannot occur in parallel: the innermost call must complete before the outermost call can begin (recall that SAFL is a call-by-value language). We do not need to generate an arbiter to serialise the calls to $H_{\mathtt{f}}$: from the structure of the program we can statically determine that the two calls will not try to access `f` simultaneously.

We use *Parallel Conflict Analysis* (see Section 4.2) in order to detect redundant arbiters. Removing unnecessary arbitration is important for two reasons:

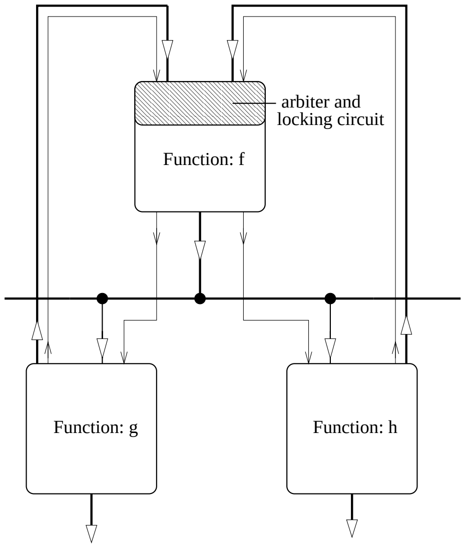

Fig. 3. A structural diagram of the hardware circuit corresponding to a shared function, f, called by functions g and h. Data buses are shown as thick lines, control wires as thin lines.

1. Arbitration takes time: in the current version of the FLaSH compiler arbitration adds one cycle latency to a call even if the requested resource is available at the time of call. Although we may accept this latency if it is small in comparison to the callee's average execution time, consider the case where the callee is a frequently used resource with a small execution delay. In this case an arbiter may significantly degrade the performance of the whole system (see Example 5.1).
2. Arbitration uses chip area: although the gate-count of an arbiter is typically small compared to the resource as a whole, the extra wiring complexity required to represent request and grant signals adds to the area of the final design.

Arbiters are inserted at the granularity of calls. This offers increased performance over inserting arbiters on a per-resource basis. For example, in a design containing a function, f, shared between five callers, we may infer that only two calls to f require an arbiter—the other three calls need not suffer the overhead of arbitration.

4.2 Parallel Conflict Analysis (PCA)

Parallel Conflict Analysis (PCA) is performed over the structure of a whole SAFL program in order to determine which function calls may occur in parallel. If a group of calls to the same function may occur in parallel then we say that the group is *conflicting*. We only need to synthesise logic to arbitrate between conflicting calls since, by definition, if a call f^α is not in a conflicting group then no other call to f can occur in parallel with f^α.

The result of PCA is a *conflict set*: a set of calls which require arbiters. For example, if the resulting conflict set is $\{f^1, f^2, f^5, g^{10}, g^{14}\}$ then we would synthesise two arbiters: one for the conflicting group $\{f^1, f^2, f^5\}$, the other for conflicting group $\{g^{10}, g^{14}\}$.

We now proceed to define PCA. Let e_f represent the body of function f. Let the predicate $\mathit{RecursiveCall}(f^\alpha)$ hold iff f^α is a recursive call (i.e. f^α occurs within the body of f). $\mathcal{C}[\![e]\!]$ returns the set of non-recursive calls which may occur as a result of evaluating expression e:

$$\begin{aligned}
\mathcal{C}[\![x]\!] &= \emptyset \\
\mathcal{C}[\![c]\!] &= \emptyset \\
\mathcal{C}[\![a(e_1,\ldots,e_k)]\!] &= \bigcup_{1\le i\le k} \mathcal{C}[\![e_i]\!] \\
\mathcal{C}[\![f^\alpha(e_1,\ldots,e_k)]\!] &= (\bigcup_{1\le i\le k} \mathcal{C}[\![e_i]\!]\ ) \cup \begin{cases} \emptyset & \text{if } \mathit{RecursiveCall}(f^\alpha) \\ \{f^\alpha\} \cup \mathcal{C}[\![e_f]\!] & \text{otherwise} \end{cases} \\
\mathcal{C}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] &= \bigcup_{1\le i\le 3} \mathcal{C}[\![e_i]\!] \\
\mathcal{C}[\![\texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0]\!] &= \bigcup_{0\le i\le k} \mathcal{C}[\![e_i]\!]
\end{aligned}$$

$PC(\mathcal{S}_1,\ldots,\mathcal{S}_n)$ takes sets of calls, $(\mathcal{S}_1,\ldots,\mathcal{S}_n)$, and returns the conflict set resulting from the assumption that calls in each $\mathcal{S}_i$ are evaluated in parallel with calls in each $\mathcal{S}_j$ $(j \neq i)$:

$$PC(\mathcal{S}_1,\ldots,\mathcal{S}_n) = \bigcup_{i\neq j} \{f^\alpha \in \mathcal{S}_i \mid \exists\beta.\ f^\beta \in \mathcal{S}_j\}$$

We are now able to define $\mathcal{A}[\![e]\!]$ which returns the conflict set due to expression e:

$$\begin{aligned}
\mathcal{A}[\![x]\!] &= \emptyset \\
\mathcal{A}[\![c]\!] &= \emptyset \\
\mathcal{A}[\![a(e_1,\ldots,e_k)]\!] &= PC(\mathcal{C}[\![e_1]\!],\ldots,\mathcal{C}[\![e_k]\!]) \cup \bigcup_{1\le i\le k} \mathcal{A}[\![e_i]\!] \\
\mathcal{A}[\![f(e_1,\ldots,e_k)]\!] &= PC(\mathcal{C}[\![e_1]\!],\ldots,\mathcal{C}[\![e_k]\!]) \cup \bigcup_{1\le i\le k} \mathcal{A}[\![e_i]\!] \\
\mathcal{A}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] &= \bigcup_{1\le i\le 3} \mathcal{A}[\![e_i]\!] \\
\mathcal{A}[\![\texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0]\!] &= PC(\mathcal{C}[\![e_1]\!],\ldots,\mathcal{C}[\![e_k]\!]) \cup \bigcup_{0\le i\le k} \mathcal{A}[\![e_i]\!]
\end{aligned}$$

Finally, for a program, p, consisting of a sequence of user-function definitions:

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

$\mathcal{A}[\![p]\!]$ returns the conflict set resulting from program, p. The letter $\mathcal{A}$ is used since $\mathcal{A}[\![p]\!]$ represents the calls which require arbiters:

$$\mathcal{A}[\![p]\!] = \bigcup_{1 \le k \le n} \mathcal{A}[\![e_k]\!]$$

Notice that the equation for $\mathcal{C}[\![f^\alpha(e_1, \ldots, e_k)]\!]$ is a little unusual in that it is not defined compositionally. This reflects the fact that PCA depends on the global structure of a whole SAFL program as opposed to just the local structure of a function definition. $\mathcal{C}[\![\cdot]\!]$ is well-defined due to the predicate *RecursiveCall* and the source restrictions on SAFL which ensure that the call-graph is acyclic.

4.3 Integrating PCA into the FLaSH Synthesis System

After computing $\mathcal{A}[\![p]\!]$ at the abstract-syntax level the FLaSH Synthesis System translates p into an intermediate flow-graph representation which makes both control and data paths explicit [18]. At this level, the `Call`-nodes which require arbitration are tagged (i.e. we tag node, n, iff n represents `Call` f^α and $f^\alpha \in \mathcal{A}[\![p]\!]$).

When the circuit for H_f is generated only tagged calls to f are fed through an arbiter, other calls are merely multiplexed. If none of the calls to f are in $\mathcal{A}[\![p]\!]$ then H_f's arbiter is eliminated completely. As Section 5.1 shows, using Parallel Conflict Analysis to remove redundant arbitration can significantly improve the performance of a large class of designs.

4.4 Avoiding Deadlock

Deadlock occurs when there is a cycle of blocked processes each waiting for a lock held by the next process in the cycle. In the context of SAFL, where functions represent hardware-level resources, a deadlocked cycle of resources can only occur if we permit cycles in the call-graph (i.e. if we permit mutual recursion). Note that we do not have to worry about self-tail-recursion since it is simply treated as local loops and does not require locks.

Although the details are beyond the scope of this paper, in [15] we show how to deal with mutual recursion whilst avoiding deadlock. For the purposes of this paper it suffices to say that deadlock can be avoided simply by rejecting mutually recursive SAFL programs.

5 Examples and Results

We provide three practical examples of applying Soft Scheduling to SAFL hardware designs. Each example illustrates a different point: Example 5.1 demonstrates that using static analysis to remove redundant arbiters is critical to achieving efficient circuits; Example 5.2 highlights the extra expressivity of Soft

Scheduling over static scheduling techniques; Example 5.3 shows that dynamically controlling access to shared resources can lead to better performance than generating a single schedule statically.

5.1 Parallel FIR Filter

Finite Impulse Response (FIR) filters are commonly used in Digital Signal Processing where they are used to remove certain frequencies from a discrete-time sampled signal. Assuming the existence of functions `read_next_value` and `write_value`, an integer arithmetic FIR filter can be described in SAFL as follows:

```
fun mult1(x,y) = x*y
fun mult2(x,y) = x*y

fun FIR(x,y,z,w) =
   let val o1 = mult1(x,2)
       val o2 = mult2(y,3)
       val next = read_next_value()
    in
       let val o3 = mult1(x,7)
           val o4 = mult2(y,9)
       in write_value(o1 + o2 + o3 + o4);
          FIR(y,z,w,next)
       end
    end
```

Recall that the semantics of the `let` statement requires all `val`-declarations to be computed fully before the body is executed (see Section 3). Although this design contains two shared combinatorial multipliers, `mult1` and `mult2`, the outermost `let` statement ensures that the calls to the shared multipliers do not occur in parallel. As a result Parallel Conflict Analysis infers that no arbitration is required.

The shared combinatorial multipliers, `mult1` and `mult2` take a single cycle to compute their result. Generating an arbiter for a shared resource adds an extra cycle latency to each call (irrespective of whether the resource is busy at the time of call). Thus, in this case, if we naively generated arbiters for all shared resources, the performance of the multipliers would be degraded by a factor of two.

This example illustrates the importance of using static analysis to remove redundant arbiters. For this design, using Parallel Conflict Analysis to remove unnecessary arbiters leads to a 50% speed increase over a policy which simply inserts arbiters on each shared resource.

5.2 Shared-Memory Multi-Processor Architecture

Figure 4 contains SAFL code fragments describing a simple shared-memory multi-processor architecture. The system consists of two processors which have

```
type Instruction = {opcode:4,operand:12}
const WRITE=1, READ=0

extern Shared_memory(WriteSelect:1, Address:12, Data:16) : 16

extern instruction_mem1(Address:12) : 16
extern instruction_mem2(Address:12) : 16

(* Processor 1: Loads instructions from instruction_mem1 *)
fun proc1(PC:12, RX:16, RY:16, A:16) : unit =
  let val instr:Instruction = instruction_mem1(PC)
      val incremented_PC = PC + 2
   in
      case instr.opcode of
        1 => (* Load Accumulator From Register *)
             if instr.operand=1
               then proc1(incremented_PC,RX,RY,RX)
               else proc1(incremented_PC,RX,RY,RY)
      | 2 => (* Load Accumulator From Memory *)
             let val v = Shared_memory(READ, instr.operand, 0)
              in proc1(incremented_PC,RX,RY,v)
             end
      | 3 => (* Store Accumulator To Memory *)
             (Shared_memory(WRITE, instr.operand, A);
              proc1(incremented_PC,RX,RY,v))
       ... etc
  end

(* Processor 2: Loads instructions from instruction_mem2 *)
fun proc2(PC:12, RX:16, RY:16, A:16) : unit =
  let val instr:Instruction = instruction_mem2(PC)
      val incremented_PC = PC + 2
   in
      case instr.opcode of
       ....
      | 2 => (* Load Accumulator From Memory *)
             let val v = Shared_memory(READ, instr.operand, 0)
              in proc2(incremented_PC,RX,RY,v)
             end
      | 3 => (* Store Accumulator To Memory *)
             (Shared_memory(WRITE, instr.operand, A);
              proc2(incremented_PC,RX,RY,v))
       ... etc
  end

fun main() : unit = proc1(0,0,0,0) || proc2(0,0,0,0)
```

Fig. 4. Extracts from a SAFL program describing a shared-memory multi-processor architecture

separate instruction memories but share a data memory. Such architectures are common in control-dominated embedded systems where multiple heterogenous processors perform separate tasks using a common memory to synchronise on shared data structures.

The example starts by defining the type of instructions (records containing 4-bit opcodes and 12-bit operands), declaring 2 constants and specifying the signatures of various (externally defined) memory functions. Bit-widths are specified explicitly using notation `X:n` to indicate that variable `X` represents an `n`-bit value. The bit-widths of function return values are also specified in this way (`unit` indicates a width of 0).

The function `Shared_memory` takes three arguments: `WriteSelect` indicates whether a read or a write is to be performed; `Address` specifies the memory location concerned; `Data` gives the value to be written (this argument is ignored if a read operation is performed). It always returns the value of memory location `Address`.

Functions `proc1` and `proc2` define two simple 16-bit processors. Argument `PC` represents the program counter, `RX` and `RY` represent processor registers and `A` is the accumulator. The processor state is updated on recursive calls—neither processor terminates.

The `main` function initialises the system by calling `proc1` and `proc2` in parallel with `PC`, `RX`, `RY` and `A` initialised to 0.

Since the SAFL code contains parallel non-terminating calls to `proc1` and `proc2` both of which share a single resource, neither static nor relative scheduling are applicable (see Section 2): this example cannot be synthesised using conventional silicon compilers.

Soft Scheduling is expressive enough to deal with non-terminating resources: a circuit is synthesised which contains an arbiter protecting the shared memory whilst allowing `proc1` and `proc2` to operate in parallel.

5.3 Parallel Tasks Sharing Graphical Display

Consider a hardware design which can perform a number of tasks in parallel with each task having the facility to update a graphical display. Many real-life systems have this structure. For example in preparation for printing an ink-jet printer performs a number of tasks in parallel: feed paper, reset position of print head, check ink levels etc. Each one of these tasks can fail in which case an error code is printed on the graphical display.

A controller for such a printer in SAFL may have the following structure:

```
extern display (data : 16) : unit

fun reset_head()  : unit = ...
                             if head_status <> 0 then
                                  display(4) (* Error code 4 *)
                             else ...
```

```
fun feed_paper()  : unit = ... display(5) ...
fun check_ink()   : unit = ... display(6) ...

fun main()        : unit =
    (feed_paper() || reset_head() || check_ink());
     do_print();
     wait_for_next_job(); main()
```

Let us assume that each of the tasks terminates in a statically bounded time. Given this assumption, both static scheduling and Soft Scheduling can be used to ensure mutually exclusive access to `display`. It is interesting to compare and contrast the circuits resulting from the application of these different techniques.

Since the tasks may invoke a common resource, applying static scheduling techniques results in the tasks being serialised. In contrast, Soft Scheduling allows the tasks to operate in parallel and automatically generates an arbiter which dynamically schedules access to the shared `display` function.

Errors occur infrequently and hence contention for the display is rare. Under this condition, and assuming that the tasks take all roughly the same amount of time, Soft Scheduling yields a printer whose initialisation time is three times faster than an equivalent statically scheduled printer. More generally, for a system with n balanced tasks, Soft Scheduling generates designs which are n times faster.

6 Conclusions and Further Work

Soft Scheduling is a powerful technique which provides a number of advantages over current scheduling technology:

More expressive: in contrast to existing scheduling methods, Soft Scheduling can handle arbitrary networks of shared inter-dependent resources.

Increased efficiency: in some circumstances, controlling access to shared resources dynamically yields significantly better performance than statically choosing a single schedule (see Example 5.3).

Higher level of abstraction: current hardware synthesis paradigms require a designer to code arbiters explicitly at the structural level. Soft Scheduling abstracts mutual exclusion concerns completely, increasing the readability of source code without sacrificing efficiency.

One of the aims of the FLaSH Synthesis System is to facilitate the use of source-level program transformation in order to investigate a range of possible designs arising from a single specification. We have shown that *fold/unfold* transformations [3] can be applied to SAFL programs to explore various allocation/binding constraints [15]. In [16] we describe a SAFL transformation to partition a design into hardware and software parts. The simplicity of our transformation system is partly due to the resource abstraction provided by Soft Scheduling—transformations involving shared resources would be much more complex if locking and arbitration details had to be considered at the SAFL-level.

An arguable disadvantage of dynamic scheduling is that is makes the timing behaviour of the final circuit difficult to analyse. Since access to shared resources is resolved dynamically it becomes much harder to prove that real-time design constraints are met. In future work we intend to investigate (a) the incorporation of timing directives into the SAFL language; and (b) the static analysis of timing properties for dynamically scheduled hardware systems.

When the parallel interleaving of non-terminating resources is required dynamic scheduling is essential (see Example 5.2); in other cases dynamic scheduling can offer increased performance (see Example 5.3). However, for fine-grained sharing of smaller resources whose execution delays are known at compile-time (such as arithmetic units), static scheduling techniques are more appropriate. Soft Scheduling provides a powerful framework which strikes a compromise between the two approaches. The designer has the flexibility either:

– to describe a single static schedule (see Example 5.1) in which case dynamic arbitration is optimised away; or
– to leave scheduling details to the compiler (see Example 5.3) in which case dynamic arbitration is inserted where needed.

Acknowledgement

This research was supported by (UK) EPSRC grant GR/N64256: "A resource-aware functional language for hardware synthesis".

References

1. Aldrich, J., Chambers, C., Sirer, E., Eggers, S. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. Proceedings of the International Symposium on Static Analysis 1999. LNCS Vol. 1694, Springer Verlag.
2. Berkel, K. van. Handshake Circuits: an Asynchronous Architecture for VLSI Programming. International Series on Parallel Computation, Vol. 5. Published by Cambridge University Press, 1993.
3. Burstall, R.M. and Darlington, J. A Transformation System for Developing Recursive Programs, JACM 24(1).
4. Cartwright, R. and Fagan, M. Soft Typing. Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation
5. De Micheli, G., Ku, D., Mailhot, F., Truong, T. The Olympus Synthesis System for Digital Design. IEEE Design & Test Magazine, October 1990.
6. De Micheli, G. Synthesis and Optimization of Digital Circuits. Published by McGraw-Hill Inc., 1994.
7. Edwards, D., Bardsley, A. Balsa 3.0 User Manual. Available from `http://www.cs.man.ac.uk/amulet/projects/balsa/`
8. Hennessy, J., Patterson, D. Computer Architecture A Quantitative Approach. Published by Morgan Kaufmann Publishers, Inc. (1990) ; ISBN 1-55860-069-8

9. IEEE. Verilog HDL Language Reference Manual. IEEE Draft Standard 1364, October 1995.
10. Ku, D., De Micheli, G. Relative Scheduling Under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits. IEEE Transactions on CAD/ICAS, June 1992.
11. Ku, D., De Micheli, G. HardwareC—a language for hardware design (version 2.0). Stanford University Technical Report No. CSL-TR-90-419.
12. Ku, D., De Micheli, G. Constrained Resource Sharing and Conflict Resolution in Hebe. Integration – The VLSI Journal, December 1991.
13. Milner, R., Tofte, M., Harper, R. and MacQueen, D. The Definition of Standard ML (Revised). MIT Press, 1997.
14. 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.
15. 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.
16. Mycroft, A. and Sharp, R. Hardware/Software Co-Design Using Functional Languages. Proc. of Tools and Algorithms for the Construction and Analysis of Systems 2001. LNCS Vol. 2031, Springer-Verlag.
17. 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.
18. Sharp, R. and Mycroft, A. The FLaSH Compiler: Efficient Circuits from Functional Specifications. AT&T Technical Report tr.2000.3.
Available from `http://www.uk.research.att.com`
19. Sharp, R. and Mycroft, A. A Higher Level Language For Hardware Synthesis. To appear: Proc. of Correct Hardware Design and Verification Methods (CHARME), 2001.
20. Tenison Tech EDA. CtoV Reference Manual.
Available from `http://www.tenisontech.com`