Recursion, Iteration and Hume Scheduling

Greg Michaelson, Robert Pointon, Gudmund Grov and Andrew Ireland

School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton, Scotland, EH14 4AS
{greg,rpointon,gg10,air}@macs.hw.ac.uk

Abstract

Converting programs from full or PR-Hume to FSM- or HW-Hume involves transforming expression recursion to box iteration. However, this can add considerable overheads through unnecessary scheduling of other boxes dependent on the iteration output. Here we explore how analysis of output behaviour can identify boxes which may be executed independently of normal super-step scheduling, without affecting program meaning, and minimising the impact of the recursion to iteration transformation. We use TLA+ to formalise the recursion to iteration transformation, and alternative scheduling strategies, enabling their verification with the TLC model checker.

1 INTRODUCTION

Hume[MK02] is a novel language that encapsulates a simple functional language within a finite state machine framework. A program consists of a collection of automata, termed *boxes*, which are *wire*d together, while the functional language is used to handle the availability and pattern matching at box input, and the production of box output.

A central motivation for Hume's development was to produce a language amenable to resource use analysis[HM03, MHJ04]. It is well known that precise static behavioural analyses cannot be developed for Turing complete languages, because of the undecidability of termination and equivalence. It is also well known that trying to program in languages whose expressiveness are restricted to enable precise analysis is unpalatable. Thus, we have conceived of Hume as a *multi-level* language where different levels have different decidability properties depending on the types and control constructs that are deployed, as shown in Table 1.

We propose the following methodology for program development:

write program in full Hume
until program costed or deemed uncostable
  apply static analyses
  if analyses break then
    transform offending construct to lower level

Level	Types/Constructs	Time/Space Analyses
HW-Hume	tuples of bits	precise costs
FSM-Hume	bounded types + conditions	accurate bounds
PR-Hume	unbounded types + primitive recursion	weak bounds
Full Hume	general recursion	undecidable

TABLE 1. Hume levels and cost properties.

2 FROM LINEAR RECURSION TO BOX ITERATION

Program transformation is central to our programming methodology. The greatest novelty lies in our ability to exploit transformations that move processing from weak costability at the expression level to stronger costability at the coordination level. Here, the key transformation is from expression recursion to box iteration, moving from full, PR- or HO- Hume to FSM-Hume.

In the well known transformation[Man74], linear recursion may be replaced by iteration:

```
linrec f g h x =
    if f x
    then return g x
    else return linrec f g h ( h x)

⇔

iter f g h x =
    while not ( f x)
        x := h x
    return g x
```

This transformation may be extended to primitive recursion, through the introduction of accumulation variables, and to higher order functions, through unfolding of HOF applications to form equivalent specialised functions, but these are not considered here.

In Hume, the iteration may be expressed by a box with a looped wire:

```
box iterbox
  in  ( i::t1, iter::t3) out ( o::t2, iter'::t3)
match
  ( i, *) → ( *, i)
| ( *, iter) → if f iter
                then ( h iter, *)
                else ( *, g iter);

wire iterbox (... , iterbox.iter')
```

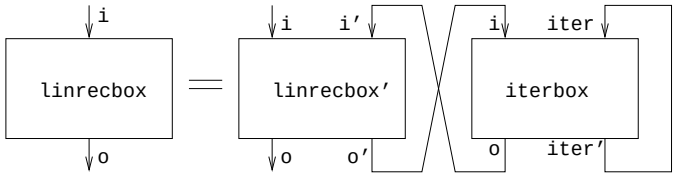

FIGURE 1. Recursion to box iteration.

```
                (... , iterbox.iter);
```

To be a valid Hume program the original `linrec` function must have been in a box. Therefore a box that directly calls a linear recursive function may be transformed to an equivalent two box form: the original box calls a new iterative box with a a looped wire to manage the state:

```
box linrecbox
  in  ( i::t1) out ( o::t2)
match
  ( x) → (...linrec f g h x...);

⇔

box linrecbox'
  in  ( i::t1, i'::t2) out ( o::t2, o'::t1)
match
  ( i, *) → ( *, i)
| ( *, iter) → (... iter..., *);

wire linrecbox' (... , iterbox.o)
                (... , iterbox.i);
wire iterbox ( linrecbox'.o', iterbox.iter')
                ( linrecbox'.i',iterbox.iter);
```

as shown schematically in Figure 1.

For example, considering multiplying two integers by repeated addition:

```
mult r x 0 = r;
mult r x y = mult (r+x) x (y-1);
```

embedded in a program to generate successive squares:

```
type integer = int 64;

box mult1
  in  ( i::integer) out ( i'::integer, o::string)
match
   x → ( x+1, (mult 0 x x) as string++"/n");

stream output to "std_out";
```

```
wire  mult1  ( mult1.i' initially 0) ( mult1.i,output);
```

Rewriting the function in curried form with an explicit condition:

```
mult (r, x, y)  = if  y==0 then  r else  mult (r+x, x, y-1)
```

we can identify:

```
f (r, x, y) = y==0
g (r, x, y) = (r+x, x, y-1)
h (r, x, y) = r
```

giving the transformed program:

```
box  mult2
  in   ( i::integer, iter'::integer)
  out  ( i'::integer, iter::(integer,integer,integer), o::string)
match
  ( x, 0)  →  ( x+1, (0, x, x), *)
| ( x, r)  →  ( x+1, (0, x, x), r  as  string++"/n");

wire  mult2  ( mult2.i' initially 0,  itermult.o initially 0)
             ( mult2.i, itermult.i, output);

box itermult
  in   ( i::(integer,integer,integer), iter::(integer,integer,integer))
  out  ( o::integer,iter'::(integer,integer,integer))
match
  (( r, x, y), *)  →  (*, ( r, x, y))
| (*, ( r, x, y))  →  if  y==0
                       then ( r, *)
                       else (*, ( r+x, x, y-1));

wire  itermult  ( mult2.iter, itermult.iter')
                ( mult2.iter', itermult.iter);
```

As well as making static analysis more tractable, this transformation also significantly reduces space requirements. For some recursive functions, it may be necessary to retain intermediate stack and heap space until the recursion terminates. In contrast, in the iterative form, space can be reclaimed on each iteration, with only the accumulating result retained on the feedback wire between iterations.

Table 2 shows the times to find the squares of the first 1411 integers using the Hume reference interpreter.

Essentially, around one million calls to the recursive function are replaced by around one million iterative box invocations. There is a net saving of around 30 seconds or 30% for the iterative boxes compared with the recursion.

Version	Time (secs)
recursive (mult1)	100.4
iterative (mult2)	70.9

TABLE 2. Multiplication recursion and iteration times.

3 FORMALISING HUME WITH TLA

TLA[Lam94] is a linear temporal logic created based on the fact that specifications are developed at different grain of atomicity, and allows us to verify that a finer grained specification implements a coarser grained. This requires what Lamport calls *stuttering steps*, i.e. steps that leaves the state unchanged. All TLA formulae are therefore *invariant under stuttering*, i.e. adding or removing stuttering steps does not change the validity of a formula. In a Hume transformation a box is transformed to a 'component' of several boxes which requires more super-steps for the computation. Allowing stuttering steps of the box before transformation is the key when verifying transformations.

In TLA both program and properties are specified in the same logic. This simplifies reasoning about properties compared to for instance Hoare logic[Hoa69]. TLA uses normal logical connectives like $\wedge, \vee$ and $\Rightarrow$, in addition to the three operators $'$,$\Box$ and $\exists$: $'$ is used in actions – if we do not prime a variable we refer to the before state and by priming it we refer to the after state; $\Box$ is the normal 'always' operator in temporal logic, i.e. $\Box F$ denotes that F is always *True*; $\exists$ is an existential quantifier with non- standard semantics used for hiding internal variables in a specification. We also require $\exists$ when verifying transformations in Hume.

Liveness constraints and (partly) the environment are not relevant here, and will therefore be ignored in the rest of the paper. The environment is partly relevant since when we verify a transformation of a component we ignore boxes not part of it. We must then create an environment which non-deterministically updates input and output wires to the component. We describe a system consisting of $n \geq 1$ boxes. The state space consists of the wires w, a hidden component s used to ensure correct scheduling. To simplify reasoning about boxes we have also introduced state b to them. $[\mathcal{N}]_x$ denotes either running action $\mathcal{N}$ or leaving the state variables x unchanged ($x' = x$). This is to ensure invariance under stuttering. Further, $\langle \cdots \rangle$ denotes a sequence. We can then specify a Hume program P as:

$$P \triangleq \exists\, s : Init_s \wedge Init_w \wedge \bigwedge_{i=1}^{n} Init_i \wedge \Box[\mathcal{N}_s \wedge \mathcal{N}_E \wedge \bigwedge_{i=1}^{n} \mathcal{N}_i]_{\langle w,b,s \rangle} \tag{1}$$

$\mathcal{N}_s$ updates s in the following order: *Execute* $\mapsto$ *Consume* $\mapsto$ *Write* $\mapsto$ *Execute* $\mapsto$ .... Except for when we are transforming a component an abstraction away from the rest of the system $\mathcal{N}_E$ is trivially *True*. $\bigwedge_{i=1}^{n} Init_i$ initialises the box components and $\bigwedge_{i=1}^{n} \mathcal{N}_i$ runs all the boxes. Since our overall focus is the coordination layer we will not go into more detail than an informal discussion of the box actions

($\mathcal{N}_i$): In a *Execute* step all *Runnable* boxes are executed; In a *Consume* step all input wires are consumed and in a *Write* step all output wires are written to. This steps provides both a logical and philosophically satisfying ownership of shared variables, i.e. wires, when communicating: In an *Execute* step the overall system owns the wires; in a *Consume* step the wires are owned by the destination box while in a *Write* step they are owned by the source box.

TLA$^+$[Lam02] is a specification language which combines TLA with a variant of ZF logic and offers tool support in form of the TLC model checker. Although ZF does not provide an adequate representation for the underlying data structures of the expression layer of Hume it provides sufficient functionality for this case where the focus is more on the coordination layer. TLC is unique in the way that we can verify refinement by model checking or, in our case, transformation as shown in the next section.

4 VERIFYING RECURSION AS ITERATION

Let P be the original program and P^T be the transformed program. We require that P^T behaves the same way as P in the following manner: The same inputs are consumed; The same output are produced when consuming the same inputs; They behave the same way after executing, e.g. if P blocks and cannot run, so shall P^T. However, we ignore the extra steps in P^T. To verify the transformation we threat it as a refinement, which in TLA is simply implication. P executes,consumes and writes in the same super-step – while in P^T these operations are performed in different step: e.g. The internal state b of a box holds among other things a consume buffer c and a output buffer o – both updated in an *Execute* step. These are updated in the *Execute* step of P. In P^T c is in the input buffer of `first` and o in the output buffer of `last` – hence c is updated in the first super-step and o in the last super-step. In addition the environment might update the input and output wires before the component of P^T terminates. We overcome this problem by not comparing state element by state element but creating a mapping from the state space of P^T called a *refinement mapping*[AL88]. We then show that the implication holds under this mapping. This mapping is achieved with the $\exists$ operator, leaving us to verify:

$$\exists\, s : P^T \Rightarrow \exists\, s, w, b : P \tag{2}$$

To show (2) must create witnesses $\overline{w}$ and $\overline{b}$ for w and b which represents the refinement mapping. To achieve this we must introduce history variables to P^T, which has to follow some well-defined introduction rules in TLA to ensure soundness. We use them to buffer changes up to the step where the component of P^T terminates. Then we use the buffered values to update the state variables. This is the only step between consuming input and writing outputs where any change is made.

The proof was achieved by the TLC model checker. Due to the amount of computation required for computing each step we only model checked a small domain (1..50). The next step will be to verify that we can replace `mult1` by

the component in P^T. This requires a proof that additional super-step between consuming inputs and writing outputs does not change how the other boxes of the program behaves. By using other scheduling algorithms the proof we have shown is sufficient, and we do not need to verify the complete program.

5 HUME SCHEDULING

While the two Hume versions are equivalent in terms of the result they produce, they may differ in their temporal behaviour, depending on the chosen scheduling strategy in the implementation.

A Hume wire is a single value buffer that connects exactly two boxes. A Hume box can only run once per schedule super-step, it will try to match its input, and upon success consume from the input wires, evaluate some result, and then block until the output wires are free to accept the result. These semantics lead to deterministic programs, where non-deterministic programs can be constructed by specifying that a box is 'fair', and this enables fair (LRU) matching of the box input patterns.

At the end of each super-step cycle, a box may be in one of the following states:

1. **Runnable** The box has successfully consumed inputs and asserted outputs.
2. **Blocked-output** The box has successfully consumed inputs but failed to assert outputs. It will attempt to assert outputs on subsequent cycles.
3. **Match-fail** The box has failed to find required inputs.

The Match-fail state is equivalent to Runnable, but is distinguished to support program tracing and debugging. In what follows, we understand Runnable to include Match-fail.

The execution cycle is illustrated in Figure 2.

One scheduling implementation of this is lock-step scheduling where all non-blocked boxes are run once and then the inputs and updated with the new outputs.[1] Thus, the normal lock-step scheduling of Hume programs is as follows:

for ever
  execute each Runnable box
  super-step

To return to the recursion to iteration transformation, where `linrecbox` takes one schedule step to complete, `iterbox` will take many, depending on the original depth of recursion. In principle this is not problematic as repeated box iteration should have the same order of cost as recursion. However, because the base scheduling mechanism always tries to run every box on each execution cycle, any boxes that depend directly or indirectly on the box iteration generating an output

[1] Due to the current lack of any data-flow, or temporal dependency analysis, lock-step scheduling is used in all valid Hume implementations.

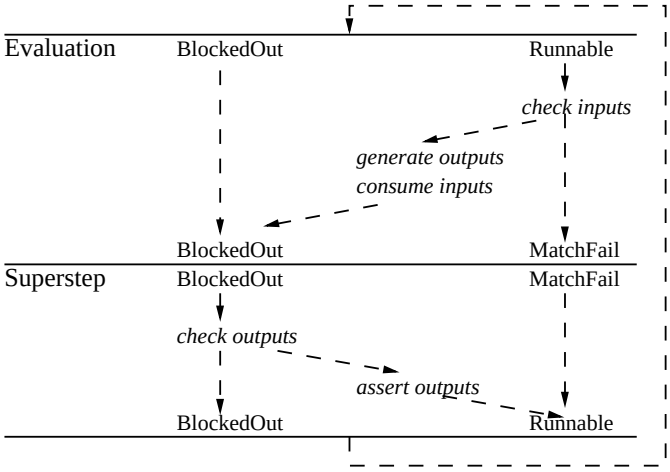

FIGURE 2. Execution cycle.

Box	Runnable	Matchfail
mult1	1412	0
mult2	1412	998589
itermult	998589	1412

TABLE 3. Scheduling states.

will be repeatedly scheduled but fail to consume inputs. Such boxes nonetheless incur a repeated unnecessary scheduling overhead.

Table 3 shows the overall scheduling behaviour of the recursive and iterative multiplication programs. Note that in this example boxes never enter the blocked output state.

In the recursive case, the single box `mult1` calls `fac` 1411 times. In the iterative case, the top level box `mult2` passes initial data to `itermult` 1411 times, with itermult failing to match inputs once prior to each communication. Thereafter, both `mult2` and `itermult` are scheduled 998,489 times, with `mult2` always failing to match inputs.

As well as being introduced by this transformation, the use of feedback wiring and iteration is a standard Hume programming construct. Here, it is common for many schedule steps to depend on the looped wire and not on any external input or output. Thus, while these conditions hold, such boxes can be scheduled repeatedly for greater efficiency. Note that changing the scheduling locally *may* change the temporal behaviour of the program and affect program meaning.

6 EFFICIENT SCHEDULING

There are at least two distinct approaches to improving the scheduling efficiency: one is to take a hierarchical view of the expression decomposition, and the other is to amend the scheduler for improved efficiency in particular cases.

6.1 Hierarchical Scheduling

By treating the decomposition of the expressions of a box into a set of boxes that are then 'nested' within the skeleton of the original box, then it is possible to use a hierarchical scheduling mechanism. The child boxes can read the input from skeleton (once input is matched), and then run using a nested instance of the Hume scheduler until they produce output for the skeleton. The scheduler within the skeleton box can only be active while the skeleton box itself is being executed.

```
for ever
  execute each Runnable box
  super-step

execute(box):
  if matched input then
    evaluate expression
    do output

execute(skeleton box):
  if matched input then
    while no output
      execute each Runnable child box
      super-step children
```

This approach preserves the super-step meaning within the original program and therefore program meaning.

We believe there is a way of transforming any nested box hierarchy into the usual flat representation. However, any resultant flat representation would have many boxes that would be idle the majority of the the time, and so efficiency would likely be lost.

Nesting of the scheduler is an overkill for what we are trying to accomplish and requires Hume programs to be decomposed with the special skeleton boxes. At present, we are trying to keep Hume stable and are resistant to feature bloat; therefore we would prefer not to add the machinery to support the skeleton boxes and nested scheduling.

6.2 Towards Staged Scheduling

By keeping the original scheduling approach but amending it to automatically schedule more efficiently in certain cases, we should gain benefits for free in many existing programs regardless of whether they contain boxes generated by expression decomposition or not.

In the usual execution of a box, a Runnable box may have either asserted outputs to other boxes and itself, or just to other boxes, or just to itself. If it has asserted outputs just to itself then it can have no impact on the ability of any other box to consume inputs. We say that such boxes have the *self-output* property.

Thus, in principle, such boxes may execute repeatedly until they assert an output for another box, without affecting the overall outcome of program execution, provided there are no strong timing dependencies elsewhere in the program.

Let us add a fourth execution cycle state of **Selfout**. Then a staged scheduling strategy might be:

```
for ever
  while no box is Runnable
    execute each Selfout box
  execute each Runnable box
  super-step
```

Where the original may suffer from starvation and have poor performance, the hierarchical approach would be expected to offer better performance but with more likelihood of starvation, where the staged approach should avoid starvation while still offering better performance than the original.

7 STATIC AND DYNAMIC SELF-OUTPUT IDENTIFICATION

It would be highly beneficial to identify statically box matches with the self-output property at compile time, to enable optimal scheduling in their presence. For HW-Hume this is straightforward as there are no conditional expressions on the right hand side of box matches. Thus, self-output matches can be identified by direct inspection of associated actions and wiring.

However, FSM-Hume introduces conditional expressions in box match actions and so analysis can at best identify matches which may self-output. If such matches are treated liberally as self-output then their boxes may be prioritised inappropriately. In contrast, if such matches are treated conservatively as non-self-output then opportunities for scheduling efficiencies will be lost. This is compounded for PR- and full Hume which allow mutually recursive functions, additionally requiring full dataflow analysis.

Thus, we have modified the reference interpreter to implement the third modified scheduling approach, as shown in Figure 3, with dynamic identification of self-output boxes. In this implementation, on the super-step we actively inspect

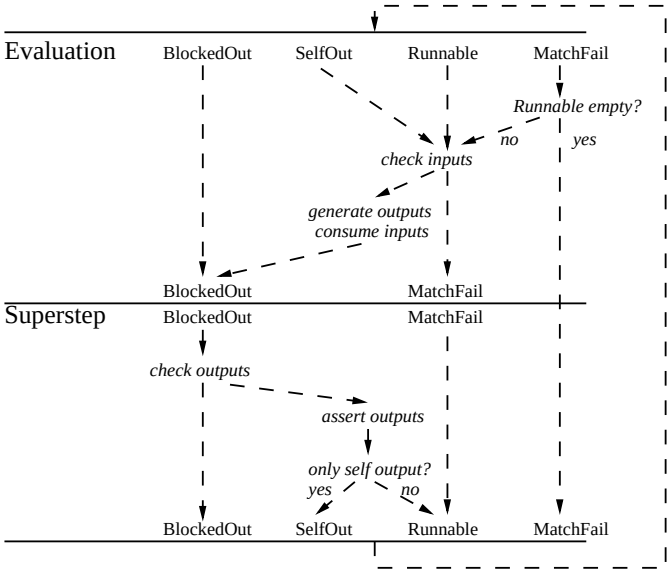

FIGURE 3. Execution cycle with self-output prioritisation.

Version	Scheduling	Time (secs)
recursive	original	100.4
iterative	original	70.9
iterative	new	57.0

TABLE 4. Execution times with self-out scheduling.

both the output values and the wiring for each box and deem a box self-output if it has only produced values on feedback wires.

Executing the iterative multiplication example on the original and modified interpreters gives a further 14% time saving for this example, as shown in Table 4. This saving results from a significant reduction in unnecessary scheduling, as shown in Table 5. By comparison with the old scheduling state behaviour shown in Table 3, with the new scheduling scheme, `mult2` is never scheduled once it has entered a failed match state, so long as `itermult` is looping in the self output state, giving a saving of around 997,000 unnecessary schedules out of around 2,000,000.

Box	Scheduling	Runnable	Matchfail	Selfout
mult2	new	1412	1412	0
itermult	new	1411	1411	97178

TABLE 5. New scheduling states.

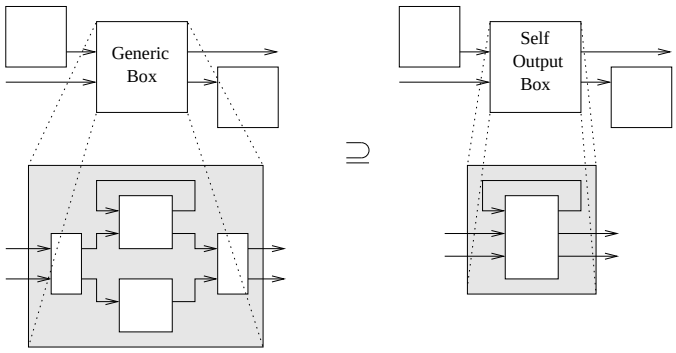

FIGURE 4. Decomposition of Hume boxes.

8 CORRECTNESS

The Hume super-step scheduling results in program output being deterministic and independent of the order of box scheduling. The transformation of functions into boxes and modified scheduling is now shown informally as preserving the program super-step semantics.

The evaluation of Hume expressions (within a box) is performed in a strict and deterministic manner, that is, there is a single thread of execution for evaluation. By observing that the single threaded evaluation of an expression means that evaluation can occur at only one locus in an expression at any single point in time, we claim that, in an equivalent set of generated boxes, one and only one box can ever be runnable at any point in time. Therefore, in transforming functional code into a set of boxes, the boxes must under-utilise the potential for concurrency.

With only one box runnable at any time, the scheduling of these boxes becomes trivial - evaluate the one box then super-step the outputs of this same box. Thus the super-step semantics are preserved but irrelevant and can be made substantially more efficient.

The original set of boxes would typically utilise concurrency and hence the super-step semantics are vital. The two main approaches suggested in this paper - hierarchical and self-output - preserve the program super-step semantics by forcing all of the transformed generated boxes to run to completion before enabling the original boxes again.

The hierarchical approach (as shown in Figure 4) treats the generated boxes as a Hume program *embedded within a Hume program*. While it preserves the necessary semantics it also enables a more general and powerful programming model, i.e. it does not restrict the nested boxes to being single threaded and it also allows hidden state. The self-output approach (as shown in Figure 4) optimises the scheduling for a particular class of generated boxes, and thus is slightly restrictive. A more general solution would mark the generated boxes at compile time and use this information along with the dynamic scheduling.

While we stated that only one box could ever be runnable at any time, it may in practise be useful to allow more than one to be runnable, but in a parallel pipelined approach. However the generated boxes are still not, and cannot be, concurrent.

9 FORMALISING SELF-OUTPUT SCHEDULING IN TLA

In the specification of a self-output scheduling algorithm (3) we replace the next action of a box by $\mathcal{N}_i$ from (1) by ${\mathcal{N}_i}^{SO}$:

$$P_3 \triangleq \exists\, s : Init_s \wedge Init_w \wedge \bigwedge_{i=1}^{n} Init_i \wedge \Box[\mathcal{N}_s \wedge \bigwedge_{i=1}^{n} {\mathcal{N}_i}^{SO}]_{\langle w,b,s\rangle} \tag{3}$$

Selfout behaves the same way as *Runnable* (*Selfout* $\Rightarrow$ *Runnable*, but they are still two distinct states. *Selfout* is only introduced to identify the cases described above – hence the scheduling in (3) will only differ from (1) when none of the boxes are *Runnable*. Let *SO* hold if there are now *Runnable* boxes:

$$SO \triangleq \bigwedge_{i=1}^{n} st_i \neq Runnable$$

By using this definition we can define the next action ${\mathcal{N}}^{SO}$: Let ${\mathcal{N}_i}^{unch}$ be the 'unchanged action' used above. Further, ${\mathcal{N}_i}^{s}$ behaves like $\mathcal{N}_i$ with the difference that if the process is in the state *Selfout*, introduced below, it behaves like it is *Runnable*. If *SO* then we only execute boxes where $st_i = Selfout$, unless we are in a *Write* (*W*) step. We then need to reassert the output wires to see if they have become *Runnable*. If $\neg SO$ all boxes are executed as in (1):

$${\mathcal{N}_i}^{SO} \triangleq \textbf{ if } SO \Rightarrow (st_i = Selfout \vee W) \textbf{ then } {\mathcal{N}_i}^{s} \textbf{ else } {\mathcal{N}_i}^{unch}$$

We let so_i hold for box *i* if the output buffer *o* only (attempts) to write to internal wires. A box enters a *Selfout* state in an *Execute* (*E*) step – and remains there unless it blocks or *so* fails after an *E* step.

We next prove that self-output scheduling creates the same program as super-step scheduling, i.e. it implements super-step scheduling. The proof is based around the fact that a *Selfout* state is the same as a *Runnable* state in super-step scheduling – shown in Lemma 1. A consequence of this is that ${\mathcal{N}_i}^{s}$ implies $\mathcal{N}_i$ and is proved in Lemma 2. Let the super-script ^{so} denote self-output scheduling if it is not clear by the context:

Lemma 1 $Selfout^{so} \Rightarrow Runnable$

Proof. By the definition of the *Selfout* possible state. □

Lemma 2 ${\mathcal{N}_i}^{s} \Rightarrow \mathcal{N}_i$

Proof. By the definition of the *Selfout* possible state and Lemma 1. □

To verify the implementation a mapping $\overline{st}$ is created for the internal box states of the self-output algorithm. $\overline{st}$ provides the witness for existentially quantified state *st* variable, and equals *st* unless $st = Selfout$ where it equals *Runnable*. The implementation is proved in Theorem 1:

Theorem 1

$$\exists\, s : Init_s^{so} \wedge Init_w^{so} \wedge \bigwedge_{i=1}^{n} Init_i^{so} \wedge \Box[\mathcal{N}_s^{so} \wedge \bigwedge_{i=1}^{n} \mathcal{N}_i^{SO}]_{\langle w,b,s\rangle} \Rightarrow$$

$$\exists\, s, st : Init_s \wedge Init_w \wedge \bigwedge_{i=1}^{n} Init_i \wedge \Box[\mathcal{N}_s \wedge \bigwedge_{i=1}^{n} \mathcal{N}_i]_{\langle w,b,s\rangle}$$

Proof. Since we are ignoring liveness here, the proof is split into two parts: (1) We must show that the initial state are identical; and (2) we must show that the *step-simulations*(next actions) are identical. The initial states are identical by definition, hence

$$\exists\, s : Init_s^{so} \wedge Init_w^{so} \wedge \bigwedge_{i=1}^{n} Init_i^{so} \Rightarrow \exists\, s, st : Init_s \wedge Init_w \wedge \bigwedge_{i=1}^{n} Init_i$$

is easy to prove. To show step-simulation we must show that

$$\exists\, s : \Box[\mathcal{N}_s^{so} \wedge \bigwedge_{i=1}^{n} \mathcal{N}_i^{SO}]_{\langle w,b\rangle} \Rightarrow \exists\, s, st : \Box[\mathcal{N}_s \wedge \bigwedge_{i=1}^{n} \mathcal{N}_i]_{\langle w,b\rangle}$$

which by standard TLA reasoning can be reduced to

$$\exists\, s : [\mathcal{N}_s^{so} \wedge \bigwedge_{i=1}^{n} \mathcal{N}_i^{SO}]_{\langle w,b\rangle} \Rightarrow \exists\, s, st : [\mathcal{N}_s \wedge \bigwedge_{i=1}^{n} \mathcal{N}_i]_{\langle w,b\rangle}.$$

We know that $[\cdots]_x$ abbreviates $\cdots \vee x' = x$, reducing our problem to

$$\exists\, s : (\mathcal{N}_s^{so} \wedge \bigwedge_{i=1}^{n} \mathcal{N}_i^{SO}) \vee (\langle w,b\rangle' = \langle w,b\rangle) \Rightarrow \exists\, s, st : (\mathcal{N}_s \wedge \bigwedge_{i=1}^{n} \mathcal{N}_i) \vee (\langle w,b\rangle' = \langle w,b\rangle).$$

Since $(\langle w,b\rangle' = \langle w,b\rangle) \equiv (\langle w,b\rangle' = \langle w,b\rangle)$ is trivial to show, we are left to prove

$$\exists\, s : (\mathcal{N}_s^{so} \wedge \bigwedge_{i=1}^{n} \mathcal{N}_i^{SO}) \Rightarrow \exists\, s, st : (\mathcal{N}_s \wedge \bigwedge_{i=1}^{n} \mathcal{N}_i).$$

^{so} is just added for readability and $\mathcal{N}_s^{so} \equiv \mathcal{N}_s$, meaning the conjuncts $\mathcal{N}_s^{so}$ and $\mathcal{N}_s$ can be removed:

$$\exists\, s : \bigwedge_{i=1}^{n} \mathcal{N}_i^{SO} \Rightarrow \exists\, s, st : \bigwedge_{i=1}^{n} \mathcal{N}_i.$$

Since both programs have the same amount of boxes, we are left to show that

$$\exists\, s : \mathcal{N}_i{}^{SO} \Rightarrow \exists\, s, st : \mathcal{N}_i.$$

holds for an arbitrary box i. By unfolding the definition of $\mathcal{N}_i{}^{SO}$ we must show that

$$\exists\, s : \textbf{if } SO \Rightarrow (st_i = Selfout \vee W) \textbf{ then } \mathcal{N}_i{}^{s} \textbf{ else } \mathcal{N}_i{}^{unch} \Rightarrow \exists\, s, st : \mathcal{N}_i.$$

We separate between the two cases (1) $(SO \Rightarrow (st_i = Selfout \vee W))$ and (2) $\neg(SO \Rightarrow (st_i = Selfout \vee W))$. Case (1) is true if either $\neg SO$ or $st_i = Selfout$ or W. For these cases the following must hold

$$(\neg SO \vee st_i = Selfout \vee W) \Rightarrow \exists\, s : \mathcal{N}_i{}^{s} \Rightarrow \exists\, s, st : \mathcal{N}_i.$$

This hold by Lemma 2. For case (2) $\neg(SO \Rightarrow (st_i = Selfout \vee W))$ can be reduced to $SO \wedge st_i \neq Selfout \wedge \neg W$. Hence we must verify

$$SO \wedge st_i \neq Selfout \wedge \neg W \Rightarrow \exists\, s : \mathcal{N}_i{}^{unch} \Rightarrow \exists\, s, st : \mathcal{N}_i.$$

Since SO we know that none of the boxes are *Runnable* and the current box is not *Selfout*, we know that the state is either *Matchfail* or *Blocked*. By $\neg W$ we now we are in an *Execute* step or a *Consume* step. Since in a *Write* step a *Matchfail* state is turned into *Runnable* it must be in a *Consume* step if the *st* is *Matchfail*, where only the input wires are updated. Since a *Matchfail* is induced they will be left unchanged. Hence the theorem holds. Furthermore, a box will leave all it's variables unchanged if it is in a *Blocked* state except when it is in a *Write* (W) step – where it attempts to write to the output wires. Since one of the assumptions are $\neg W$ we know that all variables are left unchanged. This concludes the proof. □

In Theorem 1 we verified that $P_3 \Rightarrow P_1$. We can also show equivalence between the scheduling ($P_3 \equiv P_1$) by showing the converse of this implications ($P_1 \Rightarrow P_3$). This proof is more complicated and requires the introduction of auxiliary variables, and we will not show it here.

10 FORMALISING HIERARCHICAL SCHEDULING

Hierarchical scheduling introduces new features to Hume (box skeletons) and we can only show that it implements scheduling if there are no box skeletons. This proof will not be shown here, where we will focus more on the application of this scheduling.

In §4 a proof of transformations was shown. The proof was divided into two parts: A proof that a component implements a box; A proof that the extra super-step introduced does not change the behaviour of the rest of the program. We only showed the first part of this, but are confident that hierarchical scheduling with the transformation

$$Box \Rightarrow Box\ Structure$$

simplifies each of these proofs.

In the component transformation proofs the only free variable is the output buffer – all others are existentially quantified/hidden by the ∃ operator. This is due the latency introduced by the transformation and that we cannot control the environment. The mappings also complicates the soundness proof of this approach. If we replace this component by a box structure we do not have hide any component, which very much simplifies things.

For program transformation, by introducing a box structure instead of the component we know that all other boxes will "freeze" until termination of the structure. Hence, there is no need to verify the impact the transformation has on other boxes.

11 SUMMARY

We have introduced a transformation from linear recursion to box iteration, which increases the applicability of static resource analysis and improves performance by reducing memory overheads. The transformation has been verified in TLA, which has also been used to formalise Hume's super-step scheduling.

The recursion to box iteration transformation highlights scheduling inefficiencies in the presence of self-output boxes which repeatedly process their own outputs without affecting other boxes, resulting in unnecessary scheduling of other boxes. We have presented a modified scheduling approach that substantially improves scheduling of self-ouput boxes. We have also discussed the benefits of box abstraction for generalising this improvement to enable independent Hume program components to execute in parallel. Finally, we have formalised the new scheduling algorithm in TLA and shown it to be equivalent to super-step scheduling.

We next intend to automate the transformation and apply it to realistic programs. We will also explore the impact of the modified scheduling strategy on a wider range of programs. We think that it will prove fruitful to investigate related dataflow analysis techniques. Finally, we think it important to initiate debate on how best to abstract over Hume boxes.

12 ACKNOWLEDGEMENTS

This research is supported by the EU FP6 EmBounded Project.

We would like to thank our collaborators in the EmBounded and UK DTC SEAS projects, in particular Kevin Hammond.

REFERENCES

[AL88] Martn Abadi and Leslie Lamport. The Existence of Refinement Mappings. In Yuri Gurevich, editor, *Proceedings of the Third Annual IEEE Symp. on Logic in Computer Science, LICS 1988,* pages 165–175. IEEE Computer Society Press, July 1988.

[HM03] K. Hammond and G. Michaelson. Hume: A Domain Specific Language for Real-Time Embedded Systems. In *Proceedings of GPCE'03: Generative Programming and Component Engineering, Erfurt, Germany*. Springer, LNCS, September 2003.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–585, October 1969.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[Lam02] Leslie Lamport. *Specifying Systems — The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Reading, Massachusetts, 2002.

[Man74] Z. Manna. *Mathematical Theory of Computing*. McGraw-Hill, 1974.

[MHJ04] G. Michaelson, K. Hammond, and J.Serot. FSM-Hume: Programming Resource-Limited Systems using Bounded Automata. In *Proceedings of ACM Symposium on Applied Computing, Nicosia, Cyprus,* pages 1455–1461. ACM Press, March 2004.

[MK02] G. Michaelson and K.Hammond. The Hume Language Definition and Report, Version 0.2. Technical report, Heriot-Watt University and University of St Andrews, January 2002.