Multi-core parallelisation of Hume through structured transformation

Greg Michaelson, Abyd Al Zain, and Gudmund Grov

Heriot-Watt University,Riccarton,Scotland,EH14 4AS,UK
{G.Michaelson,A.D.Alzain,G.Grov}@hw.ac.uk

Abstract. The use of the Hume box calculus to systematically transform a single box into an equivalent multi-box program offering balanced parallel implementation is discussed. The approach is illustrated through the development of a multicore matrix multiplication program.

Keywords: multicore, transformation, box calculus, Hume.

1 Introduction

Hume[20] is a contemporary language in the functional tradition, based on abstractions over concurrent finite-state automata, and oriented to domains requiring strong static guarantees that resource bounds are satisfied. Hume is a *multi-level* language, with different combinations of types and control expressions enabling a principled tradeoff between expressive power and property decidability. Thus, at one extreme full Hume is turing complete, highly expressive and of undecidable resource bounds; at the other, HW-Hume[1] has minimal expressivity but decidable bounds.

Hume is based on strong formal foundations, with stable static and dynamic semantics informing implementations and analyses. At the heart of Hume's conception is the Hume Abstract Machine (HAM), providing a unifying basis for a common tool set, in particular for closely coupled native machine code generation and resource analyses. This greatly facilitates the accurate instantiation of generic analyses for specific hardware platforms.

Hume programs are composed of *boxes* linked by *wires*, where box transitions are structured by pattern matching over input wires to identify expressions to generate values for output wires. A Hume box, like a finite state automata, is non-terminating, repeatedly cycling to match inputs and generate outputs. It is important to note that boxes have transient local state which is lost at the end of each execution cycle, with persistent state residing solely on wires.

The Hume semantics specifies that program execution is in two stages. First of all, each box attempts to match inputs and, if successful, generates outputs. At the end of this stage, no inputs are consumed or outputs asserted. In the second *super-step* stage, matched inputs are consumed and generated outputs

[1] Hardware Hume

are asserted, with boxes blocking until the next cycle if any previous outputs remain unconsumed.

Note that, in the first stage, box execution has no impact on wires, so boxes cannot interact and their execution may be in arbitrary order. Furthermore, a common Hume form is that of an iterating box which repeatedly consumes only outputs it generates itself. Such *self-output* boxes may in principle execute both stages continuously and independently of other boxes. Thus, the Hume execution model offers excellent opportunities for concurrent box execution.

Recently, we have developed a prototype shared memory, parallel implementation of the standard HAM interpreter[1] which uses OpenMP to associate each boxes with an individual thread for first stage execution on multiple cores. Initial experiments suggest that this simple approach offers excellent speedup provided all boxes have similar execution time characteristics. However, an arbitrary Hume program composed of arbitrary boxes is unlikely to offer the required degree of regularity. Thus, performance of our naive one box/one thread implementation is always limited by the time to execute the slowest box.

Our long term goal is to use box execution time analysis to enable load balancing, where multiple boxes may be assigned to a single thread. However, optimal strategies for using such analyses are still not clear. In particular, Hume resource analysis has focused on worst case as resource bound critical systems must necessarily be conservative in estimates. However, boxes are essentially assemblies of alternative matches, and in general there will be no consistency of individual match cost. In any case, match choice at run-time will depend on unknown properties of input data. Perhaps average case or probabalistic techniques might be used to better constrain box cost characterisation but these remain longer term research goals.

We have been exploring an alternative approach based on program transformation to systematically construct a multi-box program with the desired characteristics from an original functional expression. Transformation is driven by our Hume box calculus.

In the following sections we introduce the box calculus, explain the general principle for applying it to an expression to produce a well formed multi-box program, apply the technique to a matrix multiplication example, evaluate the technique on a multicore system, and reflect on contributions and limitations.

2 Hume Box Calculus

The Hume box calculus [18] is a set of rules for manipulating programs.

The main classes of rules are:

- *introduction*: wire, id box, transition;
- *elimination*: wire, id box, transition;
- *split*: box horizontally/vertically, wire, transition;
- *join*: horizontal/vertical boxes, wires, transitions.

There are also composite rules, which may be derived from the basic rules, for example to convert:

– recursive expression to/from iterative box;
– nested expression to/from vertical boxes;
– tuple expression to/from horizontal boxes.

Hume is unusual in distinguishing explicitly between the *coordination layer*, concerned with interactions amongst boxes and with the wider environment, and the *expression layer*, concerned with pattern/control transitions within boxes. It is important to note that transformations to one layer will almost invariably involve changes to the other. For example, introducing/eliminating a wire, necessarily between boxes, also requires the introduction/elimination of a pattern element for an input wire or a control element for an output wire, in every transition of both boxes. Thus, the box calculus integrates techniques from both the functional (i.e. expression) and the process (i.e. coordination) traditions.

While many rules are not correctness preserving they have well defined effects on program characteristics. Furthermore, rules which are correctness preserving may nonetheless have profound, if predictable, effects on program pragmatics, for example modifying time or space requirements, or scheduling behaviour. For the correctness of the calculus, see [17, 18].

The calculus was original proposed for the hierarchical extension of Hume, called Hierarchical Hume [19]. Whilst preservings functional correctness, changes of the coordination layer may change global properties, thus changing the overall behaviour. The advantage of Hierarchical Hume is that nesting mitigates the requirement of global analysis. However, the example in this paper is irrelevant because we do not use the '`*`' (ignore) pattern, and do not wire the transformed boxes to boxes where this pattern is applied.

3 Transforming Hume for Parallelism

We now describe the two key transformations for constructing a multi-box program from an original single box. Both will be familiar as Hume variants of standard skeletonising transformations for higher order functions (HOFs). Note that we present informal explanations followed by sketch proofs of correctness.

To simplify the presentation, we will consider transformations on a box with one input, one output and one transition, of the form:

```
box general
in (input :: type_1)
out (output :: type_N)
match pattern -> expression;
```

shown diagramatically in Figure 1.

First of all, if the match expression has the form of composed functions

var `->` $f_2(f_1(var))$

where:

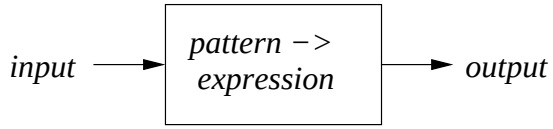

Fig. 1. General box.

box first	box second
in (input::$type_1$)	in (input::$type_3$)
out (output::$type_3$)	out (output::$type_2$)
match var -> f_1(var);	match var -> f_2(var);
wire first(...)(second.input)	wire second(first.output)(...)

Fig. 2. Pipeline form.

$f_1 :: type_1 \rightarrow type_3$; $f_2 :: type_3 \rightarrow type_2$

then the box may be transformed to a pipeline of two boxes, one for each function shown in Figure 2 and illustrated in 3(a).

The pipeline box transformation is achieved directly with the *vertical composition elimination* rule:

$$\textbf{VCompE}(\texttt{general}, \texttt{first}, [\texttt{output}], \texttt{second}, [\texttt{input}])$$

This rule assumes that box `general` has one match with expression of the form $f \cdot g$ (which is the case for this example with $f \mapsto f_2$ and $g \mapsto f_1$). It then vertically decomposes `general`, into two boxes `first` followed by `second` – with the output wire of `first` named `output` and input wire of `second` labelled `input`. Note that the input `first` has the same label as `general`, and so does the output wire of `second`.

Secondly, if the match effects a divide and conquer:

x `-> let (l,r) =` *divide* x `in` *conquer*(`(`f_1`(l),`f_2`(r))`

where:

divide :: $type_1 \rightarrow type_2 * type_3$
$f_1 :: type_2 \rightarrow type_4$
$f_2 :: type_3 \rightarrow type_5$
conquer :: $type_4 * type_5 \rightarrow type_6$

then the box may be transformed to a divide and conquer form shown in Figure 4, and graphically illustrated in Figure 3(b).

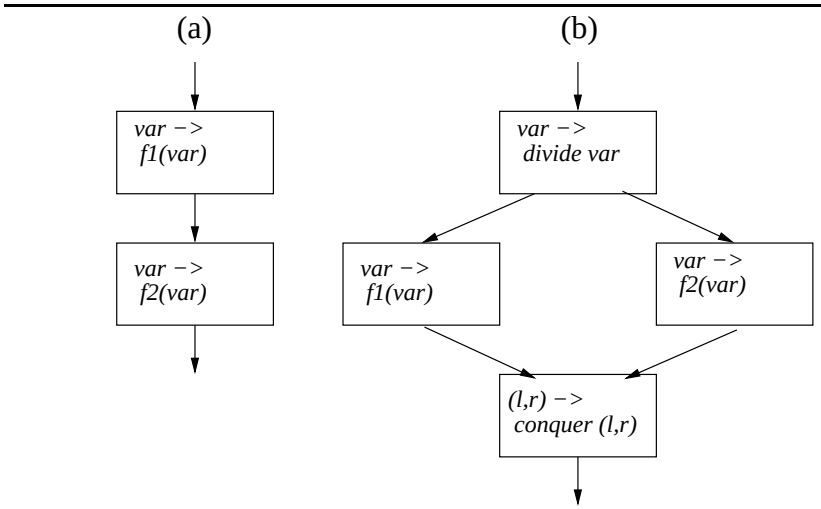

Fig. 3. Box transformed forms: (a) pipeline; (b) divide and conquer.

This transformation is more involved then the previous, and we will show it step-by-step. Figure 5 shows the effect each step has on the coordination layer.In the text we have references each transformation step in the figure – and omitted steps where this layer is unchanged.

First we apply some simple re-factoring of the expression of the original box `general`. We assume that:

$$divide\ x = (first\ x, second\ x)$$

Thus,

`let (l,r) =` $divide\ x$ `in ...`

is rewritten into

`let l =` $first\ x$ `in let r =` $second\ x$ `in ...`

We then unfold the `let` expressions, creating the following match

x `->` $conquer$`(`f_1`(`$first\ x$`),`f_2`(`$second\ x$`))`

2) We next apply the vertical composition elimination rule:

VCompE(`general`, `first`, [`left`,`right`], `conquer`, [`left`,`right`])

This will create two boxes: `first` with the match

x `->` `(`f_1`(`$first\ x$`),`f_2`(`$second\ x$`))`

box divide in (input::$type_1$) out (left::$type_2$,right::$type_3$) match var -> *divide* var;	box left in (input::$type_2$) out (output::$type_4$) match var -> f_1(var);
box right in (input::$type_3$) out (output::$type_5$ match var -> f_2(var);	box conquer in (left::$type_4$,right::$type_5$) out (input::$type_6$) match (left,right) -> *conquer*(left,right);
wire divide (...) (left.input,right.inout); wire right (divide.right) (conquer.right);	wire left (divide.left) (conquer.left); wire conquer (left.output,right.output) (...);

Fig. 4. Divide and conquer form.

followed by a box `conquer`, with the match

`(left,right) ->` *conquer* `(left,right)`

The `conquer` box will not be transformed further.

3) We then introduce an identity box `id` before the `first` box, by applying the *identity introduction* rule:

$$\textbf{IdI}(\texttt{first}, \texttt{input}, \texttt{id})$$

This introduces a box called `id` before the wire `input` of box `first` with the trivial match

x `->` x

The input/output wires of `id` becomes `v/v'` on default. We then duplicate this introduced wire, achieved by the *duplicate wire introduction* rule:

$$\textbf{DupI}(\texttt{id}, \texttt{v'}, \texttt{v''}, \texttt{first}, \texttt{input}, \texttt{y})$$

The match of `id` now becomes:

x `->` `(`x`,`x`)`

while the match of `first` becomes:

`(`x`,`y`) ->` `(`f_1`(`$first\ x$`),`f_2`(`$second\ x$`))`

4) x and y will always be identical (since they come from the `id` box which simply fan-out the same value). x may thus be replaced by y in any expression. Hence, in the second element of the expression pair of `input` we replace x by y by a simple re-factoring, thus creating the match:

(x,y) -> $(f_1(first\ x), f_2(second\ y))$

The match now consists of two input and two output pairs, where the first output only uses the first input, and the second output only uses the second input. This enables the *horizontal composition elimination* rule, which splits this box into two parallel boxes:

HCompE(`first`, [`input`].[`left`], `left`, `right`)

This creates two parallel boxes named `left` and `right`, where the input/output of box `left` is `input`/`left` (and `y`/`right` for the box `right`). The `left` and `right` boxes will have the following matches respectively:

x -> $f_1(first\ x)$ and y -> $f_2(second\ y)$

5) Both matches have the approipriate structure for the vertical composition elimination rule which we apply to both boxes:

VCompE(`left`, `first1`, [`left`], `left`, [`x`])
VCompE(`right`, `first2`, [`right`], `right`, [`y`])

The new boxes `first1` and (followed by) `left` will then have the following respective matches

x -> $first\ x$ and x -> $f_1\ x$

Similarly, the matches of `first2` and `right` become

y -> $second\ y$ and y -> $f_2\ y$

6) We can now apply the *horizontal composition introduction* rule to `first1` and `first2`, creating the `divide` box:

HCompI(`first1`, `first2`, `divide`).

which contains following match:

(x,y) -> $(first\ x, second\ y)$

Both inputs of this box comes from the `id` box, and are both duplicates of each other. Thus y can be rewritten to x in the expression:

(x,y) -> $(first\ x, second\ x)$

We can then apply the *duplicate wire elimination* rule on the input on y:

DupE(`divide`, `input`, `y`)

creating the match:

x -> $(first\ x, second\ x)$

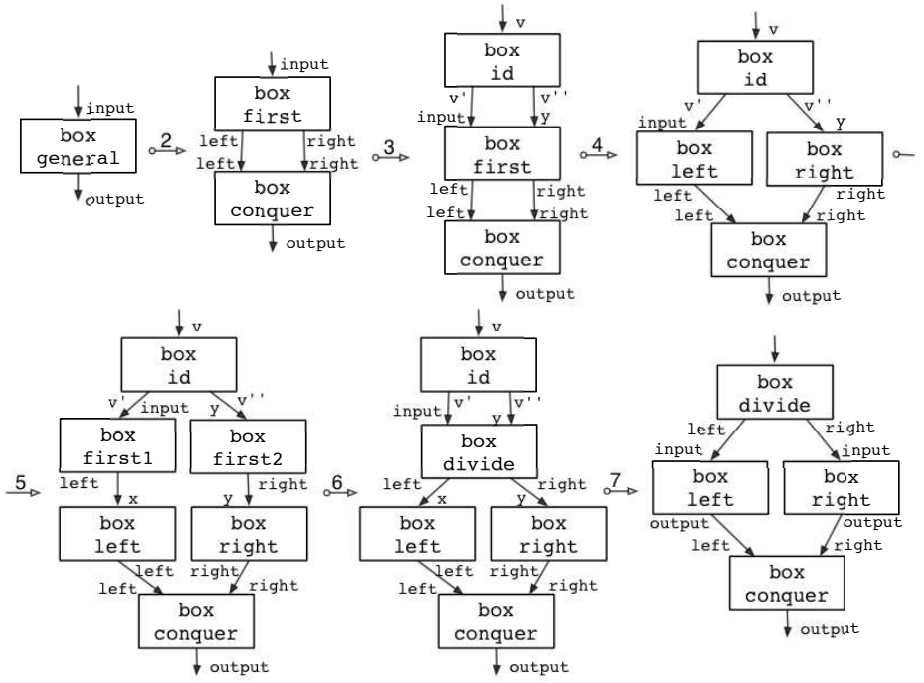

Fig. 5. Divide and conquer transformation steps.

7) We eliminate the identity box `id` with the *identity box elimination* rule, and applies some simple renaming of the inputs and outputs wires of the `left` and `right` boxes. This concludes the transformations required on the coordination layer:

> **IdE**(`id`); **VRename**(`left`,`x`,`input`); **VRename**(`left`,`left`,`output`); **VRename**(`right`,`y`,`input`); **VRename**(`right`,`right`,`output`)

In the final step we apply the opposite re-factoring of the expression layer of `divide` as we did in the first step. Firstly, we introduce the `let` expressions, thus creating the match:

> x `-> let l =` *first* x `in let r =` *second* x `in (l,r)`

At the end, we rewrite this to the *divide* function, using the equality defined above:

> x `-> let (l,r) =` *divide* x `in (l,r)`

4 Example: matrix multiplication

Consider the usual pure functional matrix multiplication based on a list representation of matrices in row major order, requiring transposition of the second matrix::

```
dist [] _ = [];
dist (h1:t1) (h2:t2) = (h1:h2):dist t1 t2;
dist (h1:t1) [] = [h1]:dist t1 [];

transpose [] = [];
transpose (row:rows) = dist row (transpose rows);

dotprod (h1:t1) (h2:t2) = h1*h2+dotprod t1 t2;
dotprod _ _ = 0;

rowmult r (c:t) = dotprod r c:rowmult r t;
rowmult _ [] = [];

rowsmult (r:t) c = rowmult r c:rowsmult t c;
rowsmult [] _ = [];

matmult m1 m2 = rowsmult m1 (transpose m2);
```

Assuming we have appropriate boxes to handle matrix input from file and matrix display, and ignoring the attendant wiring, then our initial one box program is:

```
type integer = int 64;
type intmat = [[integer]];
box matrixmult
in (m1,m2::intmat)
out (m3::intmat)
match
 (m1,m2) -> rowsmult m1 (transpose m2);
```

which will repeatedly input and multiply matrices from wires `m1` and `m2`.

Observing that the transition has the form *f(g(var))*, we apply the first transformation above to pipeline the transposition phase into the multiplication phase, as shown in Figure 6. Next, noting that given:

```
take _ [] = [];
take 0 _ = [];
take n (h:t) = h:take (n-1) t;

drop _ [] = [];
drop 0 l = l;
drom n (h:t) = drop (n-1) t;
```

box matrixmult1	box matrixmult2
in (m1::intmat,m2::intmat)	in (input::(intmat,intmat))
out (output::intmat)	out (output::intmat)
match	match (m1,m2) -> rowsmult m1 m2;
(m1,m2) -> (m1,transpose m2);	
wire matrixmult1	wire matrixmult2
(...) (matrixmult2.input);	(matrixmult1.output) (...);

Fig. 6. Pipelined matrix multiplication.

```
map f [] = [];
map f (h:t) = f h:map f t;
```

then for list l the following identities hold:

$l \equiv$ `take` n l`++drop` n l
`map` f $l \equiv$ `(map` f `(take` n l`))++(map` f `(drop` n l`))`

so `rowsmult` may be recast as an append of `map`s:

`onerow m2 r = rowmult r m2;`
`rowsmult m1 m2 = map (onerow m2) m1` $\Leftrightarrow$
`map (onerow m2) (take` N `m1)++map (onerow m2) (drop` N `m1)` $\Leftrightarrow$
`rowsmult (take` N `m1) m2++rowsmult (drop` N `m1) m2`

Now the second box's transition has the required divide and conquer form, and the second transformation applies, as shown in Figure 7 The final program has the form shown in Figure 8.

Note that we can apply the second transformation repeatedly to new boxes of the correct form and then apply the box calculus to remove redundant intermediate boxes. For exampling, transforming both `rowsmult` boxes in Figure 8 gives Figure 9(a), with intermediate trees of divide and conquer boxes. By application of horizontal box merge, wire merge and vertical box merge, this can be simplified to Figure 9(b).

5 Evaluation

We timed various versions of the matrix multiplication program on an Intel Xeon (R) 2.3 GHz processor with eight cores and 6144 KB cache. Tests involved multiplying 20x16 matrices.

Table 1 shows the run-time and speed up by number of boxes and number of cores. While times seem slow, it should be remembered that we are measuring the HAM interpreter, not native code. Speed up is very decent overall, with a best improvement of 6.8 for 8 boxes on 8 cores.

Figure 10 shows analysis of core behaviour. The third column records the number of core cycles while the core is not in a halt state. The fourth column counts the number of instructions that retire execution. For instructions

```
box matrixmult2
in (input::(intmat,intmat))
out (left,right::(intmat,intmat))
match
(m1,m2) ->
let N = length m1 div 2
in ((take N m1,m2),(drop N m1,m2));

box mm1
in (input::(intmat,intmat))
out (output:intmat)
match (m1,m2) -> rowsmult m1 m2;

box mm2
in (input::(intmat,intmat))
out (output:intmat)
match (m1,m2) -> rowsmult m1 m2;

box conquer
in (left,right::intmat)
out (output::intmat)
match (m1,m2) -> m1++m2;

wire matrixmult2
(matrixmult1.output) (mm1.input,mm2.input);
wire mm1 (matrixmult2.left)) (conquer.left);
wire mm2 (matrixmult2.right)) (conquer.right);
wire conquer (mm1.output,mm2.output) (...);
```

Fig. 7. Divide and conquer matrix multiplication.

Box	8								6					4			3	
PEs	1	2	3	4	5	6	7	8	2	3	4	5	6	2	3	4	2	3
RTs	23.4	19.9	16.5	12.9	9.8	8.2	6.8	3.4	17.2	14.4	10.0	6.2	4.9	18.2	12.4	7.8	15.3	11.1
Spd	1	1.17	1.4	1.8	2.3	2.8	3.4	6.8	1.3	1.6	2.3	3.7	4.7	1.3	1.8	3.0	1.5	2.1

Table 1. matrix-multiplication

that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. In the fifth column high `clocks per instructions retired` (CPI) indicates that instructions require more cycles to execute than they should. CPI can get as low as 0.25 cycles per instructions. This shows excellent consistency of core behaviour.

The complementary Figure 11 shows the thread call graph. After the initial startup, cores remain highly active and well balanced.

6 Related work

6.1 Multi-core

The recent trend towards multicore architectures has sparked a significant new work aimed at exploring novel programming models and runtime systems. Approaches include:

– *Parallel libraries*, such as *Pthreads* [24] and provide the ability to express parallelism directly. More advanced libraries, such as Cilk [16] or OpenMP [11], provide higher level primitives, including, for example, support for nested parallelism [13]. However, according to Bridges [7], library approaches provide little support to help achieve correct or effective parallelism.

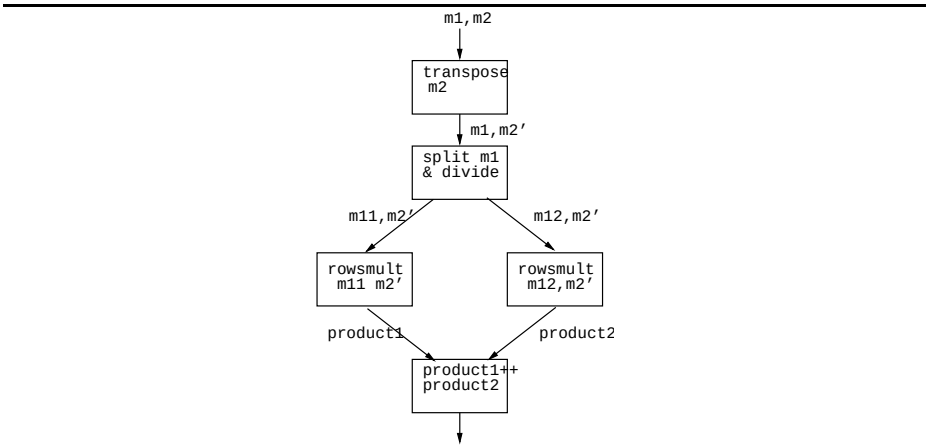

Fig. 8. Transformed matrix multiplication.

- *Message-passing approaches*, such as the GUM implementation of Haskell for multicore machines [5], or Erlang [2, 14]. The Haskell approach uses algorithmic skeletons to introduce parallelism mapped to multicore threads executing sequential components. The Erlang approach is based on explicit threads which are mapped directly to operating system threads. In GUM, like the multicore HUME implementation , all communication and thread synchronisation is implicit, whereas Erlang requires explicit use of message-passing primitives. Unlike GUM, Hume makes direct use of shared memory mechanisms. Moreover treating Hume boxes as units of execution makes it straightforward to derive parallelism that can easily be mapped to multiple cores.
- *Explicit memory transactions* [9, 21] attempt to reduce locking by exposing parallel operations as transactions. Some approaches require an explicit step to make locations or objects part of a transaction, while others make the memory operation behaviour implicit, requiring compiler or hardware support. While such approaches appear promising on paper, they have so far generally failed to deliver good performance. For example, Harris et al [21] report poor and highly variable parallel performance, using memory transaction techniques on shared-memory machines.
- *Data-parallel* approaches where parallelism is exposed by evaluating elements of bulk data structures in parallel. For example, Data-Parallel Haskell [10] provides parallel arrays and operations. Good results are reported for typical data-parallel problems. A similar approach is taken by Fluet *et al.* [15] who embed nested data-parallel constructs into an explicitly parallel Concurrent ML setting. These approaches can deliver good performance but are best suited to static, regular, data-driven parallelism, rather than the more dynamic, irregular forms that can be programmed using Hume boxes.

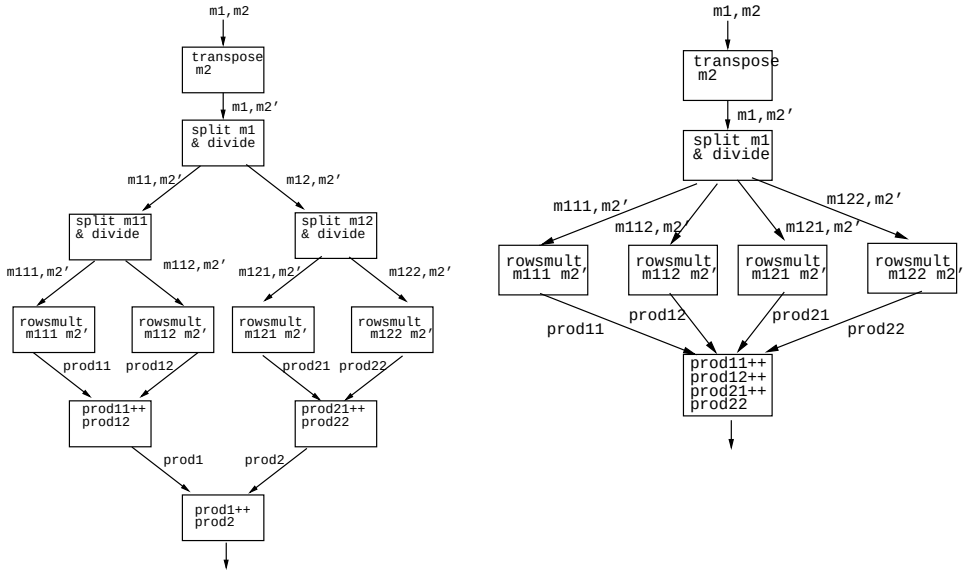

Fig. 9. Multiple divide and conquer transformations(a) and simplification(b).

Thread	Process	CPU_CLK_UNHALTED samples	INST_RETIRED.ANY samples	Clocks per Instructions Retired - CPI	CPU_CLK_UNHALTE %	INST_RETIRED.ANY %	CPU_CLK_UNHALTE events	INST_RETIRED.ANY events	ThreadID
thread5	hami	4,079	3,455	1.181	12.83%	10.61%	8,158,000,000	6,910,000,000	18507
thread6	hami	4,061	4,340	0.936	12.77%	13.33%	8,122,000,000	8,680,000,000	18511
thread7	hami	4,036	4,164	0.969	12.69%	12.79%	8,072,000,000	8,328,000,000	18510
thread9	hami	3,955	4,321	0.915	12.44%	13.27%	7,910,000,000	8,642,000,000	18513
thread10	hami	3,948	4,152	0.951	12.42%	12.75%	7,896,000,000	8,304,000,000	18514
thread11	hami	3,942	4,103	0.961	12.40%	12.60%	7,884,000,000	8,206,000,000	18515
thread12	hami	3,933	4,097	0.960	12.37%	12.58%	7,866,000,000	8,194,000,000	18512
thread14	hami	3,840	3,925	0.978	12.08%	12.06%	7,680,000,000	7,850,000,000	18516

Fig. 10. Analysis

6.2 Box calculus

In contrast to most of the approaches described above, we focus on exploiting parallelism in multicore architecture by introducing boxes as the *right-size construct* for mapping to cores, complemented by the box calculus.

The box calculus [18] was preceded by informally applying the horizontal box integration rule to establish that Finite State Machine (FSM-)Hume actually is finite state[27]. The current paper is the first application of the box calculus to a substantive problem. As in e.g. *refinement calculus*[3], which influenced notations like Z, B and Event-B, the box calculus identifies that a transformations consists of several small steps and a bigger step is achieved by combining them into strategies. We expect that this will greatly assist automatation of the proof of a rule application within a theorem prover.

Due to the strong interplay and dependencies between the Hume expression and coordination layer, a single rule application will often have impact on both layers, which is distinctive compared with synthesis techniques, like

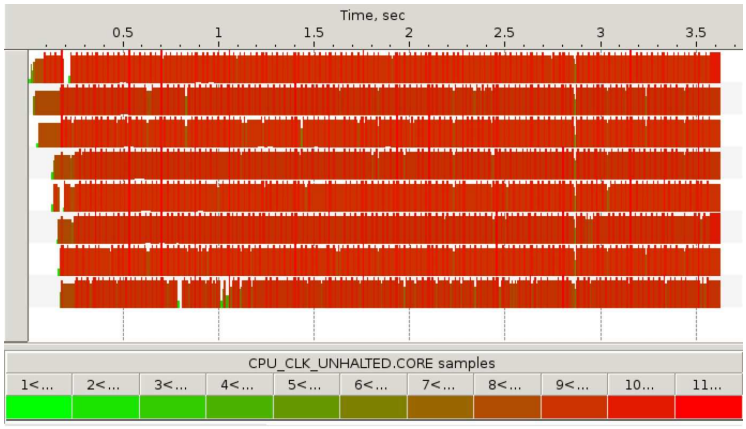

Fig. 11. Per Thread Call Graph

Bird-Meertens Formalism [6] and calculational programming [22]. Just as Hume integrates a finite state coordination language with a functional transition control language, the work presented here draws on the twin traditions of process network and functional program transformation. The coordination aspects of the rules have many similarities with those found in the box calculus for Petri nets [12] as well as process calculi[4]. The control aspects resemble classic functional programming techniques including curry/uncurry, fold/unfold[8] and functional refactoring[25]. However, with respect to functional program transformations, we expect to be able to use these rules directly for the purely expression layer refactorings (like unfolding/folding `let` expression above), while process calculi, like CCS or CSP, can be used to embed the box calculus.

However, an underlying state based representation, like TLA[23], as used in [17], seems very promising. Experiments have been conducted in integrating this TLA-based coordination layer representation with a deep embedding of the Hume expression layer within the Isabelle theorem prover [26]. We believe this representation will be ideal for representing the full box calculus.

7 Conclusion

We have shown that it is possible to achieve good multicore parallelisation of Hume programs through systematic transformation guided closely by the box calculus. The underlying implementation offers seamless parallelism without further programmer intervention.

A major objective is to ensure that all boxes have roughly the same processing cost. In our example, this was straightforward as we applied a uniform, data directed, horizontal partition to a regular linear algorithm. For general algorithms, ensuring box balance is likely to require considerably more ingenuity.

However, given the presence of robust models and analyses for space and WCET, Hume is well placed to support *transformation by cost* where applying a rule to a construct has a known effect on the construct's, and hence the whole program's, resource requirements. In the longer term, we would like to explore how the box calculus may be augmented with cost judgements, to underpin the integration of Hume cost technology into a computer assisted program transformation IDE.

Acknowledgements

This work is funded by UK EPSRC Islay project (EP/F030592, EP/F030657, EP/F03072, and EP/F031017) "Adaptive Hardware Systems with Novel Algorithmic Design and Guranteed Resource Bounds".

We would like to thank our colleague Kevin Hammond, who built the original HAM interpreter and conducted the first multicore HAM experiments.

References

1. A. Al Zain, G. Michaelson, and K. Hammond. Multi-core Parallelisation for Hume. In Z. Horvath, V. Zsok, P. Achten, and P. Koopman, editors, *Tenth Symposium on Trends in Functional Programming, Komarno, Slovakia, 2-4 June 2009*, pages 131–142, 2009.
2. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in* ERLANG. Prentice Hall, 2nd edition, 1996.
3. R.-J. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, The Netherlands, 1980.
4. J. C. M. Baeten. A Brief History of Process Algebra. *Theoretical Computer Scisence*, 335(2-3):131–146, 2005.
5. J. Berthold, S. Marlow, A. Al Zain, and K. Hammond. Comparing and Optimising Parallel Haskell Implementation. In *ICPP 09 – International Conference on Parallel Processing*, Vienna, Austria, Sept. 2009. IEEE Computer Society.
6. R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, 1997.
7. M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the Sequential Programming Model for Multi-Core. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society.
8. R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.
9. B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*. Jun 2006.
10. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a Status Report. In *DAMP'07: Workshop on Declarative Aspects of Multicore Programming)*, Nice, France, 2007. ACM Press.

11. B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.
12. R. Devillers, H. Klaudel, and R. C. Riemann. General Parameterised Refinement and Recursion for the M-net Calculus. *Theoretical Computer Science*, 300(1-3):259–300, May 2003.
13. A. Duran, M. Gonzàlez, and J. Corbalán. Automatic Thread Distribution for Nested Parallelism in OpenMP. In *19th Annual International Conference on Supercomputing (ICS '05)*, pages 121–130, New York, NY, USA, 2005. ACM.
14. Ericsson Utvecklings AB. *Erlang Home Page*.
15. M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A Heterogeneous Parallel Language. In *DAMP'07: Workshop on Declarative Aspects of Multicore Programming*, Nice, France, 2007.
16. M. Frigo, C. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI'98 — Conf. on Programming Language, Design and Implementation*, volume 33 of *ACM SIGPLAN Notices*, pages 212–223. ACM Press, 1998.
17. G. Grov. *Reasoning About Corectness Properties of a Coordination Language*. PhD thesis, School of Mathematical and Computer Sciences, Heriot-Watt University, 2009.
18. G. Grov and G. Michaelson. Towards a Box Calculus for Hierarchical Hume. In M. Morazon, editor, *Trends in Functional Programming*, volume 8, pages 71–88, 2008.
19. G. Grov, R. Pointon, G. Michaelson, and A. Ireland. Preserving Coordination Properties when Transforming Concurrent System Components. In *Coordination Models, Languages and Applications Track of the 23rd Annual ACM Symposium on Applied Computing*, volume 1 of 3, pages 126 – 127, 1515 Broadway, New York, March 2008. The Association for Computing Machinery, Inc.
20. K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
21. T. Harris, S. Marlow, and S. P. Jones. Haskell on a Shared-Memory Multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61. ACM Press, September 2005.
22. G. Hutton and J. Wright. Calculating an Exceptional Machine. In H.-W. Loidl, editor, *Trends in Functional Programming*, volume 5, pages 49–64, 2006.
23. L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
24. B. Lewis and D. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.
25. H. Li and S. Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs. In *Proceedings of 6th IEEE Workshop on Source Code Analysis and Manipulation, Philadelphia, USA*, pages 197–206, September 2006.
26. H.-W. Loidl and G. Grov. Assertion Language. EmBounded Project Deliverable, Oct. 2007. Deliverable D17. Available at `http://www.embounded.org/`.
27. G. Michaelson, K. Hammond, and J. Sérot. FSM-Hume: Programming Resource-Limited Systems using Bounded Automata. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1455–1461. ACM Press, March 2004.