Towards Resource-Certified Software: A Formal Cost Model for Time and its Application to an Image-Processing Example

Armelle Bonenfant
School of Computer Science
University of St Andrews
St Andrews, UK
ab@cs.st-and.ac.uk

Zezhi Chen
Computer Science
Heriot-Watt University
Riccarton, Edinburgh, UK
zezhi@macs.hw.ac.uk

Kevin Hammond
School of Computer Science
University of St Andrews
St Andrews, UK
kh@dcs.st-and.ac.uk

Greg Michaelson
Computer Science
Heriot-Watt University
Riccarton, Edinburgh, UK
greg@macs.hw.ac.uk

Andy Wallace
Electrical and Computer Eng.
Heriot-Watt University
Riccarton, Edinburgh, UK
A.M.Wallace@hw.ac.uk

Iain Wallace
Electrical and Computer Eng.
Heriot-Watt University
Riccarton, Edinburgh, UK

ABSTRACT

Visual tracking requires sophisticated algorithms working in real-time, and often space-limited, settings. While the input streams may be regular in structure, the algorithms are not, and must often deal with probabilistic metrics. To ensure progress in algorithm design without incurring excessive development costs, we propose a high-level programming approach married with predictable and compositional performance metrics. This enables the combination of independently developed program components into coherent software architecture, with certified resource use guarantee. Here, we present our approach and discuss its application to the development and resource analysis of a space bound mean shift algorithm for motion tracking, using the new embedded system-oriented language Hume.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification — Correctness proofs, Formal methods; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Embedded systems, resource bounds, motion tracking, functional programming

SAC'07 March 11-15, 2007, Seoul, Korea

1. INTRODUCTION

As part of a UK MOD-funded project, we are investigating the application of strong formal cost modelling to assist the construction of advanced real-time embedded systems. The challenge is to validate and verify resource-bounded code in a high-level domain-specific programming language in such a way that compiled firmware and software on a microcontrolled autonomous vehicle (UV) behaves predictably in terms of algorithmic performance, and so that predictable bounds can be obtained on both execution time and memory usage. The principal novelty of the work lies in the development of higher-level software to enable fast, certifiable prototyping and performance modelling of advanced sensor interpretation algorithms. Our approach offers the prospect of producing reliable and predictable embedded software which exploits a formal model with strong theoretical properties, but which is grounded in practical applications.

The sensor behaviour we must consider as part of our work goes beyond simple image-processing, frequency-transformation and template-matching algorithms in which the underlying data structures are static and predictable. We will therefore need to exploit advanced dynamic data structures and adaptive processing strategies that are necessary for real-time scene interpretation. This makes resource prediction very challenging or even impossible using existing approaches. We foresee two possible routes to the deployment of novel sensor algorithms in a UV, first through our new language Hume [13], which offers the twin advantages of higher-level programming combined with a considerable pedigree in space and time analysis for embedded systems [27, 32, 14]. The alternative, and more usual, engineering approach consists of empirical analysis on simulation packages that incorporate semi-formal models of the target architecture. Our success will be based on comparison of the measured, worst-case execution times and space consumptions of the final algorithms against the cost model predictions, using a wide range of operational contexts.

2. THE HUME APPROACH

In this paper, we focus on the more novel Hume approach.We have developed a new domain-specific programming language, Hume, which applies functional programming language technology to resource-bounded computations in the real-time embedded systems domain. Hume is based on concurrent finite-state-automata, termed *boxes*, linked by *wires*, and driven by *pattern-matching* and *recursive functions* over *rich data types*. It provides high-level features including strong polymorphic typing, recursion, higher-order functions, complex user-defined data structures, and automatic memory management, coupled with costable exception handling, simple control of asynchronous concurrency, interrupt handling, real-time control, and scheduling. Memory management is enforced by a model of independent box execution which allows dynamic memory for each box to be allocated independently, and where all dynamic memory allocated to a box may be automatically collected after each box iteration [14].

For simplicity of costing, Hume is a strict language: all arguments to a function are evaluated before the function body is executed. Hume programs are typically much shorter than the corresponding Ada, Java or C equivalents: in the extreme, as with other high-level functional language approaches, they may be about one-tenth of the size; and they are typically around one-third or one-quarter of the size [26]. This is significant since productivity and error rates are known to correspond roughly linearly with program size.

It is well known that many interesting properties, such as program termination, equivalence, and time and space use, are undecidable for Turing Complete languages. Thus, Hume has been conceived of as a multi-level language where different levels have different decidability properties. First, *HW-Hume*, an impoverished language for manipulating tuples of bits, is fully decidable. Similarly, *FSM-Hume*, which extends HW-Hume with richer but bounded data types and first-order functions, has strongly bounded time and space behaviour, enabling highly accurate resource predictions. *Template-Hume* further extends FSM-Hume with a fixed set of higher-order functions that generalise operations on data structures. These include `map`, which applies a function to every element of a structure, and `fold`, which applies a function across a structure to accumulate a result involving each element. The more abstract *PR-Hume*, which extends FSM-Hume with primitive recursion over unbounded data types, offers far weaker time and space guarantees. Finally, *full-Hume*, with the same expressive power as other general-purpose programming languages, also shares these analytic limitations: like C, Ada or Java, it is not possible to provide resource bounds for arbitrary full-Hume programs.

Rather than requiring the direct use of an impoverished language as has been suggested elsewhere (e.g. [30, 6]), we propose that programs are first developed in full-Hume and then adapted to lower levels, as necessary in order to expose the required resource properties through analysis. While, in practice, programmers will naturally deploy constructs that are amenable to analysis, such as bounded iteration over finite data structures, it is inevitable that analyses will fail for some full-Hume language constructs or uses. We propose to use such failures iteratively, to identify problematic constructs in programs, which may be reformulated in a less expressive Hume level, to increase the scope of analysis.

	Semantics		Cost Model
HUME	code		
	↓ translation		
HAM	code	$\xrightarrow{\text{evaluation}}$	cost

Table 1: A Framework for Resource Modelling and Analysis

In turn, such failures will guide the iterative improvement of analyses by identifying common problematic constructs for which further support is required. Finally, as part of our research programme, we will explore the formulation of meaning-preserving transformations between Hume levels, with potential for automatic program manipulation. Since lower Hume levels are a strict subset of higher Hume levels, it follows that any lower level Hume program or fragment may be considered to also be at any required higher level. Conversely, we anticipate that the abstraction mechanisms of Hume will allow automatic transformation to lower Hume levels, and that it will not normally be necessary for the programmer to consider the transformed code.

3. RESOURCE MODELS AND ANALYSES

Obtaining high-quality *guarantees* for worst-case resource consumption requires us to construct mathematical models of the target architectures, both at the abstract machine level and at the concrete architecture level. These models must expose the *behavioural* properties of the program execution that we are interested in, notably time usage and space consumption [14, 32]. In order to obtain a resource model for program source, we must also formally relate these low-level models to our source language, in this case, Hume. The approach is shown in Table 1. In this paper, we will show sample cost rules for yielding time information from the Hume source forms given above, and will assume a formal operational model for the underlying Hume abstract machine (HAM), and the associated translation and correspondance proof, which are under construction. We focus on expression forms, since these are key to constructing good cost models, but the rules can be straightforwardly extended in a structural way to cover all Hume constructs, including exceptions, concurrency and communication. Our rules are given in a derivation form as follows:

$$\mathcal{V}, \eta \vdash^{t}_{t'} e \rightsquigarrow \ell, \eta'$$

Here e represents an expression in our source language. t is an upper bound of the number of available time units to evaluate e, and t' is the number of time units still available after execution of e. The time required for evaluating e will then be $t - t'$. η/η' are the dynamic memory before/after execution of e, ℓ is the result value after execution, and $\mathcal{V}$ represents a mapping of variable names to values. In order to simplify the extraction of lower-level costs in the first instance and to allow these to be easily related to the source through an abstract machine model, we assume a simple execution model, where all values are *boxed* (heap-allocated) at run-time, and all arguments are passed by reference on the stack. This is not a fundamental limitation of the approach, however: optimisations such as unboxing or register-allocation can be easily incorporated, but must be reflected

$expr ::= int$	integer constants
$\mid x$	variables
$\mid fid \;\; expr_1 \;\; \ldots \;\; expr_n$	function application, n ≥ 1
$\mid \langle\!\langle \;\; expr_1 \;\; \ldots \;\; expr_n \rangle\!\rangle$	vector construction, n ≥ 0
$\mid$ **if** $cond$ **then** $expr_1$ **else** $expr_2$	conditionals

Figure 1: Syntax for the Subset of full-Hume Expressions used in this Paper

in the cost rules, in the underlying operational semantics and in the proofs of correspondance.

In order to cost the pattern matching, the following statement has been defined:

$$\eta \vdash^{t}_{t'} \ell : \vec{\ell}, pat : \vec{pat}, \mathcal{V}, \mathcal{A} \triangleright \vec{\ell}, \vec{pat}, \mathcal{V}', \mathcal{A}'$$

It means that one step of matching $\vec{\ell}$ against $\vec{pat}$ succeeds (with $\vec{\ell}$ interpreted within heap η). In order to complete the whole pattern match, $\vec{\ell}'$ and $\vec{pat}'$ must still be matched, which will usually be sublists of $\vec{\ell}$ and $\vec{pat}$ respectively. In addition $\mathcal{V}'$ is the environment obtained from $\mathcal{V}$ plus the bindings that occured by this step of the match; while $\mathcal{A}'$ contains the set locations that have been matched successfully in this step plus those of $\mathcal{A}$. The set $\mathcal{A}'$ will be used at the scheduler level to mark the wires which have been consumed in case of an entirely successful match. $t - t'$ is the time required to match the location ℓ against pat.

We simply write

$$\eta \vdash^{t}_{t'} \vec{\ell}, \vec{pat}, \mathcal{V}, \mathcal{A} \triangleright^{*} \vec{\ell}', \vec{pat}', \mathcal{V}', \mathcal{A}'$$

to denote that the quadruple $\vec{\ell}, \vec{pat}, \mathcal{V}, \mathcal{A}$ reduces in several steps to the quadruple $\vec{\ell}', \vec{pat}', \mathcal{V}', \mathcal{A}'$, which is irreducible under $\triangleright$.

3.1 A Time Cost Model for a Subset of Hume

This section presents formal cost rules for a subset of full-Hume expressions, corresponding to the syntax of Figure 1, representing the core of a typical strict functional language. The rules provide a model for execution cost rather than an analysis of this cost, e.g. as an abstract interpretation. They do however provide a basis for the construction of such an analysis. A sample hand derivation using these rules to predict execution cost will be given in Section 5. We first consider constant integers.

$$\frac{n \in \mathbb{Z} \qquad \textsc{New}(\eta) = \ell \qquad w = (\mathsf{int}, \mathsf{n})}{\mathcal{V}, \eta \vdash^{t' + \texttt{Tmkint}}_{t'} n \rightsquigarrow \ell, \eta[\ell \mapsto w]} \;(\textsc{Const Int})$$

The Const Int rule allocates a new location ℓ for the given constant n. The cost of the evaluation is given by the constant `Tmkint`, which is the time required by the `MkInt` instruction in the underlying abstract machine, and which can be obtained either by analysis or by measurement. Similar rules can be constructed for any other kind of constant.

The Variable rule looks up x in $\mathcal{V}$ to provide the location of the variable. The time to do this is given by cost of the underlying `PushVar` instruction, `Tpushvar`.

$$\frac{\mathcal{V}(x) = \ell}{\mathcal{V}, \eta \vdash^{t' + \texttt{Tpushvar}}_{t'} x \rightsquigarrow \ell, \eta} \qquad (\textsc{Variable})$$

The Vector rule deals with costs for literal arrays (vectors). Each component of the vector is evaluated to yield a heap location. Since memory must be allocated for each component, the cost of constructing the vector therefore depends on the number of components `Tmkvec(k)` plus the costs for evaluating each component. We show the `Tmkvec()` costs below the line to make it clear that they are incurred after evaluating the vector components.

$$\frac{\begin{array}{c}\mathcal{V}, \eta_{(i-1)} \vdash^{t_{(i-1)}}_{t_i} e_i \rightsquigarrow \ell_i, \eta_i \quad (\text{for } i = 1, \ldots, k) \\ k \geq 1 \qquad \textsc{New}(\eta_k) = \ell \qquad w = (\mathsf{constr}_c, \ell_k, \ldots, \ell_1)\end{array}}{\mathcal{V}, \eta_0 \vdash^{t_0}_{t_k - \texttt{Tmkvec(k)}} \langle\!\langle e_k \cdots e_1 \rangle\!\rangle \rightsquigarrow \ell, \eta_k[\ell \mapsto w]} \;(\textsc{Vector})$$

The Call rule deals with function applications where the function name (fid) is known (for simplicity, we omit here the higher-order case where the function name is not known). The function arguments are evaluated to give heap locations, as with vectors, and the residual costs are then calculated using a set of auxiliary rules App 0, App, UnderApp or OverApp, which deal with the different kinds of application that may be encountered in a higher-order setting. In this case, we will show only the rule for the normal first-order case, App. The cost of the call is `Tcall` and the cost of completing the call is `Tslide`, where the underlying `Slide` instruction will remove function arguments that have been passed on the stack, whilst preserving the return value.

$$\frac{\begin{array}{c}\mathcal{V}, \eta_{(i-1)} \vdash^{t_{(i-1)}}_{t_i} e_i \rightsquigarrow \ell_i, \eta_i \quad (\text{for } i = 1, \ldots, k) \\ k \geq 0 \qquad \mathcal{V}, \eta_k \vdash^{t_k - \texttt{Tcall}}_{t'_a} \textsc{App}\big(fid, [\ell_k, \ldots, \ell_1]\big) \rightsquigarrow \ell, \eta'\end{array}}{\mathcal{V}, \eta_0 \vdash^{t_0}_{t'_a - \texttt{Tslide}} fid \; e_k \cdots e_1 \rightsquigarrow \ell, \eta'} \;(\textsc{Call})$$

Conditional True is the rule for conditionals in the case where the condition is true. The preconditions to the condition are above the bar and the outcome is below the bar. The time taken to execute the compiled expression in our abstract machine is the difference between the residual times on the turnstile; in this case, the time to evaluate the condition e_1, $(t_1 - t'_1)$, plus the time to evaluate the expression e_2, plus `Tiftrue` - the time to execute the underlying abstract machine instruction. The result of execution is the value that results from executing e_2, ℓ'', and the final dynamic memory configuration is η''.

$$\frac{\begin{array}{c}\mathcal{V}, \eta \vdash^{t_1}_{t'_1} e_1 \rightsquigarrow \ell, \eta' \\ \eta'(\ell) = (\mathsf{bool}, \mathtt{tt}) \qquad \mathcal{V}, \eta' \vdash^{t'_1 - \texttt{Tiftrue}}_{t'_2} e_2 \rightsquigarrow \ell'', \eta''\end{array}}{\mathcal{V}, \eta \vdash^{t_1}_{t'_2} \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rightsquigarrow \ell'', \eta''} \;(\textsc{Conditional True})$$

Finally, the App rule is broken down into several steps. First, all the given arguments have to be matched against the patterns, where the first $j-1$ matches will fail. For each match, the cost of `Tmatchrule` is added to the cost of evaluating the match. The successful match incurs an additional cost of `Tmatchedrule`. Finally the function body is evaluated. Prior to the evaluation, we incur a cost of `Tcreatef`, and afterwards the cost of `Treturn`, corresponding to the

time costs of constructing a stack frame and returning from the function call, respectively.

$$\frac{\begin{array}{c}\Sigma^{\mathcal{F}}(\mathit{fid}) = (k, [\vec{pat}_1 \texttt{ -> } e_1, \cdots, \vec{pat}_h \texttt{ -> } e_h]) \qquad 1 \leq k = b \\ (\text{for } i = 1, \ldots, j-1) \\ \begin{cases} \eta \vdash^{\frac{t_{(i-1)} - \texttt{Tmatchrule}}{t_i}} [\ell_1, \ldots, \ell_b], \vec{pat}_i, \varnothing, \emptyset \triangleright^* \vec{\ell}_i, \vec{pat}_i{}', \mathcal{V}_i, \mathcal{A}_i \\ \vec{\ell}_i \neq [] \qquad \vec{pat}_i{}' \neq [] \end{cases} \\ \eta \vdash^{\frac{t_{(j-1)} - \texttt{Tmatchrule}}{t_j + \texttt{Tmatchedrule}}} [\ell_1, \ldots, \ell_b], \vec{pat}_j, \varnothing, \emptyset \triangleright^* [], [], \mathcal{V}_j, \mathcal{A}_j \\ \mathcal{V}_j, \eta \vdash^{\frac{t_j}{t'_{e_j}}} e_j \rightsquigarrow \ell, \eta' \end{array}}{\mathcal{V}, \eta \vdash^{\frac{\texttt{Tcreatef} + t_0}{t'_{e_j} - \texttt{Treturn}}} \textsc{App}(\mathit{fid}, [\ell_1, \ldots, \ell_b]) \rightsquigarrow \ell, \eta'} \quad (\textsc{App})$$

All the expression forms we have discussed above are simple functional values. It is, however, easy to extend the rules to cover Hume exceptions. In Hume, exceptions may be raised within expressions, but must be handled at the box level, using simple constant expressions. It follows that there is a single handler for each exception, and that exceptions cannot be nested. In this way, we are able to give bounds on the cost of raising an exception (effectively a branch to the statically determined handler), and of handling an exception (since each exception handler has a simple, fixed cost).

$$\frac{\mathit{exn} \in \mathsf{Exn} \qquad \mathcal{V}, \eta \vdash^{\frac{t_e}{t'_e}} e \rightsquigarrow \ell, \eta' \qquad w = (\mathsf{exception}_{\mathit{exn}}, \ell)}{\mathcal{V}, \eta \vdash^{\frac{\texttt{Traise} + t_e}{t'_e}} \texttt{raise } \mathit{exn}\ e \rightsquigarrow 0, \eta'[0 \mapsto w]} \quad (\textsc{Raise})$$

3.2 Towards Certification of Time Costs

Our overall research objective is to provide machine-certifiable bounds on resource usage. Having constructed our source-level cost model, we can use this to build a *provably* correct static analysis that is capable of deriving upper bounds on time usage. We propose to base our analysis on a type-and-effect system approach[1], where the type is as normal and the effect captures behavioural properties (here, time). We propose to obtain our low-level time information by *abstract interpretation* of binary-level programs, taking account of worst-case cache and pipelining behaviour. Thus, the analysis can be parameterized to the required architecture. By using a type-based *inference* algorithm, *verifiable* certificates of resource usage can also be generated independently for each program component, and these certificates may subsequently be combined in a compositional fashion, thereby ensuring overall integrity of resource usage. This framework for determining and composing certificates for resource-bounded computations in an incremental fashion will support reuse of both software components and analyses.

Note that while the cost model will correctly provide upper bound costs for arbitrary levels of Hume programs, it will not be possible to construct static analyses that are capable of predicting costs for *arbitrary* Hume programs. In particular, we may only be able to automatically predict costs for restricted recursive forms, e.g. those whose cost equations correspond to polynomial time. However, it is our expectation that the information provided by the static analysis may guide the programmer to produce algorithms for which costs can be provided.

3.3 Concrete Time Costs

In order to ground the cost model, we need to provide concrete values for each time cost, `Tmkint` etc. Table 2 shows the measured costs of executing each abstraction machine instruction on a 1.25GHz PowerPC G4 (Apple Macintosh Powerbook running MacOSX 10.4.4). Each cost is determined from the greatest time obtained from 1,000,000 executions of the corresponding abstract machine instruction. Since we are primarily interested in system responsiveness, and would anticipate executing deployed Hume programs on a dedicated platform if real-time constraints were an issue, we have chosen to measure CPU time rather than wall-clock time. However, we took care to run the measurements on a system that was running no other user activity (including the graphical operating system), and which had minimal system activity.

While these measurements do not therefore provide absolute guarantees, we therefore have a high degree of confidence in the accuracy of these measurements. In the near future, we intend to work with AbsInt GmbH to obtain precise measurements of worst-case cost, using abstract interpretation of the underlying machine code, based on an exact model of the processor architecture [10], including precise models of cache [11] and pipeline behaviours [23] for certain architectures used in embedded systems designs. Using this approach, we have already obtained promising measurements for a few abstract machine instructions.

`Tmkint`	=	$0.0458\mu s$
`Tmkvec(n)`	=	$0.007\mu s \times n + 0.0556\mu s$
`Tpushvar`	=	$0.110\mu s$
`Tiffalse`	=	$0.051\mu s$
`Tiftrue`	=	$0.051\mu s$
`Tgoto`	=	$0.0428\mu s$
`Tcall`	=	$0.068\mu s$
`Tcallprim()`	=	depends on the primitive
`Tcreatef`	=	$0.05\mu s$
`Treturn`	=	$0.122\mu s$
`Tslide`	=	$0.109\mu s$
`Tmatchrule`	=	$0.053\mu s$
`Tmatchedrule`	=	$0.039\mu s$

Table 2: Time costs for sample HAM instructions on a 1.25GHz PowerPC G4

Since the cost of `Tcallprim()` depends on the primitive operation (addition, subtraction etc.), and on the types of the operands, we are unable to give a single cost in Table 2. We have, however, measured costs for a number of common primitives.

4. ENCODING THE MEAN SHIFT ALGORITHM

The *mean shift* tracking algorithm [9] is a kernel-based method that is normally applied using a symmetric Epanechnikov kernel [8] within a pre-defined elliptical or rectangular window. The input is an image sequence, and the output is a track of the target in the image plane over the whole sequence. The idea is to *model* a region of an initial image by a probability density function that describes the first order statistics of that region. If the image sequence

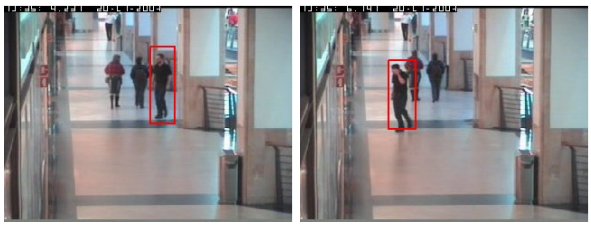

Figure 2: Example of CDT tracking

Version	Boxes	Functions	(utility)	Chars
C++	N/A	12	N/A	11067
Hume List	3	49	20	7411
Hume Vec	3	43	16	6457
Hume Box	13	25	9	11572

Table 3: Comparison of code sizes

is in colour then the usual choice is to employ a colour histogram. Then, the task is to find a *candidate* position for the target in a subsequent image by finding the minimum distance between the model and candidate histograms using an iterative procedure. We have improved the basic technique by using a chamfer distance transform (CDT) to define the kernel weights. Provided the CDT-kernel is representative of the target-background separation in the image, this leads to improved tracking through the sequenceThe algorithm can be summarised as follows:

```
Define target centroid, y0, in first frame
Apply segmentation
 (e.g.  using homogeneity criteria,
 background subtraction)
 to separate foreground (target) and background
Compute CDT-kernel
Form model histogram, q, in colour, space
Repeat
  Fetch next frame
  Segment window centred on y0
  Repeat
    Compute candidate histogram p(y0)
     in colour space using CDT-kernel
    Find next candidate location
    Compute Error, |y1 − y0|
  Until |y1 − y0| ≤ ϵ , a threshold
  Set |y1 − y0| for next frame
Until (end of sequence)
```

Figure 2 shows four frames from a sequence taken from the Caviar project [12] in which the CDT-kernel produces better results than the basic Epanechnikov method. We have implemented MatLab, C++ and three different Hume versions of the basic mean shift tracking code. In the first full-Hume version of the algorithm, all calculation is carried out using recursive functions, with potentially unbounded lists for the data structures. The second version is at the higher-order Template-Hume level, where, rather than allowing user-defined recursion, only pre-defined higher-order functions may be used in the style of FP [4] or the Bird-Meertens formalism [5, 24]. This vector-based implementation is very similar to the original list-based version. However, bounded vectors have replaced unbounded lists, and directly-recursive functions have been replaced by higher-order functions to construct and modify vectors. The third version is in a form very close to FSM-Hume: recursion is replaced by iteration over boxes and functions are only used to establish constants. This version is considerably more complicated than the first two versions, requiring substantially more boxes and associated wiring, but it is much more amenable to current costing technology. As shown in Table 3, the list- and vector-based Hume programs are significantly smaller than the C++ equivalent; the box-based Hume version is around the same size. Table 4 compares execution times for each of the three languages.

Language code	Matlab	Hume interpreter	Hume compiler	C++
Image I/O	0.832	4.099	0.446	0.487
Type convert	0.841	12.077	1.243	0.663
Max value	0.230	1.813	0.096	0.077
Computing kernel	0.661	9.991	0.514	0.430
	2.564	27.98	2.299	1.657

Table 4: Comparison of time (in sec)

5. A SAMPLE TIME ANALYSIS

We illustrate our approach by showing how an analysis can be constructed for `findNewCentre`, a key function from the *meanshift* algorithm. This function corresponds to the inner loop of the algorithm above that determines the final candidate position, y_1. This recursively determines convergence, and can be constructed using our rules. The analysis shown here has been produced by hand, but we intend to automate this in the next stage of the project. The definition of `findNewCentre` is:

```
findNewCentre centre dx olddx nloops frame Qu  =
 if
  dx==<<0,0>> || nloops>4 || addCoord dx olddx == <<0,0>>
 then centre
 else findNewCentre (addCoord centre dx)
         (computeDisplacement
           (updateWeights (updateModel
                           frame
                           (addCoord centre dx)
                           theKern)
              Qu frame (addCoord centre dx)) theDeriv)
        dx (nloops+1) frame Qu;
```

The call to the function `findNewCentre` has six concrete arguments. Each of these is evaluated, taking some time cost, and the call is then processed by passing each argument by reference on the stack, shown here as App .

$$\frac{Arg_1 \quad Arg_2 \quad Arg_3 \quad Arg_4 \quad Arg_5 \quad Arg_6 \qquad \mathcal{V}, \eta_4 \vdash^{t_6}_{t'_a} \textsc{App}(\texttt{findNewCentre}, [\ell_6, \ldots, \ell_1]) \rightsquigarrow \ell, \eta'}{\mathcal{V}, \eta_0 \vdash^{t_0}_{t_{call}} \texttt{findNewCentre ctr dx} \langle\langle 0, 1 \rangle\rangle \texttt{ 0 frame Qu} \rightsquigarrow \ell, \eta'}$$

Costs for each of the arguments can be similarly derived. Variables are simply looked up, constants (e.g. 0, 1) are

allocated in dynamic memory (representing specific cost in the abstract machine we have chosen to model), as are constructed values such as the vector $\langle\langle 0, 1\rangle\rangle$ representing the fourth argument.

$$Arg_1 \frac{\mathcal{V}(\texttt{Qu}) = \ell_1}{\mathcal{V}, \eta_0 \vdash^{t_0}_{t_1} \texttt{Qu} \rightsquigarrow \ell_1, \eta_0}$$

$$Arg_2 \frac{\mathcal{V}(\texttt{frame}) = \ell_2}{\mathcal{V}, \eta_0 \vdash^{t_1}_{t_2} \texttt{frame} \rightsquigarrow \ell_2, \eta_0}$$

$$Arg_3 \frac{\textsc{New}(\eta_0) = \ell_3 \qquad w_3 = (\mathsf{int}, 0) \qquad \eta_0[\ell_3 \mapsto w_3] = \eta_3}{\mathcal{V}, \eta_0 \vdash^{t_2}_{t_3} 0 \rightsquigarrow \ell_3, \eta_3}$$

$$Arg_4 \left\{ \frac{\begin{array}{c} \textsc{New}(\eta_3) = \ell'_4 \\ \dfrac{w'_4 = (\mathsf{int}, 1) \qquad \eta_3[\ell'_4 \mapsto w'_4] = \eta'_4}{\mathcal{V}, \eta_3 \vdash^{t_3}_{t'_4} 1 \rightsquigarrow \ell'_4, \eta'_4} \\ \textsc{New}(\eta'_4) = \ell''_4 \\ \dfrac{w''_4 = (\mathsf{int}, 0) \qquad \eta'_4[\ell''_4 \mapsto w''_4] = \eta''_4}{\mathcal{V}, \eta'_4 \vdash^{t'_4}_{t''_4} 0 \rightsquigarrow \ell''_4, \eta''_4} \\ \textsc{New}(\eta''_4) = \ell_4 \\ w_4 = (\mathsf{constr}_v, \ell'_4, \ell''_4) \qquad \eta''_4[\ell_4 \mapsto w_4] = \eta_4 \end{array}}{\mathcal{V}, \eta_3 \vdash^{t_3}_{t_4} \langle\langle 0, 1\rangle\rangle \rightsquigarrow \ell_4, \eta_4} \right.$$

$$Arg_5 \frac{\mathcal{V}(\texttt{dx}) = \ell_5}{\mathcal{V}, \eta_4 \vdash^{t_4}_{t_5} \texttt{dx} \rightsquigarrow \ell_5, \eta_4}$$

$$Arg_6 \frac{\mathcal{V}(\texttt{ctr}) = \ell_6}{\mathcal{V}, \eta_4 \vdash^{t_5}_{t_6} \texttt{ctr} \rightsquigarrow \ell_6, \eta_4}$$

Arguments are given to the App rule. Patterns are matched and the body of the function is evaluated.

$$\frac{\begin{array}{c} \eta_4 \vdash^{t_6 - \texttt{Tcreatef}}_{t_{pat}} [\texttt{ctr}, \texttt{dx}, \texttt{olddx}, \texttt{nloops}, \texttt{frame}, \texttt{Qu}], \left\{ \begin{array}{c} \mathcal{V}_{pat} = \mathcal{V}_1[\texttt{Qu} \mapsto \ell_1] \qquad \mathcal{A} = \mathcal{A}_1 \cup \{\ell_1\} \\ [\ell_6, \ldots, \ell_1], \\ \varnothing, \\ \emptyset \end{array} \right\} \triangleright^* \left\{ \begin{array}{c} [], \\ [], \\ \mathcal{V}_{pat}, \\ \mathcal{A} \end{array} \right. \\ \mathcal{V}_{pat}, \eta_4 \vdash^{t_{pat}}_{t_e} \texttt{if c then ctr else rec} \rightsquigarrow \ell, \eta' \end{array}}{\mathcal{V}, \eta_4 \vdash^{t_6}_{t_e - \texttt{Treturn}} \textsc{App}(\texttt{findNewCentre}, [\ell_6, \ldots, \ell_1]) \rightsquigarrow \ell, \eta'}$$

We must now consider the cost of the function definition. There are two cases representing the initial situation (the condition is computed as true) and the general recursive case (the condition is computed as false), respectively.

```
c=
 dx==<<0,0>> || nloops>4 || addCoord  dx  olddx == <<0,0>>

rec = findNewCentre (addCoord centre dx)
        (computeDisplacement
          (updateWeights  (updateModel
                            frame
                            (addCoord centre dx)
                            theKern)
             Qu frame (addCoord centre dx)) theDeriv)
        dx (nloops+1) frame Qu
```

$$\texttt{initial} \left\{ \frac{\begin{array}{c} \vdots \\ \dfrac{}{\mathcal{V}_{pat}, \eta_4 \vdash^{t_{pat}}_{t_{pat} - t_c} \texttt{c} \rightsquigarrow \ell_c, \eta_c} \\ \eta_c(\ell_c) = (\mathsf{bool}, \mathtt{tt}) \\ \mathcal{V}_{pat}(\texttt{ctr}) = \ell_6 \\ \dfrac{}{\mathcal{V}_{pat}, \eta_c \vdash^{t_{pat} - t_c - \texttt{Tiftrue}}_{t_{then}} \texttt{ctr} \rightsquigarrow \ell_6, \eta_c} \end{array}}{\mathcal{V}_{pat}, \eta_4 \vdash^{t_{pat}}_{t_{then}} \texttt{if c then ctr else rec} \rightsquigarrow \ell_6, \eta_c} \right.$$

$$\texttt{loop} \left\{ \frac{\begin{array}{c} \vdots \\ \dfrac{}{\mathcal{V}_{pat}, \eta_4 \vdash^{t_{pat}}_{t_{pat} - t_c} \texttt{c} \rightsquigarrow \ell_c, \eta_c} \\ \eta_c(\ell_c) = (\mathsf{bool}, \mathtt{ff}) \\ \vdots \\ \dfrac{}{\mathcal{V}_{pat}, \eta_c \vdash^{t_{pat} - t_c - \texttt{Tiffalse}}_{t_{else}} \texttt{rec} \rightsquigarrow \ell_{else}, \eta_{else}} \end{array}}{\mathcal{V}_{pat}, \eta \vdash^{t_{pat}}_{t_{false}} \texttt{if c then ctr else rec} \rightsquigarrow \ell_{else}, \eta_{else}} \right.$$

It is now necessary to determine values for each of the intermediate times based on the underlying machine. For the abstract machine we have chosen, these are:

$$\begin{array}{rclcrcl} t_1 & = & t_0 - \texttt{Tpushvar} & & t''_4 & = & t'_4 - \texttt{Tmkint} \\ t_2 & = & t_1 - \texttt{Tpushvar} & & t_4 & = & t''_4 - \texttt{Tmkvec(2)} \\ t_3 & = & t_2 - \texttt{Tmkint} & & t_5 & = & t_4 - \texttt{Tpushvar} \\ t'_4 & = & t_3 - \texttt{Tmkint} & & t_6 & = & t_5 - \texttt{Tpushvar} \end{array}$$

$$\begin{array}{rcl} t_{pat} & = & t_6 - \texttt{Tcreatef} - \texttt{Tmatchrule} \\ & & -6 \times \texttt{Tmatchvar} - \texttt{Tmatchedrule} \\ t_{else} & = & t_{pat} - t_c - \texttt{Tiffalse} - t_{rec} \\ t_{false} & = & t_{else} - \texttt{Tgoto} \\ t_{then} & = & t_{pat} - t_c - \texttt{Tiftrue} - \texttt{Tpushvar} \\ t_e & = & t_{false} \text{ OR } t_e = t_{then} \\ t'_a & = & t_e - \texttt{Treturn} \\ t_{call} & = & t'_a - \texttt{Tcall} - \texttt{Tslide} \end{array}$$

The cost of the initial base case is then:

$$\begin{array}{rcl} t_{init} & = & t_0 - t_{call} \\ & = & \texttt{Tcall} + 4 \times \texttt{Tpushvar} + 3 \times \texttt{Tmkint} + \texttt{Tmkvec(2)} \\ & & + \texttt{Tcreatef} + \texttt{Tmatchrule} + 6 \times \texttt{Tmatchvar} \\ & & + \texttt{Tmatchedrule} + t_c + \texttt{Tiftrue} + \texttt{Tpushvar} \\ & & + \texttt{Treturn} + \texttt{Tslide} \end{array}$$

and the cost of the general case ($t_e = t_{else}$) is:

$$\begin{array}{rcl} t_{loop} & = & \texttt{Tcall} + 4 \times \texttt{Tpushvar} + 3 \times \texttt{Tmkint} + \texttt{Tmkvec(2)} \\ & & + \texttt{Tcreatef} + \texttt{Tmatchrule} + 6 \times \texttt{Tmatchvar} \\ & & + \texttt{Tmatchedrule} + t_c + \texttt{Tiffalse} + t_{rec} \\ & & + \texttt{Tgoto} + \texttt{Treturn} + \texttt{Tslide} \end{array}$$

We can conclude that the cost of k iterations of the loop will therefore be $t_{init} + k \times t_{loop}$. In order to validate our results, we have compared the time that is predicted using our analysis with an actual hand-derived measurement (on 1.25GHz PowerPC G4). For our chosen data set of 129MB data (an actual scene), k has the value 5, and the measurements and predictions are as below:

Predicted t_{init} (with t_c measured)	10.5μs
Predicted t_{loop} (with t_{rec} measured)	293.8ms
Predicted total ($t_{init} + 5 \times t_{loop}$)	1,469ms
Measured total ($k = 5$)	1,463.6ms

Since we do not yet have a fully automatic analysis, in this paper, we have chosen to measure t_{rec} and t_c rather than calculating these values by hand (a long, tedious and rather unilluminating exercise). In the longer term, we would expect all such values to be predicted automatically.

In all cases, our measurements represent the upper bound of at least 100 executions. The predicted time over-estimates the measured execution time by 0.4%, in this case i.e. it provides a *safe and high-quality upper-bound* on execution cost. This confirms earlier results we have obtain on simpler program fragments, where we were able to consistently obtain upper bounds on actual execution costs.

Note that although, in this case, the solution is a simple linear function and can therefore use the sample simple solver technology as Hofmann and Jost [16, 17, 18, 22], in general, the cost function will be more complex, and powerful recurrence solving technology will therefore be required [32]. We are in the process of constructing an automatic analysis based on combining these techniques. Note also that the execution time cost is a function of the size of the input scene. Although we have not yet measured multiple different scenes in order to improve confidence in our result, this difference will affect only the absolute costs of t_{rec}/t_c and not the general result reported here.

6. RELATED WORK

Accurate time and space cost-modelling is an area of known difficulty for functional language designs [27]. Hume is thus, as far as we are aware, unique both in being a practical language based on strong automatic cost models, and in being specifically designed to allow straightforward space- and time-bounded implementation for hard real-time systems, those systems where tight real-time guarantees must be met. A number of functional languages have, however, looked at *soft* real-time issues [2, 33, 34], there has been work on using functional notations for hardware design (essentially at the HW-Hume level) [15, 7, 21], the Timber language includes *monadic* constructs for specifying strong real-time properties [25], and there has been much recent theoretical interest both in the problems associated with costing functional languages [27, 20, 6, 29, 30] and in bounding space/time usage [19, 28, 16, 34], including work on statically predicting heap and stack memory usage [31].

In their proposal for Embedded ML, Hughes and Pareto [19] have combined the earlier *sized type system* [20] with the notion of *region types* [28] to give bounded space and termination for a first-order strict functional language [19]. Their language is restricted in a number of ways: most notably in not supporting higher-order functions, and in requiring the programmer to specify detailed memory usage through type specifications. The practicality of such a system is correspondingly reduced. Burstall[6] proposed the use of an extended *ind case* notation in a functional context, to define inductive cases from inductively defined data types. While *ind case* enables static confirmation of termination, Burstall's examples suggest that considerable ingenuity is required to recast terminating functions based on a laxer syntax. Turner's *elementary strong functional programming* [29, 30] has similarly explored issues of guaranteed termination in a purely functional programming language. Turner's approach separates finite data structures such as tuples from potentially infinite structures such as streams. This allows the definition of functions that are guaranteed to be primitive recursive, but at a cost in additional programmer notation compared with the Hume approach shown here.

7. CONCLUSIONS AND FURTHER WORK

This paper has outlined our approach to constructing software with strong resource guarantees, introduced a mathematically-based resource model for time usage in a strict functional language and briefly explored its application to a realistic image processing algorithm written in the novel language Hume. Our results show that, in principle, good cost models can be constructed for such applications. We have demonstrated this for one realistic computation that forms the core of a vision algorithm. The work thus forms the basis for constructing formal certificates of bounds on time usage. It extends our earlier work by considering the more complex situation of time usage in close correpondance with a formal model of machine execution whose concrete costs can be strongly based using either measurement of primitive operations or by abstract interpreation of underlying machine instructions. We next plan to:

- construct formally verifiable automatic program analyses based on our cost models, thereby enhancing system integrity;
- use the advanced abstract interpretation tools developed by AbsInt GmbH [10] to provide bounded cost information for abstract machine operations, incorporating worst-case information on cache and pipeline behaviours;
- produce program logics based on earlier work on mobile code [3],that will allow these time costs to be extracted as verifiable certificates of bounded execution time;
- develop new high-level algorithms for ego-motion computation, target tracking and environmental mapping for autonomous vehicle control, which meet strong time and space resource requirements, and which use our language technology;
- extend our models and analyses to incorporate components written using more conventional approaches such as Matlab-generated C, subject to possible limitations on source code forms.

8. ACKNOWLEDGEMENTS

The work reported in this paper was funded by the Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre established by the UK Ministry of Defence. Iain Wallace was supported by a Nuffield Foundation Undergraduate Research Bursary. We would also like to thank our collaborators in the EU EmBounded project, in particular Robert Pointon and Steffen Jost, for their comments and assistance.

9. REFERENCES

[1] T. Amtoft, F. Nielson, and H. Nielson. *Type and Effect Systems: Behaviours for Concurrency.* Imperial College Press, 1999.

[2] J. Armstrong, S. Virding, and M. Williams. *Concurrent Prog. in Erlang*. Prentice-Hall, 1993.

[3] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A Resource-aware Program Logic for Grail. In *Proc. ESOP'04 — European Symposium on Programming*, 2004.

[4] J. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. In *Comm. ACM*, **21(8)**: 613–641, 1978.

[5] R. Bird. Constructive Functional Programming. In *STOP Summer School on Constructive Algorithmics, Abeland*, 1989.

[6] R. Burstall. Inductively Defined Functions in Functional Programming Languages. Technical Report ECS-LFCS-87-25, 1987.

[7] K. Claessen and M. Sheeran. A Tutorial on Lava: a Hardware Desc. and Verification System. Aug. 2000.

[8] D. Comaniciu and P. Meer. Mean Shift: A Robust Approach toward Feature Space Analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619, 2002.

[9] D. Comaniciu, V. Ramesh, and P. Meer. Kernel-Based Object Tracking. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 25(5):564–575, 2003.

[10] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. EMSOFT 2001, 1st Workshop on Embedded Software*, Springer-Verlag LNCS 2211, pages 469–485. 2001.

[11] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache Behavior Prediction by Abstract Interpretation. *Science of Comp. Programming*, 35(2):163–189, 1999.

[12] R. Fisher. CAVIAR: Context Aware Vision using Image-based Active Recognition, `http://homepages.inf.ed.ac.uk/rbf/caviar/`, 2005.

[13] 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)*,Springer-Verlag LNCS, 2003.

[14] K. Hammond and G. Michaelson. Predictable Space Behaviour in FSM-Hume. In *Proc. Implementation of Functional Langs.(IFL '02), Madrid, Spain*, Springer-Verlag LNCS 2670, 2003.

[15] J. Hawkins and A. Abdallah. Behavioural Synthesis of a Parallel Hardware JPEG Decoder from a Functitonal Specification. In *Proc. EuroPar 2002*, Aug. 2002.

[16] M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[17] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proc. POPL 2003 — 2003 ACM Symp. on Principles of Prog. Langs.*, pages 185–197. ACM, 2003.

[18] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *Proc. ESOP 2006 – 2006 European Symp. on Programming*, pages 22–37, 2006.

[19] R. Hughes and L. Pareto. Recursion and Dynamic Data Structures in Bounded Space: Towards Embedded ML Programming. In *Proc. 1999 ACM Intl. Conf. on Functional Programming (ICFP '99)*, pages 70–81, 1999.

[20] R. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proc. POPL'96 — 1996 ACM Symp. on Principles of Programming Languages*, Jan. 1996.

[21] J. L. J. Matthews and B. Cook. Microprocessor Specification in Hawk. In *Proc. International Conference on Computer Science*, 1998.

[22] S. Jost. *Linearly Bounded Heap Space Analysis, Ludwig-Maximilians-Universität, München, Germany (in preparation)*. PhD thesis, 2006.

[23] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *Proc. 9th International Static Analysis Symposium SAS 2002*, Springer-Verlag LNCS 2477, pages 294–309. 2002.

[24] S.-C. Mu and R. S. Bird. Rebuilding a Tree from its Traversals: A Case Study of Program Inversion. In *Proc. Australian Conf. on Prog. Lang. and Systems (APLAS 2003)*, pages 265–282, 2003.

[25] J. Nordlander, M. Carlsson, and M. Jones. Programming with Time-Constrained Reactions (unpublished report). `http://www.cse.ogi.edu/pacsoft/projects/Timber/publications.htm`. 2006.

[26] J. Nyström, P. Trinder, and D. King. High-level distribution for the rapid production of robust telecoms software: comparing c++ and erlang. *Concurrency and Computation: Practice & Experience.*, 18, December 2006.

[27] A. Rebón Portillo, K. Hammond, H.-W. Loidl, and P. Vasconcelos. A Sized Time System for a Parallel Functional Language (Revised). In *Proc. Implementation of Functional Langs.(IFL '02), Madrid, Spain*, Springer-Verlag LNCS 2670, 2003.

[28] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 Feb. 1997.

[29] D. Turner. Elementary Strong Functional Programming. In *Proc. 1995 Symp. on Functl. Prog. Langs. in Education — FPLE '95*, LNCS. Springer-Verlag, Dec. 1995.

[30] D. Turner. Total Functional Programming. *Journal of Universal Computing*, 10(7):751–768, 2004.

[31] L. Unnikrishnan, S. Stoller, and Y. Liu. Automatic Accurate Stack Space and Heap Space Analysis for High-Level Languages. Tech. Report 538, Comp. Sci. Dept., Indiana University, Apr. 2000.

[32] P. Vasconcelos and K. Hammond. Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proc. Implementation of Functional Languages (IFL 2003)*, 2004.

[33] M. Wallace and C. Runciman. Extending a Functional Programming System for Embedded Applications. *Software: Practice & Experience*, 25(1), Jan. 1995.

[34] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *Intl. Conf. on Functional Programming (ICFP '01)*, Sept. 2001.