

Towards Formally Verifiable Resource Bounds for Real-Time Embedded Systems

Kevin Hammond* Christian Ferdinand † Reinhold Heckmann †
Roy Dyckhoff* Martin Hofmann ‡ Steffen Jost* Hans-Wolfgang Loidl ‡
Greg Michaelson § Robert Pointon § Norman Scaife ¶ Jocelyn Sérot ¶
Andy Wallace §

Abstract

This paper describes ongoing work aimed at the construction of formal cost models and analyses that are capable of producing verifiable guarantees of resource usage (space, time and ultimately power consumption) in the context of real-time embedded systems. Our work is conducted in terms of the domain-specific language Hume, a language that combines *functional programming* for computations with *finite-state automata* for specifying reactive systems. We describe an approach in which high-level information derived from source-code analysis can be combined with worst-case execution time information obtained from abstract interpretation of low-level binary code. This abstract interpretation on the machine-code level is capable of dealing with complex architectural effects including cache and pipeline properties in an accurate way. It has been applied to several large-scale commercial safety-critical systems, including the flight control system for the Airbus A380.

1 Introduction

Accurate modelling and verification of resource usage is a major open problem in real-time embedded systems. In contrast to conventional software, typical firmware and software for embedded systems impose very strong requirements on both space and time usage. This reflects the cost sensitivity of typical embedded systems designs: with high production volumes, small differences in unit hardware cost (recurring expenses) lead to large variations in profit. At the same time software production costs (non-recurring engineering expenses) must be kept under control, and time-to-market must be minimised.

1.1 Challenges for Real-Time Embedded Systems Software

Historically, much embedded systems firmware and software was written for specific hardware using native assembly code. Rapid increases in software complexity improvement means that there has been a transition [12] to the use of C/C++. Despite this, 80% of all embedded systems are delivered

*School of Computer Science, University of St Andrews, North Haugh, St Andrews, Scotland, KY16 9SX. **email:** {kh,rd,jost}@dcs.st-and.ac.uk.

†AbsInt GmbH, Saarbrücken, Germany. **email:** {cf,heckmann}@absint.com

‡Ludwig-Maximilians Universität, München.

email: {mhofmann,hwloidl}@informatik.uni-muenchen.de

§Depts. of Comp. Sci. and Elec. Eng., Heriot-Watt University, Riccarton, Edinburgh, Scotland. **email:** {G.Michaelson, A.M.Wallace}@hw.ac.uk

¶LASMEA, Université Blaise-Pascal, Clermont-Ferrand, France
email: Jocelyn.SEROT@univ-bpclermont.fr

This work has been supported by EU Framework VI grant IST-2004-510255 and by EPSRC Grant EPC/0001346.

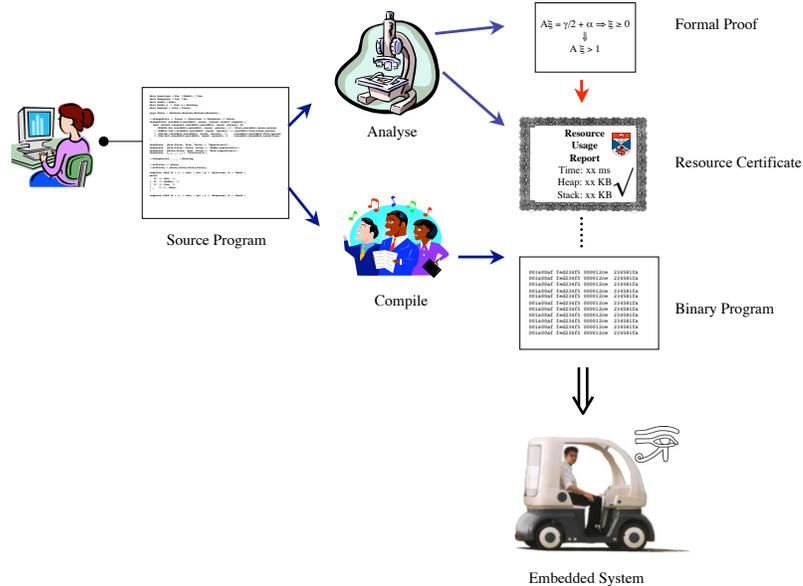


Figure 1: A Vision for Resource Certification

late [15], and massive amounts are spent on bug fixes: according to Klocwork, Nortel, for example, spends \$14,000 correcting each bug found once a system is deployed.

Many of the faults in C/C++ programs are caused by poor programmer management of memory resources [42], exacerbated by the programming being at a relatively low level of abstraction. There is thus pressure to reduce software engineering costs by using modern automatic memory management techniques (in which we include both static techniques and dynamic techniques such as garbage collection), by exploiting very high-level programming notations and even by automatic code generation from, e.g., UML models [26]. However, the difficulty of determining accurate bounds on space and time usage by manual inspection or by standard timing analyses increases with the use of high-level programming abstractions. Determination of such bounds is especially vital in the construction of *dependable* embedded systems software.

We envisage (Figure 1) future real-time embedded system software engineers programming in very high-level *next generation* programming notations, whilst being supported by fully automatic tools for analysing time and space behaviour. These tools will provide *automatically verifiable certificates* of resource usage that will allow software to be built in a modular and compositional way, whilst providing strong guarantees of overall system cost. In this way, we will progress towards the strong standards of mathematically-based engineering that are present in other, more mature, industries, whilst simultaneously enhancing engineering productivity and reducing time-to-market for embedded systems.

Our vision will be achieved by constructing tools that will *automatically* assign good upper bounds on resource usage (including time and space behaviour) for sophisticated programs that use advanced data structures, recursion, strong polymorphic typing and automatic memory management, and that will provide *automatically verifiable* certificates of such usage. Whilst such bounds are of general interest, they are most pertinent in the embedded systems arena, where hard real-time guarantees must frequently be provided and memory is highly restricted.

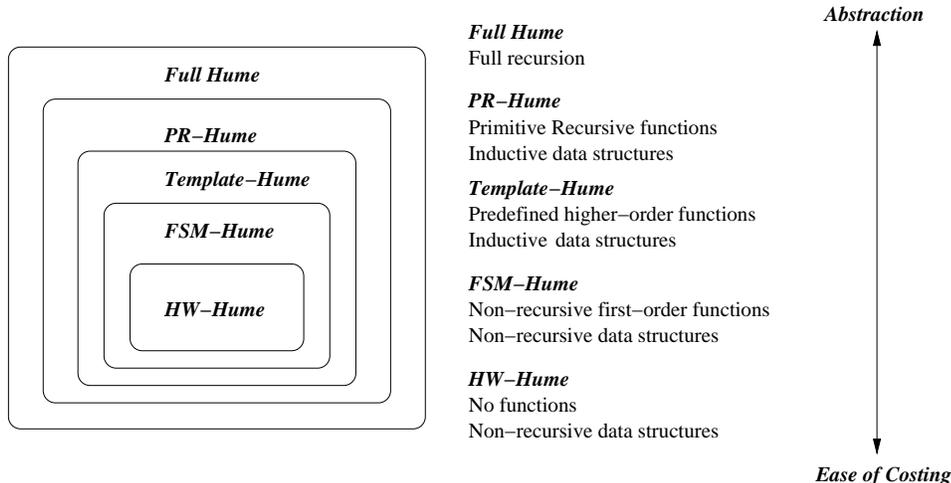


Figure 2: Expression Levels in the Hume Language

1.2 The EmBounded Project

A major problem with embedded systems is that the available system resources (processor, computer memory, power) are necessarily restricted due to cost and other considerations. If it were possible to determine strong bounds on the use of such resources, then there would be significant benefits in terms of manufacturing cost, reliability and performance. However, determining such bounds automatically is a hard problem, and doing so manually is becoming impracticable as embedded software increases in complexity.

The EU Framework VI EmBounded project (IST-2004-510255), led by Dr Kevin Hammond aims to research this problem. Our vision is one where certificates of the bounds on resource usage can be obtained from a source program through automatic analysis independently of the usual software compilation process. These certificates may be verified using formal proof techniques derived from mathematics and logic.

The EmBounded project involves academics and other scientists in France, Germany and the UK, building on world-leading strengths in programming language design, program analysis, and embedded applications. Researchers in Theoretical Computer Science at Ludwig-Maximilians-Universität (München, Germany) and St Andrews will develop new theoretical models of bounded resource usage and the associated automatic analyses. Programming language designers and implementers at St Andrews and Heriot-Watt will capture the requirements of these analyses within a new programming language, Hume. Researchers at AbsInt GmbH (Saarbrücken, Germany), will relate the resource models to actual computers, and researchers at the Laboratoire des Sciences et Matériaux d'Électronique – LASMEA (Clermont-Ferrand, France), will apply the technology to the sensor and control systems used in the CyCab autonomous vehicle: a self-controlled electric car resembling a golf buggy, and capable of speeds up to 30 km/h (<http://www.robosoft.fr>).

2 The Hume Language

Our work is undertaken in the context of Hume [16, 18], a functionally-based domain-specific high-level programming language for real-time embedded systems. Hume is designed as a layered language where the *coordination layer* is used to construct reactive systems using a finite-state-automata based notation; while the *expression layer* is used to structure computations using a strict purely functional rule-based notation that maps patterns to expressions. The coordination layer expresses reactive Hume programs as a static system of interconnecting *boxes*. If each box has bounded space cost internally, it follows that the system as a whole also has bounded space cost.

Similarly, if each box has bounded time cost, a simple schedulability analysis can be used to determine reaction times to specific inputs, rates of reaction and other important real-time properties.

Expressions can be classified according to a number of levels (Figure 2), where lower levels lose abstraction/expressibility, but gain in terms of the properties that can be inferred. For example, the bounds on costs inferred for primitive recursive functions (PR-Hume) will usually be less accurate than those for non-recursive programs, while cost inference for Full Hume programs is undecidable in general (and we therefore restrict our attention to PR-Hume and below). Previous papers have considered the Hume language design in the general context of programming languages for real-time systems [18, 17], described a heap and stack analysis for FSM-Hume [19], and considered the relationship of Hume with classical finite-state machines [35]. The main contributions of this paper are the provision of a new *cost-potential* based model for stack and heap usage including higher-order functions, recursion and exceptions, and an outline of the design for a complete analysis of worst-case execution time for Hume.

2.1 Finite State Automata

Finite state automata provide a basis for constructing simple state-changing systems. They may also be used to give a natural model of concurrency. Finite state automata comprise a set of linked states, with transitions showing the changes from one state to another based on the inputs that are received. Because pure finite-state-automata are so simple, there is a natural fit between finite-state-automata and hardware, and it is easy to show that such automata have bounded time and space costs. Some low-level programming languages, such as Esterel [6] also use an essentially pure finite-state approach, and mechanistic systems such as lexers and parsers are also commonly automaton-based. The primary deficiencies of finite state approaches are:

- there may be a huge number of states for even fairly simple software programs;
- it may be necessary to decompose programming problems into very low-level abstractions;
- there are theoretical limitations on the classes of problem that can be solved using automata.

While each state may be simple in itself, the explosion in the number of states means even small, simple programs can be too complex to understand easily. Moreover, it can be cumbersome to write simple functions and other operations as automata. Hume attempts to systematically address these objections as follows:

- finite-state automata are used to structure concurrency only – computations are written using conventional programming notations;
- high-level programming notations are used to collapse complex sets into a few manageable automata;
- combining high-level programming notations with automata greatly extends the classes of problem that can be solved to cover all *computable* problems.

In Hume, programs are formed from concurrent *boxes*, which respond to inputs and produce outputs on one or more *wires*. Computations within boxes are described using normal high-level programming notations rather than as automata. While there is a broad analogy with the use of high-level *objects* as concurrent agents, and object-based designs can thus be exploited at the high-level, the analogy should not be stretched too far – boxes are much more structured than objects, in particular in restricting communication patterns, in relating inputs directly to outputs, and in providing a static rather than dynamic process network. This discipline allows us to automate testing, to demonstrate deadlock-freedom using an automatic analysis, and to enforce strong bounds on program costs.

2.2 Functional Programming

Purely functional programming provides a good basis for constructing software with excellent formal properties. Because functional programs are both *declarative* and deterministic, they are much easier to reason about using mathematically-derived techniques than either object-based or imperative approaches [28]. In fact, many advanced compiler optimisations work on an internal representation that is purely functional, and compilers can go to great lengths to isolate parts that are not purely functional, so that they can take advantage of these techniques. However, to obtain these advantages:

- a) programmers must be trained to exploit high-level functional abstractions, which some programmers find difficult;
- b) there may be a poor match between program and machine implementation, making it difficult to construct software that must access low-level features; and
- c) performance can be significantly worse than the best imperative implementations (though performance may be better than say C++ or even Fortran in some cases [?]).

Hume attempts to systematically address these objections by:

- a) finite-state automata are a natural way to decompose concurrent programs, and provide analogies to the higher levels of object-based program decomposition, without the strictures of overly low-level objects;
- b) state changes are made explicit through finite-state automata, and explicit operating system interactions;
- c) it is possible to provide a straightforward translation from Hume source to the corresponding machine code, whether direct or through an intermediate abstract machine; and
- d) Hume has been designed to allow the best compiler optimisations to be exploited – hindrances to compiler optimisation have been designed out as far as possible – time performance is roughly 10 times that for Sun’s embedded KVM, and dynamic space usage is both guaranteed to be bounded and a fraction of that required by Java or C++.

The combination of finite state automata with functional programming therefore gives a powerful programming basis without sacrificing crucial low-level capabilities.

2.3 Simple Hume Example: a Vending Machine

```
type Int = int 32;

data Coins = Nickel | Dime;
data Drinks = Coffee | Tea;
data Buttons = BCoffee | BTea | BCancel;

-- coffee vending box

vend drink cost v = if v >= cost then (drink, v-cost, *) else ( *, v, * );

box coffee
in ( coin :: Coins, button :: Buttons, value :: Int )
out ( drink :: Drinks, value' :: Int, return :: Int )
```

```

match
  ( Nickel, *,      v ) -> ( *, v + 5, * )
| ( Dime,  *,      v ) -> ( *, v + 10, * )
| ( *,    BCoffee, v ) -> vend Coffee 10 v
| ( *,    BTea,    v ) -> vend Tea 5 v
| ( *,    BCancel, v ) -> ( *, 0, v )
;

showdrink Coffee = "Coffee";
showdrink Tea = "Tea";

wire inp      ( stdin ) ( coffee.coin, coffee.button );
wire coffee ( inp.coin, inp.button, coffee.value' initially 0 ) ( outp.drink, coffee.value, outp.ret );
wire outp   ( coffee.drink, coffee.return ) ( stdout );

```

3 Program Analyses for Real-Time Embedded Systems

Static determination of *worst-case execution time* (WCET) in real-time systems is an essential part of the analyses of over-all response time and of quality of service [40]. However, WCET analysis is a challenging issue, as the complexity of interaction between the software and hardware system components often results in very pessimistic WCET estimates. For modern architectures such as the Motorola PPC755, for example, WCET prediction based on simple weighted instruction counts may result in an over-estimate of time usage by a factor of 250. Obtaining high-quality WCET results is important to avoid seriously over-engineering real-time embedded systems, which would result in considerable and unnecessary hardware costs for the large production runs that are often required.

Three competing technologies can be used for worst-case execution time analysis: *experimental* (or testing-based) approaches, *probabilistic measurement* and *static analysis*. Experimental approaches determine worst-case execution costs by (repeated and careful) measurement of real executions, using either software or hardware monitoring. However, they cannot guarantee upper bounds on execution cost. Probabilistic approaches similarly do not provide absolute guaranteed upper bounds, but are cheap to construct, deliver more accurate costs than simple experimental approaches [3, 4].

Abstract Interpretation for Accurate WCET Analysis Motivated by the problems of measurement-based methods for WCET estimation, AbsInt GmbH has investigated a new approach based on static program analysis [33, 21]. The approach relies on the computation of abstract cache and pipeline states for every program point and execution context using *abstract interpretation*. These abstract states provide safe approximations for all possible concrete cache and pipeline states, and provide the basis for an accurate timing of hardware instructions, which leads to safe and precise WCET calculations that are valid for all executions of the application.

Static Analysis of Memory Bounds Memory management is another important issue in real-time and/or embedded systems with their focus on restricted memory settings. Some languages provide automatic dynamic memory management without strong guarantees on time performance (e.g. Java [37]), whilst others rely on more predictable but error-prone explicit memory management (e.g. C, C++, RTSj or Ada). One recent approach [11] is to exploit memory *regions* for some or all allocations and to combine annotations with automatic inference. Such approaches do not, however, provide real-time guarantees, and typically require manual intervention in the allocation process. Moreover, static region analysis can be overly pessimistic [11] for long-lived allocations.

Regardless of the memory management method, there is a strong need for static guarantees of memory utilisation bounds.

4 A Stack and Heap Cost Model for Hume

We can now define a cost model for stack and heap usage for (PR-)Hume derived from the actual execution costs that apply in the prototype Hume Abstract Machine and based on work from Jost’s forthcoming PhD thesis [32]. As part of our work, we have defined models for both coordination and expression layers, but for brevity and clarity we present only the rules for expressions in this paper. We have already produced a prototype implementation of this cost model based on a sized type system [30] up to FSM-Hume and are now working on the extension to the analysis to include higher-order functions and primitive recursion, as described here, and to combine our work on sized type based analysis [49] with the Hofmann and Jost linear typing approach [25].

The statement $\mathcal{V}, \eta \stackrel{p}{p'} \stackrel{m}{m'} e \rightsquigarrow \ell, \eta'$ may be read as follows: expression e evaluates under the configuration \mathcal{V}, η in a finite number of steps to a result value stored at location ℓ in heap η' , provided that there were p stack and m heap units available before computation. Furthermore, at least p' stack and m' heap units are unused after the evaluation is finished.

The symbol $\stackrel{p}{p'} \stackrel{m}{m'}$ actually defines a family of rules, each rule of the family having p, p', m, m' instantiated, provided that all premises are met, in particular the implicit side condition $p, p', m, m' \in \mathbb{N}$. Hence the rule

$$\frac{}{\mathcal{V}, \eta \stackrel{p'+1}{p'} \stackrel{m}{m-2} e \rightsquigarrow \ell, \eta'}$$

is to be interpreted as an abbreviation of the rule

$$\frac{p = p' + 1 \quad m = m' + 2 \quad p \in \mathbb{N} \quad p' \in \mathbb{N} \quad m \in \mathbb{N} \quad m' \in \mathbb{N}}{\mathcal{V}, \eta \stackrel{p}{p'} \stackrel{m}{m'} e \rightsquigarrow \ell, \eta'}$$

Constants $\zeta_F, \zeta_L, \zeta_H \in \mathbb{N}$ represent the stack space required to hold the frames of a function call header, local definition frame header and case frame header respectively.

4.1 Properties

From this semantics, it is possible to derive a number of behavioural properties. The most important of these are that the cost model correctly captures the potential change in heap usage and that the result of execution is always left as an extra value on the stack. The proofs follow straightforwardly from the semantic definitions.

These properties are verified against the operational model of the Hume Abstract Machine (HAM) behaviour via a formal translation relating Hume source to HAM abstract machine code. The HAM operational model includes a formal description of the machine implementation of the stack and heap using by the abstract machine. Both the operational model and the translation have been constructed and we are now in the process of formulating the equivalence proof relating the Hume and HAM levels.

5 Worst Case Execution Timing Analysis using Abstract Interpretation

Our objective as part of the EU Framework VI EmBounded Project (IST-510255) is to develop a combined high- and low- level analysis for worst-case execution time. We will achieve this by extending the stack and heap cost model presented above with the addition of parameters representing

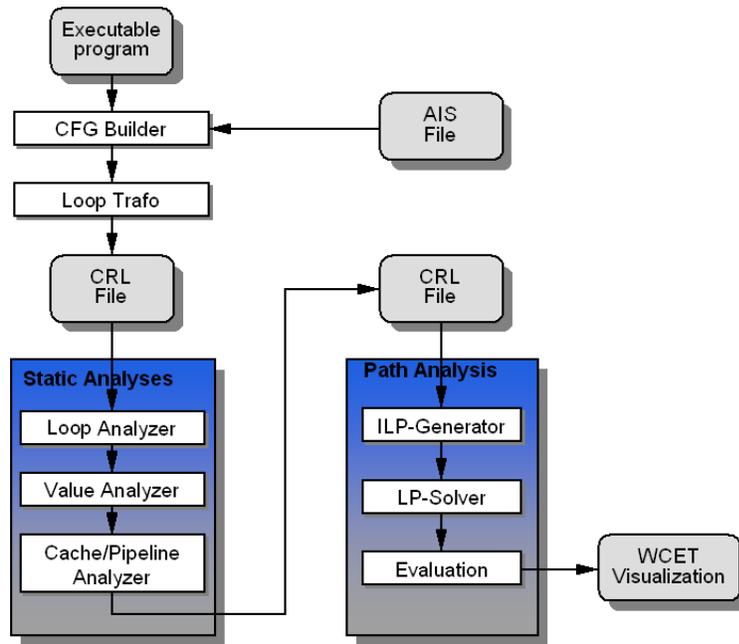


Figure 3: Phases of WCET computation

actual timing costs. Our ultimate objective is to permit the production of accurate worst-case cost information from source level programs.

The AbsInt **aiT** tool (described below) uses abstract interpretation to efficiently compute a safe approximation for all possible cache and pipeline states that can occur at a given program point. These results can be combined with ILP (Integer Linear Programming) techniques to safely predict the worst-case execution time and a corresponding worst-case execution path.

The AbsInt analysis works at a code snippet level, analysing imperative C-style code snippets to derive safe upper bounds on the worst case time behaviour. Whilst the AbsInt analysis works at a level that is more abstract than simple basic blocks, providing analyses for loops, conditionals and non-recursive subroutines, it is not presently capable of managing the complex forms of recursion which occur in functional languages such as our own PR-Hume, Haskell or SML. We are thus motivated to link the two levels of analysis, combining information on recursion bounds and other high-level constructs from the Hume source analysis with the low-level worst-case execution time analysis from the AbsInt analysis.

5.1 Phases of WCET Computation

In AbsInt’s approach [13] the WCET of a program task is determined in several phases (see Figure 3):

- **CFG Building** decodes, i.e. identifies instructions, and reconstructs the control-flow graph (CFG) from an executable binary program;
- **Value Analysis** computes address ranges for instructions accessing memory;
- **Cache Analysis** classifies memory references as cache misses or hits [14];
- **Pipeline Analysis** predicts the behavior of the program on the processor pipeline [33];

- **Path Analysis** determines a worst-case execution path of the program [43].

The cache analysis phase uses the results of the value analysis phase to predict the behaviour of the (data) cache based on the range of values that can occur in the program. The results of the cache analysis are then used within the pipeline analysis to allow prediction of those pipeline stalls that may be due to cache misses. The combined results of the cache and pipeline analyses are used to compute the execution times of specific program paths. By separating the WCET determination into several phases, it becomes possible to use different analysis methods that are tailored to the specific subtasks. Value analysis, cache analysis, and pipeline analysis are all implemented using abstract interpretation [10], a semantics-based method for static program analysis. Integer linear programming is then used for the final path analysis phase.

5.2 aiT – WCET Analyzers

The techniques described above have been incorporated into AbsInt’s **aiT** WCET analyser tools, that are in commercial use in several organisations. The input to these tools is the executable program to be analysed, user annotations describing the targets of any indirect jumps and calls that are not resolved by the automatic analysis, user annotations describing the maximal iteration counts of loops that are not determined by the automatic loop bound analysis, a description of the (external) memories and buses (i.e. a list of memory areas with minimal and maximal access times), and a task to be analysed (identified by a start address).

5.3 Linking the High- and Low-Level Analyses

In order to link the two levels of analysis, we must extend our stack and heap cost model for Hume to include timing information. The top-level description is a straightforward extension to our previous form: $\Sigma; \mathcal{V}, \eta \mid_p^{\mathcal{P}} \mid_m^{\mathcal{M}} \mid_{t'}^{\mathcal{T}} e \rightsquigarrow \ell, \eta'$, where t and t' are time potentials. Each rule in the model must now be adapted to include time potential which will vary monotonically in a similar way to the heap potential. This time potential must be verified against the actual times for execution on the Hume Abstract Machine using information obtained from the **aiT** tool. In this way, we will have constructed a complete time model and analysis from Hume source to actual machine code.

Pragmatically, in order to obtain timing information from the **aiT** tool, our high level analysis must be adapted to output information on the limits on recursion bounds and other high-level constraints derived from the program source that can be fed to the **aiT** tool using its native system specification language (**aiS**). This information must be provided in terms of the compiled executable code that has been produced from the Hume source rather than directly from the source itself. It will therefore also be necessary to provide details of the compilation process in an appropriate form.

6 Related Work

6.1 Functional Languages for Soft Real-Time Programming

Accurate time and space cost-modelling is an area of known difficulty for functional language designs [39]. 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 [1, 50, 51], there has been work on using functional notations for hardware design (essentially at the HW-Hume level) [20, 9, 31], the Timber language includes *monadic* constructs for specifying strong real-time properties [38], and there has been much recent theoretical interest both in the problems associated with costing functional languages [39, 30, 8, 46, 47] and in bounding space/time usage [29, 45, 23, 51], including work on statically predicting heap and stack memory usage [48].

The most widely used soft real-time functional language is the impure, strict language Erlang [1], a concurrent language with a similar design to Concurrent ML [41]. Erlang has been used by Ericsson to construct a number of successful telecommunications applications in the telephony sector [5], including a real-time database, Mnesia [52]. Erlang is concurrent, with a lightweight notion of a process. Unlike Hume boxes, Erlang processes are constructed using explicit spawn operations, with communication occurring through explicit *send* and *receive* operations to nominated processes.

6.2 Functional Languages imposing Syntactic Restrictions

Other than our own work [39, 49], we are aware of three main studies of formally bounded time and space behaviour in a functional setting [8, 29, 46]. All three approaches are based on restricted language constructs to ensure that bounds can be placed on time/space usage. In their proposal for Embedded ML, Hughes and Pareto [29] have combined the earlier *sized type system* [30] with the notion of *region types* [45] to give bounded space and termination for a first-order strict functional language [29]. 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[8] 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* [46, 47] 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.

6.3 Other Approaches to Bounding Space Usage

Compile-time garbage collection techniques attempt to eliminate some or all heap-based memory allocation through strong static means. One approach that has recently found favour is the use of region types [45]. Such types allow memory cells to be tagged with an allocation *region*, whose scope can be determined statically. When the region is no longer required, all memory associated with that region may be freed without invoking a garbage collector. In non-recursive contexts, the memory may be allocated statically and freed following the last use of any variable that is allocated in the region. In a recursive context, this heap-based allocation can be replaced by (possibly unbounded) stack-based allocation.

Hofmann’s linearly-typed functional programming language LFPL [22] uses linear types to determine resource usage patterns. So-called *diamond* resource types are used to count constructors. First-order LFPL definitions can be computed in bounded space, even in the presence of general recursion. Hofmann has recently considered the extension of LFPL to higher-order functions with reference to *non size-increasing* recursive definitions on lists [24], where the size of all intermediate computations is bounded by the size of the inputs. Where definitions are restricted to primitive recursion only, this then guarantees polynomial size complexity. Unfortunately, for arbitrary higher-order functions, the cost of introducing closures means that an unbounded stack is required.

Finally, Camelot and Grail [34] use a *proof carrying code* approach that allows formal properties of resource usage to be expressed in the form of easily checked certificates. Camelot is a resource-aware functional programming language that can be compiled to a subset of JVM bytecodes; Grail is a functional abstraction over these bytecodes. This abstraction possesses a formal operational semantics that allows the construction of a program logic capable of capturing program behaviours such as time and space usage [2]. The objective of the work is to synthesise proofs of resource bounds in the Isabelle theorem prover, and to attach these proofs to mobile code in the form of

more easily verifiable proof derivations. In this way the recipient of a piece of mobile code can cheaply and easily verify its resource requirements.

7 Conclusions and Future Work

In this paper we have introduced the Hume language and shown how a cost model can be constructed that is capable of exposing stack and heap cost information for higher-order, primitive recursive expressions. We have also outlined how our work can be extended in order to synthesise worst-case memory usage and execution time using a combination of source- and binary-based analysis.

Our work is formally based and motivated: we aim to construct formal models of behaviour at source program and abstract machine levels (as exemplified here by the cost model for stack and heap usage); have provided elsewhere a formal translation between these levels; and will synthesise actual worst-case execution time costs using abstract interpretation of binary programs. The abstract interpretation developed by AbsInt, and which we intend to use for Hume, comprises a complete formal model of a processor architecture including cache and pipeline effects. This analysis is currently undergoing a process of formal certification for use in the flight control systems of the Airbus A380, and has also been applied to other safety-critical systems [44].

The approach we have outlined here will automatically synthesise cost information from program source, but supports only limited provision of programmer information. In a recent paper [7], we have developed a *dependently-typed* framework which is capable of expressing dynamic execution costs through the type system. A key feature of a dependently typed setting is that it is possible to express more complex properties of programs than the usual simply typed frameworks in use in languages such as Standard ML [36] or Haskell [27]. In fact, computation is possible at the type level, and it is also possible to expose proof requirements that must be satisfied. In this way it is possible to exploit information that may be possessed by the programmer in order to direct cost analysis and the construction of the associated proofs. We anticipate that we will be able to construct a hybrid type checking/synthesis system that will possess benefits of both approaches: flexibility through checking of dependent types and simplicity through synthesis of most resource bounds.

References

- [1] J. Armstrong, S.R. Viriding, and M.C. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.
- [2] D. Aspinall, L. Beringer, M. Hofmann, and H.-W. Loidl. A resource-aware program logic for a jvm-like language. In *Trends in Functional Programming, Volume 4*. Intellect, 2004.
- [3] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *Proc. 12th Euromicro International Conference on Real-Time Systems*, Stockholm, June 2000.
- [4] G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS 2002)*, Austin, TX. (USA), December 2002.
- [5] S. Blau and J. Rooth. AXD-301: a New Generation ATM Switching System. *Ericsson Review*, 1, 1998.
- [6] F. Boussinot and R. de Simone. The Esterel Language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

- [7] Edwin Brady and Kevin Hammond. A Dependently Typed Framework for Static Analysis of Program Execution Costs. In *Proc. Implementation and Applications of Functional Language, 2005*, Dublin, September 2005.
- [8] R. Burstall. Inductively Defined Functions in Functional Programming Languages. Technical Report ECS-LFCS-87-25, Univ. of Edinburgh, April 1987.
- [9] K. Claessen and M. Sheeran. A Tutorial on Lava: a Hardware Description and Verification System. August 2000.
- [10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
- [11] M. Deters and R.K. Cytron. Automated Discovery of Scoped Memory Regions for Real-Time Java. In *Proc. ACM Intl. Symp. on Memory Management, Berlin, Germany*, pages 132–141, June 2002.
- [12] Embedded.com. Poll: What language do you use for embedded work? <http://www.embedded.com/pollArchive/?surveyno=2228>, 2003.
- [13] 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, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer-Verlag, 2001.
- [14] Christian Ferninand. *Cache Behavior Prediction for Real-Time Systems, Saarland Univerity, Saarbrücken, Germany*. PhD thesis, 1997.
- [15] The Ganssle Group. Perfecting the Art of Building Embedded Systems. <http://www.ganssle.com>, May 2003.
- [16] K. Hammond. Hume: a Bounded Time Concurrent Language. In *Proceedings of the IEEE Conf. on Electronics and Control Systems (ICECS '02)*, pages 407–411, Kaslik, Lebanon, December 2000. IEEE Press.
- [17] K. Hammond. Is it Time for Real-Time Functional Programming? In *Trends in Functional Programming, volume 4*. Intellect, 2004.
- [18] K. Hammond and G.J. 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.
- [19] K. Hammond and G.J. Michaelson. Predictable Space Behaviour in FSM-Hume. In *Proc. Implementation of Functional Langs.(IFL '02), Madrid, Spain*, number 2670 in Lecture Notes in Computer Science. Springer-Verlag, 2003.
- [20] J. Hawkins and A.E. Abdallah. Behavioural Synthesis of a Parallel Hardware JPEG Decoder from a Functional Specification. In *Proc. EuroPar 2002*, August 2002.
- [21] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003. Special Issue on Real-Time Systems.
- [22] M. Hofmann. A Type System for Bounded Space and Functional In-place Update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

- [23] M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.
- [24] M. Hofmann. The strength of non size-increasing computation. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, pages 258–289, 2002.
- [25] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL’03 — Symposium on Principles of Programming Languages*, New Orleans, LA, USA, January 2003. ACM Press.
- [26] C. Holland. Telelogic 2nd Generation Tools. *Embedded Sysys. Europe*, August 2002.
- [27] Paul Hudak, Simon L. Peyton Jones, and Paul Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.
- [28] R.J.M. Hughes. Lazy memo-functions. In *Functional Programming Languages and Computer Architecture*, pages 129–146. Springer-Verlag LNCS 201, 1985.
- [29] R.J.M. Hughes and L. Pareto. Recursion and Dynamic Data Structures in Bounded Space: towards Embedded ML Programming. In *ICFP’99 — International Conference on Functional Programming*, pages 70–81, Paris, France, September 1999. ACM Press.
- [30] R.J.M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *POPL’96 — Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996. ACM.
- [31] J. Launchbury J. Matthews and B. Cook. Microprocessor Specification in Hawk. In *Proc. International Conference on Computer Science*, 1998.
- [32] Steffen Jost. *Linearly Bounded Heap Space Analysis, Ludwig-Maximilians-Universität, München, Germany (in preparation)*. PhD thesis, 2006. in preparation.
- [33] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *Proc. 9th International Static Analysis Symposium SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 294–309. Springer-Verlag, 2002.
- [34] K. Mackenzie and N. Wolverson. Camelot and Grail: Compiling a Resource-Aware Functional Language for the Java Virtual Machine. In *this book*, 2004.
- [35] G. Michaelson, K. Hammond, and J. Sérot. The Finite State-ness of Finite State Hume. In *Trends in Functional Programming, Volume 4*. Intellect, 2004.
- [36] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [37] K. Nilsen. Issues in the Design and Implementation of Real-Time Java. *Java Developers’ Journal*, 1(1):44, 1996.
- [38] Johan Nordlander, Magnus Carlsson, and Mark Jones. Programming with Time-Constrained Reactions (unpublished report). <http://www.cse.ogi.edu/pacsoft/projects/Timber/publications.htm>. 2006.
- [39] A.J. 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*, number 2670 in *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

- [40] P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.
- [41] J.H. Reppy. CML: a Higher-Order Concurrent Language. In *Proc. 1991 ACM Conf. on Prog. Lang. Design and Impl. (PLDI '91)*, pages 293–305, June 1991.
- [42] M. Sakkinen. The Darker Side of C++ Revisited. Technical Report 1993-I-13, University of Jyväskylä, 1993.
- [43] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, December 1998.
- [44] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *Proc. 2003 Intl. Conf. on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, 2003.
- [45] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 February 1997.
- [46] D.A. Turner. Elementary Strong Functional Programming. In *Proc. 1995 Symp. on Funct. Prog. Langs. in Education — FPLE '95*, LNCS. Springer-Verlag, December 1995.
- [47] D.A. Turner. Total Functional Programming. *Journal of Universal Computing*, 10(7):751–768, 2004.
- [48] L. Unnikrishnan, S.D. Stoller, and Y.A. Liu. Automatic Accurate Stack Space and Heap Space Analysis for High-Level Languages. Technical Report 538, Computer Science Dept, Indiana University, April 2000.
- [49] P.B. Vasconcelos and K. Hammond. Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proc. Implementation of Functional Languages (IFL 2003)*, 2004.
- [50] M. Wallace and C. Runciman. Extending a Functional Programming System for Embedded Applications. *Software: Practice & Experience*, 25(1), January 1995.
- [51] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *Intl. Conf. on Functional Programming (ICFP '01)*, Florence, Italy, September 2001. ACM.
- [52] C. Wikström and H. Nilsson. Mnesia — an industrial database with transactions, distribution and a logical query language. In *Proc. Intl. Symp. on Cooperative Database Systems for Advanced Applications*, 1996.