

Worst-Case Execution Times for a Purely Functional Language

Armelle Bonenfant¹, Christian Ferdinand², Kevin Hammond¹, and Reinhold Heckmann²

¹ School of Computer Science, University of St Andrews, St Andrews, UK
² AbsInt GmbH, Saarbrücken, Germany

Abstract. This paper provides guaranteed bounds on worst-case execution times for a strict, purely functional programming notation. Our approach involves combining time information obtained using a low-level commercial analyser with a high-level source-derived model to give worst-case execution time information. We validate our results using concrete timing information obtained using machine code fragments executing on a Renesas M32C/85 microcontroller development board. Our results confirm experimentally that our worst-case execution time model is a good predictor of execution times.

1 Introduction

Information on worst-case execution time is essential for programming a variety of *dependable systems*, such as those found in safety-critical or mission-critical domains. With their emphasis on functional correctness, functional programming languages would appear to be a good match to dependable systems requirements. However, it is also necessary to provide an equally rigorous approach to behavioural information, especially worst-case execution times. In the functional programming community, there have been some successes in obtaining recursion bounds, typically for linearly-bounded programs (e.g. [20]), and we are working on extending these approaches to non-linear cases [31]. However, without good quality time information, these approaches will not provide strong guarantees of worst-case execution time. Simple timing metrics, based on e.g. step counts [10] are clearly inadequate in this respect.

In this paper, we consider how to obtain *guaranteed* upper bounds on execution time for a strict, purely functional expression notation. Expressions are related to an underlying abstract machine implementation through a formal translation process. We obtain concrete costs for each abstract machine instruction and provide a model to relate these costs to the functional language source. Our work is undertaken in the context of Hume [17, 16], a functionally-based language that takes a layered approach to language design. embedding purely functional expressions within a high-level process model. In principle, however, our approach is equally applicable to other strict functional programming languages, or even, by extrapolation, to other paradigms. This paper provides the

first set of guaranteed WCET results for a functional language (in fact to our knowledge, the first set that has been formally related to any high-level programming language), and develops a new approach to WCET based on aggregating costs of individual abstract machine examples. For the simple architecture we have tested (a Renesas M32C/85 microprocessor typical of real-time embedded systems applications), this approach gives a surprisingly good estimate of the actual execution cost.

This paper is structured as follows: the remainder of this section discusses possible approaches to predicting worst-case executions; Section 2 describes the Hume Abstract Machine [14] that forms the target for our measurements; Section 3 introduces AbsInt’s **aiT** tool for measuring worst-case execution times of low-level programs; Section 4 discusses experimental results and provides a comparison with the **aiT** tool; Section 5 outlines a cost model for Hume and gives experimental results showing that the cost model correctly predicts upper bounds on worst-case execution times for a compiled implementation of the HAM on a Renesas M32C/85 microcontroller development board; in Section 6, we discuss related work; finally, Section 7 concludes and considers further work.

1.1 Predicting Worst-Case Execution Time (WCET)

Obtaining high-quality WCET results is important in order 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 to obtain worst-case execution times: *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. While they may give good estimates of actual execution costs, they cannot usually *guarantee* upper bounds on execution cost. Probabilistic approaches build on experimental measurements by measuring costs for repeated executions over a suite of test cases [2, 3]. Under the assumption that the test suite provides representative data, it is then possible to construct statistical profiles that can be used to determine worst-case execution time to some stated probability. Absolute guarantees cannot, however, be provided. Finally, *static analysis* approaches (e.g. [10, 23]) construct detailed and precise models of processor instruction timings in order to be able to predict worst-case timings. This typically involves constructing accurate models of the processor state, including cache and pipeline information.

The primary advantage of measurement or probabilistic approaches is that they may be applied to arbitrary computer architectures, without detailed knowledge of the underlying design, and using relatively unsophisticated timing techniques. In contrast, static analyses require detailed architectural knowledge and painstaking effort to construct. Moreover, some architectural features, such as Pseudo-LRU replacement policies for caches present specific difficulties. However static analysis approaches provide the only guaranteed bounds of worst-

case execution time, and are therefore to be preferred for use in safety-critical or mission-critical systems.

1.2 Research methodology

Our approach involves extending recent work on static analysis of space costs for source-level functional programs [18], where we have considered the use of sized types [32] to expose bounds on recursive function definitions, to cover worst-case execution times. This involves combining information about high-level language constructs obtained from source-level analysis with low-level timing information. By basing our time metrics on a high-level abstract machine, the Hume Abstract Machine (HAM), we can provide a strong compilation structure that can easily be re-targeted to different platforms, without restricting future compilation directly to machine code. We also obtain a set of metrics that can be rapidly applied to the analysis of as-yet-unwritten programs, without the need for sophisticated and time-consuming programmer intervention to guide the tools, as is currently required. The disadvantage is that there may be some performance losses compared with the most sophisticated global optimisation techniques. At this point in our research, we feel that this is a reasonable trade, though it is an issue that we intend to revisit in future.

This paper reports results based on comparing the bounds obtained by static analysis against measured execution times for individual HAM instructions on a concrete target architecture: the Renesas M32C/85 microcontroller [7]. This is a microprocessor architecture typical of many used in sensor network and similar embedded systems – desktop processors such as modern Pentium IVs are rarely used in real-time embedded systems, both for cost reasons, and because their architectures make it hard to predict real-time costs. While not seriously restricting future architectural choices for our analyses, it provides a relatively simple, but realistic architecture on which we may be able to obtain accurate timings. It also exploits an existing Hume port compiling to concrete machine code: we use the `ham2c` translation, which compiles HAM instructions through C to produce machine code that can be executed directly on a bare-bones development board. The use of bare hardware is important, since it gives us a good real-time experimental framework. The microcontroller board we are using has a total of 16KB of memory. We use a compiled implementation of the HAM for the M32C, and assemble the machine code, the runtime system and all dynamic memory requirements into this space.

2 The Hume Abstract Machine (HAM)

This section outlines a formal compilation scheme for translating Hume programs into HAM instructions. Our intention is to demonstrate that a formal (and ultimately provable) model of compilation can be constructed for Hume. By constructing a formal translation to real machine code from HAM code, it is then possible to verify both correctness of the compiler output and time/space cost models. We provide full information here so that it is possible to properly

$\mathcal{C}_E \rho (c e_1 \dots e_n)$	$= \mathcal{C}_E \rho e_n ++ \dots ++ \mathcal{C}_E \rho e_1 ++ \langle \text{MkCon } c \ n \rangle$
$\mathcal{C}_E \rho (f e_1 \dots e_n)$	$= \mathcal{C}_E \rho e_n ++ \dots ++ \mathcal{C}_E \rho e_1 ++ \langle \text{Call } f, \text{Slide } n \rangle$
$\mathcal{C}_E \rho (i)$	$= \langle \text{MkInt32 } i \rangle$
\dots	
$\mathcal{C}_E \rho (*)$	$= \langle \text{MkNone} \rangle$
$\mathcal{C}_E \rho (var)$	$= \langle \text{PushVar } (\rho \ var) \rangle$
$\mathcal{C}_E \rho (\text{if } c \ \text{then } t \ \text{else } f)$	$= \mathcal{C}_E \rho c ++ \langle \text{If } lt \rangle ++ \mathcal{C}_E \rho f ++$ $\langle \text{Goto } ln, \text{Label } lt \rangle ++ \mathcal{C}_E \rho t ++$ $\langle \text{Label } ln \rangle$
$\mathcal{C}_E \rho (\text{let } d_1 \dots d_n \ \text{in } e)$	$= \text{let } \rho' = \text{bindDefs } \langle d_1, \dots, d_n \rangle \rho \ \text{in}$ $\langle \text{Call } ll, \text{Goto } ln, \text{Label } ll, \text{CreateFrame } n \rangle ++$ $\mathcal{C}_{Let} \rho \ 0 \ d_1 ++ \dots ++ \mathcal{C}_{Let} \rho \ (n-1) \ d_n ++$ $\mathcal{C}_E \rho' e ++ \langle \text{Return, Label } ln \rangle$
$\mathcal{C}_{Let} \rho \ n \ (id = e)$	$= \mathcal{C}_E \rho e ++ \langle \text{MakeVar } n \rangle$

Fig. 1. Compilation Rules for Expressions

situate the time cost results given in Section 3.2 and so that the cost model of Section 5 can both be understood and extended by the reader. A formal semantics of the HAM would, of course, clearly be redundant for this purpose (since it does not convey time information it would, in fact, be useless), and we therefore omit a description here (a complete description of the HAM may, however, be obtained, if required, from <http://www.embounded.org>). Figures 1–4 outline rules for compiling Hume abstract syntax forms into the HAM in [14], as a formal compilation scheme similar to that for the G-machine [1]. These rules have been used to construct a compiler from Hume source code to the HAM, whose main component is a 500-line Haskell module translating abstract syntax to HAM instructions. The compilation scheme makes extensive use of a simple sequence notation: $\langle i_1, \dots, i_n \rangle$ denotes a sequence of n items. The $++$ operation concatenates two such sequences. Many rules also use an environment ρ which maps identifiers to $\langle \text{depth}, \text{offset} \rangle$ pairs.

Four auxiliary functions are used, but not defined here: *maxVars* calculates the maximum number of variables in a list of patterns; *bindDefs* augments the environment with bindings for the variable definitions taken from a declaration sequence – the *depth* of these new bindings is 0, whilst the depth of existing variable bindings in the environment is incremented by 1; *bindVars* does the same for a sequence of patterns; and *labels* generates new labels for a set of function/box rules. Note that where labels *lt*, *ln*, *lx* etc. are used, these are assumed to be unique in the obvious way: there is at most one **Label** pseudo-instruction for each label in the translated program. Labels for boxes and function blocks are derived in a standard way from the (unique) name of the box or function. Finally, priming (e.g. ρ') is simply used for naming purposes.

$\mathcal{C}_D \rho (\mathbf{box} \ b \ ins \ outs \ \mathbf{fair} \ rs \ \mathbf{handle} \ xs) = \mathcal{C}_B \rho \ true \ b \ ins \ outs \ rs$ $\mathcal{C}_D \rho (\mathbf{box} \ b \ ins \ outs \ \mathbf{unfair} \ rs \ \mathbf{handle} \ xs) = \mathcal{C}_B \rho \ false \ b \ ins \ outs \ rs$ $\mathcal{C}_D \rho (f = \langle p_1 \rightarrow e_1 \dots p_n \rightarrow e_n \rangle) =$ $\quad \mathbf{let} \ nvars = \mathit{maxVars} \langle p_1, \dots, p_n \rangle \ \mathbf{in}$ $\quad \langle \mathbf{Label} \ f, \mathbf{CreateFrame} \ nvars \rangle ++$ $\quad \mathcal{C}_F \rho \langle \langle p_1 \rangle \rightarrow e_1, \dots, \langle p_n \rangle \rightarrow e_n \rangle ++$ $\quad \langle \mathbf{Function} \ f \ (\mathit{labels} \ f) \rangle$ $\mathcal{C}_B \rho \ f \ b \ (in_1, \dots, in_i) \ (out_1, \dots, out_m) \ rs =$ $\quad \mathbf{let} \ nvars = \mathit{maxVars} \langle in_1, \dots, in_i \rangle \ \mathbf{in}$ $\quad \langle \mathbf{Label} \ b \rangle ++$ $\quad \langle \mathbf{CopyInput} \ (i - 1), \dots, \mathbf{CopyInput} \ 0 \rangle ++$ $\quad \langle \mathbf{Push} \ 2, \mathbf{CreateFrame} \ nvars \rangle ++$ $\quad (\mathit{if} \ f \ \mathit{then} \ \langle \mathbf{StartMatches} \rangle \ \mathit{else} \ \langle \rangle) ++ \mathcal{C}_R \rho \ f \ m \ rs \ ++$ $\quad \langle \mathbf{Box} \ b \ \dots \rangle$

Fig. 2. Compilation Rules for Declarations and Box Bodies

The rules are structured by abstract syntax class. The rules for translating expressions (\mathcal{C}_E etc. – Figure 1) are generally straightforward, but note that function frames are created to deal with *let*-expressions and other similar structures, which then exploit the function calling mechanism. This allows the creation of local stack frames. It would obviously be possible to eliminate the function call for *let*-expressions provided the stack frame was properly set up in order to allow access to non-local definitions.

Hume programs define a number of concurrent processes. Each process is defined in terms of a “box” that maps some inputs to some outputs [17]. Box inputs/outputs are connected to form a static process network. The rules for translating box and function declarations are shown in Figure 2. These rules create new stack frames for the evaluation of the box or function, label the entry points and introduce appropriate pseudo-instructions. In the case of box declarations, it is also necessary to copy inputs to the stack using **CopyInput** instructions and to deal with fair matching. Box bodies are compiled using $\mathcal{C}_R/\mathcal{C}_{R'}$ (Figure 3). These rules compile matches for the outer level patterns using \mathcal{C}_P , then compile inner pattern matches using \mathcal{C}_A , before introducing **Consume** instructions for non-* input positions. The RHS can now be compiled. If more than one result is to be produced, the tuple of outputs is unpacked onto the stack. A **Check-Outputs** is inserted to verify that the outputs can be written using appropriate **Write** instructions. Finally, a **Reorder** is inserted if needed to deal with fair matching, and a **Schedule** returns control to the scheduler. The compilation of function/handler bodies using $\mathcal{C}_F/\mathcal{C}_{F'}$ is similar, except that $\mathcal{C}_{P'}$ is used rather than \mathcal{C}_P , there is no need to deal with box inputs/outputs or fair matching, and a **Return** rather than **Schedule** is inserted at the end of each compiled rule. For simplicity, but without loss of generality, we ignore exception handlers.

$\mathcal{C}_R \rho f m \langle r_1, \dots, r_n \rangle$	$=$	$\mathcal{C}_{R'} \rho f m r_1 ++ \dots ++ \mathcal{C}_{R'} \rho f m r_n$
$\mathcal{C}_{R'} \rho f m (\langle p_1, \dots, p_n \rangle \rightarrow e)$	$=$	$\mathbf{let} \rho' = \mathit{bindVars} \langle p_1, \dots, p_n \rangle \rho \mathbf{in}$ $\langle \mathit{Label} \mathit{lr}, \mathit{MatchRule} \rangle ++$ $\mathcal{C}_P p_1 ++ \dots ++ \mathcal{C}_P p_n ++$ $\mathcal{C}_A p_1 ++ \dots ++ \mathcal{C}_A p_n ++$ $\mathcal{C}_C 0 p_1 ++ \dots ++ \mathcal{C}_C (n-1) p_n ++$ $\mathcal{C}_E \rho' e ++$ $(\mathit{if} m > 1 \mathit{then} \langle \mathit{Unpack} \rangle \mathit{else} \langle \rangle) ++$ $\langle \mathit{CheckOutputs} \rangle ++$ $\langle \mathit{Write} (n-1) \dots \mathit{Write} 0 \rangle ++$ $(\mathit{if} f \mathit{then} \langle \mathit{Reorder} \rangle \mathit{else} \langle \rangle) ++$ $\langle \mathit{Schedule} \rangle$
$\mathcal{C}_C n (*)$	$=$	$\langle \rangle$
$\mathcal{C}_C n (p)$	$=$	$\langle \mathit{Consume} n \rangle$
$\mathcal{C}_F \rho \langle r_1, \dots, r_n \rangle$	$=$	$\mathcal{C}_{F'} \rho r_1 ++ \dots ++ \mathcal{C}_{F'} \rho r_n$
$\mathcal{C}_{F'} \rho (\langle p_1, \dots, p_n \rangle \rightarrow e)$	$=$	$\mathbf{let} \rho' = \mathit{bindVars} \langle p_1, \dots, p_n \rangle \rho \mathbf{in}$ $\langle \mathit{Label} \mathit{lf}, \mathit{MatchRule} \rangle ++$ $\mathcal{C}_{P'} p_1 ++ \dots ++ \mathcal{C}_{P'} p_n ++$ $\mathcal{C}_A p_1 ++ \dots ++ \mathcal{C}_A p_n ++$ $\mathcal{C}_E \rho' e ++$ $\langle \mathit{Return} \rangle$

Fig. 3. Compilation Rules for Rule Matches and Functions

Finally patterns are compiled using $\mathcal{C}_P/\mathcal{C}_{P'}$ (Figure 4), where \mathcal{C}_P inserts the **MatchNone**/**MatchAvailable** instructions that are needed at the box level, and $\mathcal{C}_{P'}$ compiles simple patterns. Constructed values are matched in two stages: firstly the constructor is matched, and then if the match is successful, the matched object is deconstructed on the stack to allow its inner components to be matched against the inner patterns. These nested patterns are compiled using \mathcal{C}_A and \mathcal{C}_N . \mathcal{C}_A inserts **CopyArg** and **Unpack** instructions to decompose function/box arguments, where \mathcal{C}_N deals with the general nested case using **Copy** instructions to replicate items that are in the local stack frame.

3 Static Analysis using the aiT Tool

Motivated by the problems of measurement-based methods for WCET estimation, AbsInt GmbH has investigated an approach based on static program analysis [22, 19, 13]. 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 tim-

$\mathcal{C}_P (*)$	=	$\langle \text{MatchNone} \rangle$
$\mathcal{C}_P (_*)$	=	$\langle \text{MatchNone} \rangle$
$\mathcal{C}_P (p)$	=	$\langle \text{MatchAvailable} \rangle ++ \mathcal{C}_{P'} p$
$\mathcal{C}_{P'} (i)$	=	$\langle \text{MatchInt32 } i \rangle$
...		
$\mathcal{C}_{P'} (c p_1 \dots p_n)$	=	$\langle \text{MatchCon } c n \rangle$
$\mathcal{C}_{P'} (var)$	=	$\langle \text{MatchVar } var \rangle$
$\mathcal{C}_{P'} _$	=	$\langle \text{MatchAny} \rangle$
$\mathcal{C}_A (c p_1 \dots p_n)$	=	$\mathcal{C}_{A'} \langle p_1, \dots, p_n \rangle$
$\mathcal{C}_A (p_1, \dots, p_n)$	=	$\mathcal{C}_{A'} \langle p_1, \dots, p_n \rangle$
$\mathcal{C}_A (x p)$	=	$\mathcal{C}_{A'} \langle p \rangle$
$\mathcal{C}_A p$	=	$\langle \rangle$
$\mathcal{C}_{A'} \langle p_1, \dots, p_n \rangle$	=	$\langle \text{CopyArg } n, \text{Unpack} \rangle ++$ $\mathcal{C}_N p_1 ++ \dots ++ \mathcal{C}_N p_n ++$ $\mathcal{C}_{P'} p_1 ++ \dots ++ \mathcal{C}_{P'} p_n$
$\mathcal{C}_N \langle p_1, \dots, p_n \rangle$	=	$\langle \text{Copy } n, \text{Unpack} \rangle ++$ $\mathcal{C}_N p_1 ++ \dots ++ \mathcal{C}_N p_n ++$ $\mathcal{C}_{P'} p_1 ++ \dots ++ \mathcal{C}_{P'} p_n$

Fig. 4. Compilation Rules for Patterns

ing of hardware instructions, which leads to safe and precise WCET calculations that are valid for all executions of the application. The results of AbsInt GmbH's *aiT* tool [12] can be combined using Integer Linear Programming techniques to safely predict the worst-case execution time and a corresponding worst-case execution path. Whilst the analysis works at a level that is more abstract than simple basic blocks, it is not capable of managing the complex high-level constructs that we require. It can, however, provide useful and accurate worst-case time information about lower level constructs. We are thus motivated to link the two levels of analysis, combining information on recursion bounds and other high-level constructs that we will obtain from the Hume source analysis we are constructing, with the low-level worst-case execution time analysis that can be obtained from the AbsInt analysis. In order to achieve this, we will eventually require two-way information flow between the analyses. In the short-term, it is sufficient to provide one-way flow from the language-level analysis to the lower-level analysis.

The **aiT** tool is a robust commercial tool. It has previously been applied to several other architectures used in embedded systems with similarly good results. It has also proved sufficiently flexible to deal with a variety of application domains including real-time operating systems [26], an automotive communications system [5], construction vehicles [27], and avionics [28]. The use of an abstract machine as the analysis target represents a new challenge for the **aiT**

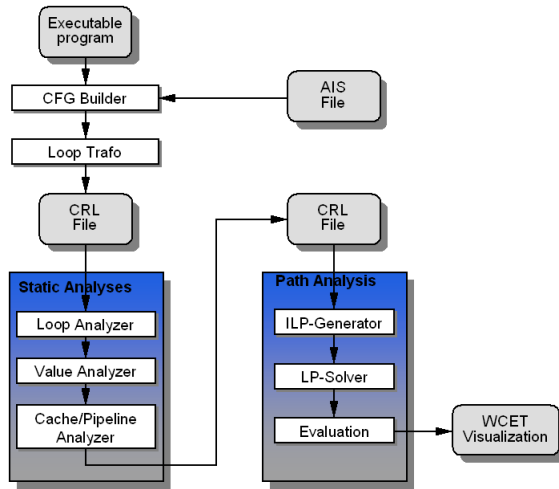


Fig. 5. Phases of WCET computation

tool, however, since the structure of instructions that need to be analysed can be significantly different from those that are hand-produced, and the associated technical problems in producing cost information can therefore be more complex.

3.1 Determining WCET using the aiT tool

The **aiT** tool determines the worst-case execution time of a program task in several phases, as shown in Figure 5. These phases are:

- **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 [11];
- **Pipeline Analysis** predicts the behavior of the program on the processor pipeline [22];
- **Path Analysis** determines a worst-case execution path of the program [30].

The cache analysis phase uses the results of the value analysis phase to predict the behavior 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 [9], a

Instruction	gcc	IAR	Ratio
Call	73	70	1.04
Copy	43		
CopyArg	40	35	1.14
CreateFrame	76	72	1.06
Goto	5	3	1.67
If (true)	41	32	1.28
If (false)	41	32	1.28
MakeVar	43	36	1.19
MatchExn	808		
MatchedRule	11	11	1.00
MatchInt	811	137	5.92
MatchRule	22	22	1.00
MatchVar	46	36	1.28
MkBool	136		

Instruction	gcc	IAR	Ratio
MkChar	136		
MkCon 2	348	242	1.44
MkFun 0	198	165	1.20
MkInt	136	91	1.49
MkNone	26	21	1.24
MkVector 3	392	205	1.91
Pop	13		
Push	12	11	1.09
PushVar	40	35	1.14
Return	1756		
Schedule	410	602	0.68
Slide	62	53	1.17
SlideVar	94		
TailCall	91	178	0.51

Fig. 6. aiT HAM analysis: gcc and IAR compiled code

semantics-based method for static program analysis. Integer linear programming is then used for the final path analysis phase.

3.2 Worst-Case Execution Time for HAM Instructions

Figure 6 lists guaranteed worst-case execution time results for a subset of Hume Abstract Machine instructions, ordered alphabetically, and reported in terms of clock cycles. These timings were obtained using the **aiT** tool from code generated using the **ham2c** Hume to C compiler, cross-compiling through either gcc Version 3.4 or the IAR C compiler [29] to the Renesas M32C. As expected from a commercial compiler targeting a few architectures, the IAR compiler generally produces more efficient code than gcc, with our results being 42% lower on average, and up to 5.92 times more efficient in the case of **MatchInt**. In a few cases, the **aiT** tool was unable to provide timing information directly, requiring additional information such as loop bounds to be provided in order to produce timing results. Missing entries in this table represent cases where this information could not be obtained. In the long term, we anticipate that we will be able to provide this information by analysis of Hume source constructs. In the short term, we have calculated the information by hand, where possible. For some instructions, however, we were unable to provide this information for the IAR-compiled code, and results for these instructions are therefore given only for gcc-produced code.

4 Experimental Timings

We have developed an approach based on repeated timing of code fragments. Each fragment to be timed is executed a certain (large) number of times. This ensures that we obtain a measurable time, even for times that are below the clock threshold. In order to ensure that computations can be repeated, it is necessary to save and restore the computation state between executions, thereby incurring some time overhead. So that this overhead does not affect our timing results, we must therefore first take a witness timing that simply incurs this overhead. This is subtracted from the measured time to give an average time for the code fragment of interest. Since the M32C/85 clock is cycle-accurate, it is also possible to obtain an exact execution time for a given code fragment. We have adapted the timing approach described above to give measured worst-case execution times, by recording the measured maximum time for the code fragment of interest. The same approach can also be used to give best-case timings.

4.1 Timing Results

Figure 7 shows average execution and worst-case execution times obtained using the timing approach described above, for HAM instructions compiled using the IAR compiler. Each average and worst-case entry has been obtained from 10000 individual timings. We can see from the table that the worst-case times and average-case times are very similar for most instructions, indicating that the instruction timings are highly consistent in practice. Since certain instructions are parameterised on some argument (for example, **MkVector** is parameterised on the vector size), in these cases, we have measured several points and applied linear interpolation to obtain a cost formula. It is interesting to note that in these cases, the linear factor is identical for both WCET and average times and the constants are also very close. In each case, we have subtracted the least time obtained from timing the empty sequence of instructions (39 clock cycles), in order to give a conservative worst-case time. Since the worst-case time for the empty sequence was 42 cycles, this means that the worst-case may, in fact, be up to three cycles less than the numbers reported here. Since we must save and restore the abstract machine state (and this will, as a side effect, clear the cache and other processor state), we needed to develop code that does this correctly. A few abstract machine instructions have therefore not been costed, mainly because they perform more complex state changes that may require additional intervention. It is worth noting that the values included in this table give a good timing predictor, but one that could only be used to provide absolute worst-case guarantees under some statistical probability.

4.2 Quality of the Static Analysis using the aiT Tool

Figure 8 compares the upper bounds on worst-case execution timing obtained using the aiT tool from Figure 6 with the corresponding measured worst cases from Figure 7. We can see that in all cases apart from **MatchRule**, the static

Instructions	AVG	WCET	Ratio
Ap	760	761	1.00
Call	61	62	1.02
Callprim			
== Bool	246	251	1.02
* Float	240	242	1.01
+ Float	262	267	1.02
== Float	255	260	1.02
- Int	114	119	1.04
* Int	130	132	1.02
/ Int	168	177	1.05
+ Int	114	119	1.04
< Int	215	220	1.02
== Int	216	221	1.02
> Int	217	223	1.03
Consume	27	31	1.24
Copy	27	31	1.15
CopyArg	27	30	1.11
CreateFrame	51	57	1.12
Goto	1	2	2.00
If (true)	24	29	1.21
If (false)	24	26	1.08
MakeVar	26	31	1.19
MatchAny	6	10	1.67
MatchAvailable	7	10	1.43
MatchBool	24	29	1.21
MatchCon	22	26	1.18
MatchedRule	8	12	1.50
MatchExn	22	28	1.27
MatchFloat	24	29	1.21
MatchInt	23	29	1.26

Instructions	AVG	WCET	Ratio
MatchNone	6	10	1.67
MatchRule	18	23	1.28
MatchString n	$3 \times n$ + 45	$3 \times n$ + 47	
MatchTuple	6	10	1.67
MatchVar	26	31	1.19
MaybeConsume	20	28	1.40
MkBool	63	70	1.11
MkChar	63	70	1.11
MkCon n	$41 \times n$ + 84	$41 \times n$ + 89	
MkFun n	$42 \times n$ + 108	$42 \times n$ + 113	
MkInt	64	65	1.02
MkNone	15	21	1.40
MkString n	$13 \times n$ + 133	$13 \times n$ + 140	
MkTuple n	$41 \times n$ + 63	$41 \times n$ + 66	
MkVector n	$41 \times n$ + 63	$41 \times n$ + 65	
Pop	6	9	1.50
Push	6	9	1.50
PushVar	27	30	1.11
PushVarF	37	40	1.08
Raise	374	377	1.01
Return	112	116	1.04
Slide	41	44	1.07
SlideVar	58	63	1.09
Unpack	114	118	1.04

Fig. 7. Experimental average and worst-case timings for HAM instructions

analysis gives an upper bound that is greater than or equal to the measured execution time. For **MatchRule**, the static analysis yields an upper bound that is one cycle smaller than our measured worst-case. Since our worst case timings are conservative, and may have an experimental error of up to three clock cycles, as described above, we conclude that the static analysis correctly yields upper bounds on execution costs for these HAM instructions. For the instructions we have compared, the bound given by the static analysis is at most 50% greater than the measured worst-case (for **Goto**, representing a difference of only one clock cycle); the mean difference is 22%, with a standard deviation of 16%. We

Instructions	aiT bound	Measured WCET	Ratio
Call	70	62	1.13
CopyArg	35	30	1.17
CreateFrame	72	57	1.26
Goto	3	2	1.50
If (true)	32	29	1.10
If (false)	32	26	1.23
MakeVar	36	31	1.16
MatchRule	22	23	0.96

Instructions	aiT bound	Measured WCET	Ratio
MatchVar	36	31	1.16
MkCon 2	242	170	1.42
MkFun 0	165	113	1.46
MkInt	91	65	1.40
MkNone	21	21	1.00
Push	11	9	1.22
PushVar	35	30	1.17
Slide	53	44	1.20

Fig. 8. Quality of the Static Analysis

conclude that the static analysis provides an accurate upper bound on execution time.

5 Worst-Case Execution Time for Hume Expressions

In this section, we outline a cost model for deriving worst-case time based on the Hume operational semantics, and compare the results we obtain against measured execution times. Our cost model is defined in terms of a formal operational semantics for the Hume Abstract Machine that is related back to Hume source expressions. Our cost rules are given in a derivation form as follows:

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

where e represents an expression in our source language. t is an upper bound of the number of time units available to evaluate e , and t' is the number of time units available left 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 provide pattern matching costs, we add rules of the form:

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

This means that a single step match of ℓ against pat succeeds (with ℓ interpreted within heap η). In order to complete the whole pattern match, the sublists $\vec{\ell}$ and \vec{pat} must still be matched. Further, \mathcal{V}' is the environment \mathcal{V} extended with any bindings made in this, and \mathcal{A}' extends \mathcal{A} with the locations that have been matched successfully in this step. Finally, $t - t'$ is the time required to match the location ℓ against pat . We then 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 .

In order to illustrate our approach we include only a few representative rules here. A complete set of rules, forming a complete cost model for Hume in terms of the costs incurred by the HAM, may be found at <http://www.embounded.org>. Each rule in the cost model is derived from the formal translation of Section 2, and this translation also allows us to derive formal properties including the soundness of the cost model against the formal operational semantics of the HAM which we have previously constructed. The first rule we consider (CONST INT) deals with constant integers. We first allocate a new location ℓ for the given constant n . The cost of the evaluation is given by the constant \mathbf{Tmkint} , which is the time required by the \mathbf{MkInt} instruction, as calculated above. Similar rules can be constructed for any other kind of constant.

$$\frac{n \in \mathbb{Z} \quad \mathbf{NEW}(\eta) = \ell \quad w = (\mathbf{int}, n)}{\mathcal{V}, \eta \vdash \frac{t' + \mathbf{Tmkint}}{t'} n \rightsquigarrow \ell, \eta[\ell \mapsto w]} \quad (\mathbf{CONST INT})$$

The VARIABLE rule looks up x in \mathcal{V} to obtain the variable location. The time this takes is given by the cost of the underlying $\mathbf{PushVar}$ instruction, $\mathbf{Tpushvar}$.

$$\frac{\mathcal{V}(x) = \ell}{\mathcal{V}, \eta \vdash \frac{t' + \mathbf{Tpushvar}}{t'} x \rightsquigarrow \ell, \eta} \quad (\mathbf{VARIABLE})$$

The VECTOR rule deals with costs for literal arrays (vectors). It is an example of a typical rule that exposes costs for constructed expressions. Each component of the vector is evaluated to yield a heap location. Since memory must be allocated for each component and this will incur some time cost, the cost of constructing the vector therefore depends on the number of components $\mathbf{Tmkvec}(k)$ plus the time costs for evaluating each component. While this is not theoretically necessary, we use the convention of showing the $\mathbf{Tmkvec}()$ costs below the line to make it clear that they are incurred after evaluating each of the vector components in the HAM implementation.

$$\frac{\begin{array}{l} \mathcal{V}, \eta_{(i-1)} \vdash \frac{t_{(i-1)}}{t_i} e_i \rightsquigarrow \ell_i, \eta_i \quad (\text{for } i = 1, \dots, k) \\ k \geq 1 \quad \mathbf{NEW}(\eta_k) = \ell \quad w = (\mathbf{constr}_c, \ell_k, \dots, \ell_1) \end{array}}{\mathcal{V}, \eta_0 \vdash \frac{t_0}{t_k - \mathbf{Tmkvec}(k)} \langle\langle e_k \dots e_1 \rangle\rangle \rightsquigarrow \ell, \eta_k[\ell \mapsto w]} \quad (\mathbf{VECTOR})$$

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 $\mathbf{Tmatchrule}$ is added to the cost of evaluating the match. The successful match incurs an additional cost of $\mathbf{Tmatchedrule}$. Finally the function body is evaluated. Before and after evaluation, we incur costs of $\mathbf{Tcreateframe}$ and $\mathbf{Treturn}$ respectively, corresponding to the time costs of constructing a stack frame and returning from the function call.

Clock cycles	aiT (gcc)	aiT (IAR)	average (IAR)	WCET (IAR)	Measured
T_{init}	4704	4314	2390	2449	
T_{loop}	2317	2249	2173	2215	
T_{global}	16289	15559	13255	13524	13120
Ratio predicted/measured	1.24	1.19	1.01	1.03	

Fig. 9. Example: findNewCentre

and T_{rec}) is then $T_{global} = T_{init} + k \times T_{loop}$

Figure 9 gives concrete values for these variables produced using either the aiT tool or by measurement. Experimental figures for T_{call} etc are obtained from Figure 7, while those for aiT are obtained from Figure 6. Note that while there is, in principle, no technical problem with obtaining upper bounds on T_c and T_{rec} using the aiT tool, since we do not have figures for all HAM instructions, we have chosen to measure these values, giving worst-case execution times of $T_c = 1429$ and $T_{rec} = 539$. For similar reasons, we have also used the gcc figure for the **Return** instruction. This is a clear over-estimate of the worst-case time. We can see from the figure that the aiT tool gives a result that is within 24% or 19% of the actual measured cost for T_{global} , and that the experimental result is within 1% for the average measurement and 3% for the WCET measurement. This shows both that good predictions can be obtained for code fragments using the aiT tool, and that the approach of summing abstract machine costs to provide an overall time predictor can be a valid one.

6 Related Work

This paper represents the first study of guaranteed worst-case execution times for functional languages of which we are aware, and the first to consider time predictions based on analysis of underlying abstract machine instructions. Our static analysis approach is unusual in being based on high-level analysis of source code, rather than being reconstructed from the more usual low-level control-flow or data-flow graphs (e.g. [2, 5, 8, 27]), which are inevitably approximate in terms of representing programmer intent or high-level language constructs. A good survey of other recent work on WCET analysis can be found in [33].

The use of atomic timings to predict overall worst-case execution times has previously been addressed in the literature. For example, Wong [34] has considered the use of sequences of bytecodes, and Meyerhöfer and Lauterwald [24] have considered code blocks for the same purpose. Generally, however, authors have concluded that bytecodes (and even basic blocks) are at too small a granularity to form reliable timings. Where individual JVM bytecodes are measured, it can also be tedious and problematic to save/restore system state by hand [25]. There are also issues with garbage collection etc., and a real-time implementation is

clearly required. By considering a simple architecture with a cycle-accurate clock we have, however, been able to obtain measured worst-case execution times for bytecode instructions. Because we have full access to the abstract machine implementation, we are able to systematically save/restore complete system states for many HAM instructions. We have also confirmed that, for at least one simple architecture, the Renesas M32C, composition of bytecode cost information is a good WCET predictor for sequences of such bytecodes. We are not aware of any previous work that gives similar results.

7 Conclusions and Further Work

By conducting a series of experiments and applying a low-level static analysis to a representative subset of the Hume Abstract Machine instructions, we have been able to demonstrate that our time cost model is capable of predicting good upper bounds on execution time of a given Hume program both on a bytecode-by-bytecode basis (Section 4) and overall for a simple function (Section 5). Our results are shown on a microcontroller hardware platform that is typical of those found in small embedded systems. We have shown that we can obtain guaranteed worst-case execution times for a number of HAM instructions that are within 50% of the measured worst-case. Although we need to conduct further experiments to verify our results, we have been able to show that we can derive guaranteed worst-case times that are within 24% of the measured worst-case. While we have given **aiT** timings for a sufficiently representative subset of HAM instructions to allow us to explore cost information for some simple examples, we have not yet been able to obtain information for all HAM instructions. We do not anticipate any technical problems in doing this, however. We also anticipate that the general approach we have described here will apply to similar abstract machine settings, such as the JVM.

The time cost model we have outlined here is formally derived from an underlying operational semantics of the HAM. We are in the process of constructing a static analysis to automatically obtain time information for Hume source programs. This analysis builds in an essential way both on the time cost model and on the concrete time information we have presented here. By demonstrating that the cost model is a good predictor of worst-case execution time, we have increased confidence that, provided our static analysis conforms to the cost model, it will also be a good predictor of worst-case execution times.

8 Acknowledgments

This work has been generously supported by the Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre established by the UK Ministry of Defence; by EU Framework VI grant IST-2004-510255 (Em-Bounded); by EPSRC Grant EPC/0001346; and by a personal Support Fellowship from the Royal Society of Edinburgh. We would like to thank Charlotte Bjuren, who assisted with obtaining detailed timing information; Robert

Pointon, who assisted with the Renesas implementation and timing instructions; and Steffen Jost and Hans-Wolfgang Loidl, who assisted with the cost model.

References

1. L. Augustsson. *Compiling Lazy Functional Languages, Part II*. PhD thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, Sweden, 1987.
2. G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *Proc. 12th Euromicro Intl. Conf. on Real-Time Systems (ECRTS 2000)*, Stockholm, June 2000.
3. G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proc. 23rd IEEE Real-Time Systems Symposium (RTSS 2002)*, December 2002.
4. A. Bonenfant, Z. Chen, K. Hammond, G.J. Michaelson, A. Wallace, and I. Wallace. Towards resource-certified software: A formal cost model for time and its application to an image-processing example. In *ACM Symposium on Applied Computing (SAC '07), Seoul, Korea, March 11-15, 2007*.
5. S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying static WCET analysis to automotive communication software. In *17th Euromicro Conference of Real-Time Systems, (ECRTS'05)*, Mallorca, Spain, July 2005.
6. D. Comaniciu, V. Ramesh, and P. Meer. Kernel-based object tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(5):564–575, 2003.
7. Renesas Technology Corp. <http://www.renesas.com>. *Home Page*, 2006.
8. M. Corti and T. Gross. Approximation of the Worst-Case Execution Time Using Structural Analysis. In *Proc. ACM International Conference on Embedded Software (EMSOFT '04)*, 2004.
9. 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.
10. K. Cray and S. Weirich. Resource Bound Certification. In *POPL'00 — Symposium on Principles of Prog. Langs.*, pages 184–198,, Boston, MA, January 2000.
11. C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems, Saarland University, Saarbrücken, Germany*. PhD thesis, 1997.
12. 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*, pages 469–485. Springer-Verlag LNCS 2211, 2001.
13. C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2):163–189, 1999.
14. K. Hammond. Exploiting Purely Functional Programming to Obtain Bounded Resource Behaviour: the Hume Approach. In *Central European Summer School on Functional Programming, July 2005*. Springer-Verlag LNCS, to appear, 2006.
15. K. Hammond, C. Ferdinand, R. Heckmann, R. Dyckhoff, M. Hofmann, S. Jost, H.-W. Loidl, G.J. Michaelson, R. Pointon, N. Scaife, J. Sérot, and A. Wallace. Towards Formally Verifiable WCET Analysis for a Functional Programming Language. In *Proc. Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, April 2006.
16. K. Hammond and G. Michaelson. Bounded Space Programming using Finite State Machines and Recursive Functions: the Hume Approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2006, in preparation.

17. K. Hammond and G.J. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. GPCE '03: Conf. Generative Programming and Component Engineering*, pages 37–56. Springer-Verlag LNCS 2830, 2003.
18. 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.
19. 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.
20. 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.
21. R.J.M. Hughes. The Design and Implementation of Programming Languages, DPhil Thesis, Programming Research Group, Oxford. July 1983.
22. 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.
23. Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proc. RTSS '95: IEEE Real-Time Systems Symposium*, page 298, 1995.
24. M. Meyerhöfer and F. Lauterwald. Towards Platform-Independent Component Measurement. In *Proc. WCOP 2005 – Tenth International Workshop on Component-Oriented Programming, Glasgow*, July 2005.
25. P. Puschner and G. Bernat. WCET Analysis of Reusable Portable Code. In *Proc. 13th Euromicro Intl. Conf. on Real-Time Syst. (ECRTS 2001)*, pages 45–52, 2001.
26. D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static timing analysis of real-time operating system code. In *Proc. ISOLA'04: Intl. Symposium on Leveraging Applications of Formal Methods*, Cyprus, October 2004.
27. D. Sehlberg. Static WCET analysis of task-oriented code for construction vehicles. Master's thesis, Mälardalen University, October 2005.
28. J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *Proc. 2005 Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.
29. IAR Systems. <http://www.iar.com/>. *Home Page*, 2006.
30. H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proc. RTSS '98: IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, December 1998.
31. P.B. Vasconcelos. *Cost Inference and Analysis for Recursive Functional Programs*. PhD thesis, University of St Andrews, 2006, in preparation.
32. P.B. Vasconcelos and K. Hammond. Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proc. Implementation of Functional Languages (IFL '03)*, pages 86–101. Springer-Verlag LNCS 3145, 2004.
33. Reinhard Wilhelm. Determining bounds on execution times. In R. Zurawski, editor, *Handbook on Embedded Systems*, pages 14–1,14–23. CRC Press, 2005.
34. P. Wong. *Bytecode Monitoring of Java Programs*, MSc thesis, University of Warwick, 2003.