

Automatically Costed Autonomous Mobility

Xiao Yan Deng, Phil Trinder and Greg Michaelson
School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, EH14 4AS, Scotland,
{xyd3,trinder,greg}@macs.hw.ac.uk

Abstract

To share resources in open distributed systems we have developed autonomous mobile programs, which periodically use a cost model to decide where to execute in a network. In addition self-aware mobile coordination for common patterns of computation over collections are encapsulated by autonomous mobility skeletons.

Performance can be improved if an autonomous mobile program can predict the cost of the entire program rather than a single iteration. We propose a cost calculus that calculate the costs for the remainder of a computation at arbitrary program points. We extend our autonomous mobility skeleton cost models to be parametrised on the cost of the remainder of the program, and propose costed autonomous mobility skeletons. An automatic cost analyser which implements the calculus has been built in Jocaml, which outputs Jocaml programs with higher-order functions replaced by costed autonomous mobility skeletons.

1 Introduction

Developments in networks have made it possible to exploit the computational power and resources available in global networks [2]. To manage load on large and dynamic networks we have developed *autonomous mobile programs* (AMPs)[6] that periodically make a decision about where to execute in a network. The decisions are informed by cost models that measure current performance, the relative speeds of alternative network locations, and communication costs.

A disadvantage of directly programming AMPs is that the cost model, mobility decision function, and network interrogation are all explicit in the program. To encapsulate the self-aware mobile control for common patterns of computation over collections, we have explored *autonomous mobility skeletons* (AMSs) in [4]. AMSs are akin to algorithmic skeletons in being polymorphic higher-order functions, but where algorithmic skeletons abstract over parallel

coordination, AMSs abstract over autonomous mobile coordination. For example the standard `map` applies a given function to a sequence of elements and returns a sequence of results. The `automap` AMS performs the same computation as `map`, but may cause the program to migrate to a faster location.

A limitation of the AMSs is that they assume that a single higher-order function is the dominating computation for the program. For example, if there are multiple higher-order functions in the program, e.g. in program `(automap f1 l1); (automap f2 l2)`, the second `automap` is the remainder after the first `automap`. When `automap f1 l1` makes the decision to move or not, it might not move if it only considers the cost of itself, but it might move if it also considers the cost of `automap f2 l2`. In general to deploy autonomous mobility effectively, it is necessary to know the cost of the remainder of the program, and not just the cost of a single iteration.

Thus, we explore a calculus to manipulate, and ultimately automatically statically extract costs for the remainder/continuation of a computation, at arbitrary points at compile time. We have constructed an *automatic continuation cost analyser* which implements the cost calculus for a Jocaml subset to produce cost equations parametrised on program variables in context, and may be used to find both cost in higher-order functions and the continuation cost of the higher-order functions. We extend our AMS cost model to be parametrised on the cost of the remainder of the program. Costed autonomous mobility skeletons (CAMSs) are built which not only encapsulate the common pattern of autonomous mobility but take additional cost parameters representing the costs of the remainder of the program.

The structure of the paper is as follows. **Section 2** discusses related work. **Section 3** introduces costed autonomous mobility skeletons. **Section 4** defines a small strict higher-order language \mathcal{J}' and builds the cost calculus for \mathcal{J}' . **Section 5** describes the implementation of an automatic continuation cost analyser which implements the cost calculus. **Section 6** compares the performance of the programs using CAMSs with the programs using AMSs.

2 Background

2.1 Autonomous Mobile Programs & Autonomous Mobility Skeletons

AMPs have strong connections with both agents and autonomous systems. An agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives[22, 18]. Mobility is the ability to transport itself from one machine to another and retaining its current state. Agents with mobility are called *mobile agents*[12].

Autonomous systems are also called autonomic computing systems, and a definition has been given by IBM: autonomic computing system can manage themselves given high-level objectives from administrators[9]. The essence of autonomic computing systems is self-management requiring self-management are *self-configuration*, *self-optimization*, *self-healing*, and *self-protection*.

AMPs are mobile agents and self-optimization systems. They are aware of their processing resource needs and sensitive to the environment in which they execute, and are able to dynamically relocate themselves to minimise processing time in the presence of varying external loads on shared processing elements. They also have important differences from agents and autonomous system. Firstly, unlike previous mobile agent approaches, AMPs are autonomous i.e. making decision themselves when and where to move according to the cost model. AMPs also differ from tradition autonomous systems[9, 1, 19], which use schedulers to decide whether to move, AMPs themselves can make this decision.

AMSs[4] encapsulate self-aware mobile coordination for common patterns of computation over collections. Autonomous mobile programs with AMSs do not need to insert additional code for making migration decision.

Mobile languages give programmers control over the placement of code or active computations across the network for sharing computational resources e.g. CPU speed [10]. For example Jocaml[8] is a functional programming language. Java Voyager[14] and JavaGo[17] extend Java. We have previously presented AMPs[6] and the `automap`, `autofold` and `AutoIterator` AMSs[4], together with performance measurements of Jocaml, Java Voyager, and JavaGo implementations on modest LANs.

2.2 Cost Analysis

In the systems for sharing computational power, *cost models* are used to estimate the cost of a program in terms of

time and to predict the behaviour of the program[15]. There are two levels of cost models in general. *Computation cost models* estimate the sequential computation time for programs. *Coordination cost models* predict the coordination and communication behaviours of parallel, distributed and mobile programming. Usually, coordination cost models take costs which have been got from the computation cost models into account to make more efficient coordination decisions.

A generic AMP coordination cost model and problem specified cost models has been built in[6]. This paper focuses on the *computation cost models* for AMPs. This kind of cost analysis usually happens at compile time, so is also called static cost analysis. Different static cost models have been built for different systems. Cohen and Zuckerman consider cost analysis of Algol-60 programs[3]. Wegbreit works on cost analysis of Lisp programs addressed the treatment of recursion[21]. Ramshaw[13] discusses the formal verification of cost specifications. Many of the cost analysis use semantics-based methods e.g. Rosendahl[16] uses abstract interpretation for cost analysis, and Wadler[20] uses projection analysis. Loidl has built static cost semantics for for language \mathcal{L} in[11], and Reistad and Gifford built static cost for data-dependent expressions in[15].

All these computation cost analyses produce the cost of expressions, but not the costs following the expressions, which is more useful to predict the behaviour of the programs. We built a static cost calculus for a Jocaml subset. That calculates the continuation cost of each expression in a program.

3 Costed Autonomous Mobility Skeletons

Costed autonomous mobility skeletons (CAMSs) are built to improve the performance of AMSs. CAMSs e.g. `camap` and `cafold` are implemented in Jocaml. `camap f [a1; ...; an] costf costafter`, performs the same computation as `automap f [a1; ...; an]`, but takes another two arguments `costf` and `costafter`, recording the cost of `f` and the continuation cost of the higher-order functions. Similarly, the Jocaml `cafold` is implemented.

The CAMS cost model improves the AMS cost model, which only considers the cost in the skeleton. Equation (1) defines the total work of the program, where W_{all} is the cost of the skeleton.

$$W_{all} = costf * (length\ of\ the\ list) \quad (1)$$

The CAMS cost model is parametrised on both the cost in the skeletons and the continuation cost of the skeletons. Equation (2) shows the total work is the cost in the CAMS and the `costafter` of the CAMS.

$$W_{all} = costf * (length\ of\ the\ list) + costafter \quad (2)$$

4 Cost Calculus for \mathcal{J}'

To illustrate the concept of *costafter* i.e. the continuation cost of a program, this section gives a small language, $e ::= n \mid e+e$, where n is integer. Using expression $2+3$ as an example, the *costafter* of 2 can be calculated as $\frac{E \vdash_c 3 \$ c_3 \quad E \vdash_c + \$ c_+}{E \vdash_c 2 \triangleq (2+3) \mathcal{L} c_3 + c_+}$, where c_3 is the cost of 3, c_+ is the cost of “+”, and the *costafter* of 2 is $c_3 + c_+$. The semantic functions \vdash_c and \vdash_a produce the cost and *costafter* of expressions in the environment E .

The problem with this calculation is if there are two similar expression or one expression in two or more place in the program, it is difficult to identify the expression whose *costafter* needs to be calculated. To solve this problem, every expression in a program is given a unique number, which is called its *index*. Then the general three stages to calculate the *costafter* are: *indexing* the program, calculating the *cost* of expressions in the program, and calculating the *costafter* of the point to be required. Calculating the cost of expression is standard and use techniques similar to e.g. cost models in [11, 16, 20]. Calculate the *costafter*, which is to predict a continuation cost in a program, is novel.

A cost calculus has been built for language $\mathcal{J}[5]$, a subset of Jocaml. \mathcal{J} is a core functional language and readily able to describe non-trivial programs like matrix multiplication and ray tracing. To explain the principles, we introduce an even simpler language \mathcal{J}' , a subset of \mathcal{J} . Note that this cost calculus does not consider the type system and size system of the language.

4.1 Syntax of Language \mathcal{J}'

$e ::=$	k v $\text{fun } v \rightarrow e$ ee $e \text{ op } e$ $\text{map } ee$ $e (* e *)$ $\langle n, e \rangle$	expression constant variable lambda application operation map user cost index
$op ::=$	$+ \mid - \mid * \mid /$ $> \mid < \mid >= \mid <= \mid = \mid ! =$ $::$ $;$	operator arithmetical logical cons sequential composition

Figure 1: Syntax of \mathcal{J}'

Figure 1 shows the abstract syntax of \mathcal{J}' . To simplify the presentation it is assumed that variable names (v) in the

program are unique. \mathcal{J}' is a core functional language with two unusual expression. The *index* expression is presented because the whole program has been indexed. Costing recursive functions is undecidable. Thus to deal with the cost of recursive functions, *user costs* are introduced into \mathcal{J}' .

4.2 Indexing Expressions

Figure 2 shows an example of indexing an AST. In the figure, tree A is the original abstract syntax tree for expression e , $((a*b)*c)$, and tree B is the indexed abstract syntax tree. The indexed expression for e is: $\langle 5, (*, \langle 3, (*, \langle 1, a \rangle, \langle 2, b \rangle) \rangle) \rangle * \langle 4, c \rangle \rangle$.

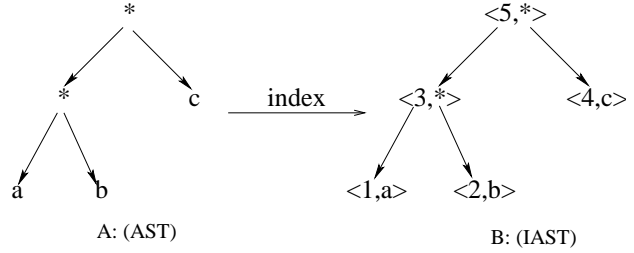


Figure 2: Indexing Example

4.3 Cost Semantics

Figure 3 shows part of the cost semantics of \mathcal{J}' . Semantic function $\vdash_c : env \rightarrow e \rightarrow (e * cost)$ takes the environment (env) and an expression (e), and returns a tuple of the expression and the cost of the expression ($cost$) under the environment E .

$$\frac{}{E \vdash_c k \$ 0} \quad (3)$$

$$\frac{}{\{v, c\} + E \vdash_c v \$ c + 1} \quad (4)$$

$$\frac{E \vdash_c e_1 \$ c_1 \quad E \vdash_c e_2 \$ c_2}{E \vdash_c e_1 \text{ op } e_2 \$ 1 + c_1 + c_2} \quad (5)$$

$$\frac{E \vdash_c e_1 \$ c_1 \quad E \vdash_c e_2 \$ c_2}{E \vdash_c \text{map } e_1 e_2 \$ c_1 * (\text{length } e_2) + c_2} \quad (6)$$

$$\frac{E \vdash_c e \$ c}{E \vdash_c \langle i, e \rangle \$ c} \quad (7)$$

.....

Figure 3: Partial Cost Semantics for \mathcal{J}'

Equation (3) infers the cost of a constant as 0 in environment E . Equation (4) shows that the cost of the value of variable (v), here c , has been stored in the environment, so the total cost of variable (v) is the cost to access the variable and the cost of the value of the variable, giving

$c + 1$. Equation (5) performs the cost of operation expressions ($e_1 \text{ op } e_2$). If the cost of e_1 is c_1 , and the cost of e_2 is c_2 then the cost of $e_1 \text{ op } e_2$ is $1 + c_1 + c_2$, where 1 is the cost for getting the operator. Equation (7) shows that the cost of index expression $\langle i, e \rangle$ is the cost of expression e (c). Figure 4 shows the progress of costing the expression in Figure 2.

$$\begin{aligned}
& \text{cost} \langle 5, (\langle 3, (\langle 1, a \rangle * \langle 2, b \rangle) \rangle * \langle 4, c \rangle) \rangle \\
& \Rightarrow \text{cost} (\langle 3, (\langle 1, a \rangle * \langle 2, b \rangle) \rangle * \langle 4, c \rangle) \quad (7) \\
& \Rightarrow 1 + \text{cost} \langle 3, (\langle 1, a \rangle * \langle 2, b \rangle) \rangle + \text{cost} \langle 4, c \rangle \quad (5) \\
& \Rightarrow 1 + (1 + \text{cost} \langle 1, a \rangle + \text{cost} \langle 2, b \rangle) + \text{cost} \langle 4, c \rangle \quad (7,5) \\
& \Rightarrow 1 + (1 + \text{cost } a + \text{cost} \langle 2, b \rangle) + \text{cost} \langle 4, c \rangle \quad (7) \\
& \Rightarrow 1 + (1 + (1 + c_a) + \text{cost} \langle 2, b \rangle) + \text{cost} \langle 4, c \rangle \quad (4) \\
& \Rightarrow 1 + (1 + (1 + c_a) + (1 + c_b)) + \text{cost} \langle 4, c \rangle \quad (7,4) \\
& \Rightarrow 1 + (1 + (1 + c_a) + (1 + c_b)) + (1 + c_c) \quad (7,4)
\end{aligned}$$

Figure 4: An Example of Costing in \mathcal{J}'

4.4 Continuation Costs

There are two approaches to calculate the cost of the continuations (*costafter*) in a program. One is translating the direct program into continuation passing style (CPS)[7] to obtain the continuation of the current expression, then calculating the cost of the continuation. Another approach is to pass the continuation cost directly rather than pass the continuation back to the current expression. There are two advantages of this better approach. First, we do not need to translate the program into CPS. Second, it is easier to pass an integer, the cost, back to the current expression than the execution state of the program e.g. the call stack or values of variables.

Figure 5 shows part of the *costafter* semantics. Semantic function $\vdash_a : env \rightarrow e \rightarrow e \rightarrow cost$ takes the environment (env) and two expressions and returns a cost i.e. the *costafter* of the first expression in the second expression. Equation (8) states that if expression e is equal to e'

$$\begin{aligned}
& \frac{e \equiv e'}{\text{E } \vdash_a e \leq e' \mathcal{L} 0} \quad (8) \\
& \frac{e \in e_1 \quad \text{E } \vdash_a e \leq e_1 \mathcal{L} c_1 \quad \text{E } \vdash_c e_2 \mathcal{L} c_2}{\text{E } \vdash_a e \leq (e_1 \text{ op } e_2) \mathcal{L} 1 + c_1 + c_2} \quad (9a) \\
& \frac{e \in e_2 \quad \text{E } \vdash_a e \leq e_2 \mathcal{L} c_2}{\text{E } \vdash_a e \leq (e_1 \text{ op } e_2) \mathcal{L} c_2 + 1} \quad (9b) \\
& \frac{}{\text{E } \vdash_a e \leq (e_1 \text{ op } e_2) \mathcal{L} 0} \quad (9c) \\
& \frac{\text{E } \vdash_a e \leq e_1 \mathcal{L} c}{\text{E } \vdash_a e \leq \langle i, e_1 \rangle \mathcal{L} c} \quad (10) \\
& \dots
\end{aligned}$$

Figure 5: *Costafter* Semantics

$$\begin{aligned}
& \text{costafter } b \text{ in} \\
& \langle 5, (\langle 3, (\langle 1, a \rangle * \langle 2, b \rangle) \rangle * \langle 4, c \rangle) \rangle \\
& \Rightarrow \text{costafter } b \text{ in} \\
& (\langle 3, (\langle 1, a \rangle * \langle 2, b \rangle) \rangle * \langle 4, c \rangle) \quad (10) \\
& \Rightarrow 1 + \text{costafter } b \text{ in} \\
& \langle 3, (\langle 1, a \rangle * \langle 2, b \rangle) \rangle + \text{cost} \langle 4, c \rangle \quad (9a) \\
& \Rightarrow 1 + \text{costafter } b \text{ in} \\
& (\langle 1, a \rangle * \langle 2, b \rangle) + \text{cost} \langle 4, c \rangle \quad (10) \\
& \Rightarrow 1 + (1 + \text{costafter } b \text{ in } \langle 2, b \rangle) + \text{cost} \langle 4, c \rangle \quad (9b) \\
& \Rightarrow 1 + (1 + \text{costafter } b \text{ in } b) + \text{cost} \langle 4, c \rangle \quad (10) \\
& \Rightarrow 1 + (1 + 0) + \text{cost} \langle 4, c \rangle \quad (8) \\
& \Rightarrow 1 + (1 + 0) + (1 + c_c) \quad (\text{Figure 4})
\end{aligned}$$

Figure 6: An Example of *Costafter*

($e \equiv e'$) then the *costafter* of e in e' is 0, where “ \equiv ” is syntax equality. Equations (9a), (9b), and (9c) define the *costafter* of e in operation expression ($e_1 \text{ op } e_2$). If e_1 contains e , then the *costafter* of e in ($e_1 \text{ op } e_2$) is the *costafter* of e in e_1 (c_1), plus the cost of e_2 (c_2) plus 1, which is the cost for getting the operator. Equation (10) shows that the *costafter* of e in index expression $\langle i, e_1 \rangle$ is the same as the *costafter* of e in expression e_1 .

Figure 6 shows an example of calculating the *costafter* of b in the indexed expression $\langle 5, (\langle 3, (\langle 1, a \rangle * \langle 2, b \rangle) \rangle * \langle 4, c \rangle) \rangle$.

5 Automatic Continuation Cost Analyser

5.1 Structure of the Automatic Continuation Cost Analyser

The cost calculus has been implemented as an automatic cost analyser in Jocaml. The analyser produces cost equations parametrised on program variables in context, and finds both cost in higher-order functions and the *costafter* of the higher-order functions, and converts higher-order functions to CAMSs with *costafter*s. As *costafter* is the cost of the continuations, the analyser is also called an *automatic continuation cost analyser*. The analyser takes programs in a subset of Jocaml with higher-order functions as input and outputs Jocaml AMPs with CAMSs.

Figure 7 shows the structure of the automatic continuation cost analyser. Parser takes Jocaml programs with higher-order functions, and outputs the ASTs. Indexer takes the AST and decorates every nodes, i.e. gives each expression a unique integer as an index, so the output is IAST. Indexer is an implementation of the index semantics in Section 4.2. Coster takes the IAST and outputs the *costafter* for each node. The coster has two parts: the first implements the cost semantics in Section 4.3 to calculate the cost of each expression, the second part implements the *costafter* semantics in Section 4.4 to calculate the *costafter* of each expression. The second part will use the costs

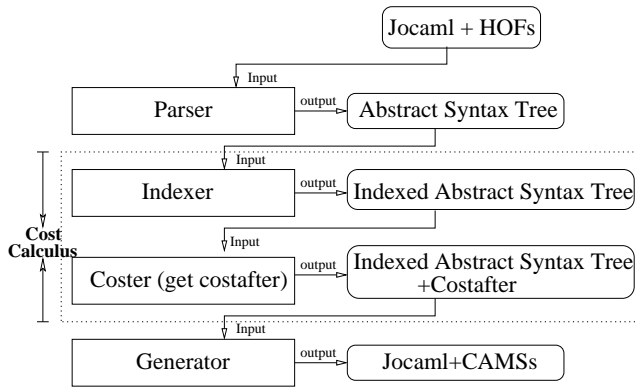


Figure 7: Automatic Continuation Cost Analyser

from the first part. The Generator generates a Jocaml AMP which has the same functionality as the original Jocaml program but using CAMSs instead of higher-order functions. For example, if the original program is `map f l`, the object program after the generator is `camap f l costf costafter`, where `costf` is the cost of `f` applied to the first element of `l`, which can be calculated using the cost semantics, and `costafter` is the cost after the `map` expression in the program, which can be calculated using `costafter` semantics.

5.2 An Example of the Cost Analyser

This section uses expression e ,

```
(map (fun x -> x+1) [1;2]);
(map (fun y -> y-1) [3;4])
```

as an example to explain how the analyser converts higher-order functions to CAMS. e has two sub-expressions e_1 , $(\text{map } (\text{fun } x \rightarrow x+1) [1;2])$, and e_2 , $(\text{map } (\text{fun } y \rightarrow y-1) [3;4])$. So e can be presented as $e_1; e_2$. To construct CAMSs, four issues should be considered: (1) the cost of $(\text{fun } x \rightarrow x+1)$, (2) the cost of $(\text{fun } y \rightarrow y-1)$, (3) the `costafter` of e_1 in e , and (4) the `costafter` of e_2 in e .

According to equations in Section 4.3, the cost of $(\text{fun } x \rightarrow x+1)$ can be calculated as $(1 + ((1+0)+0)) (\text{hd } [1;2])$, and simplifies to 2. The cost of e_2 is $((\text{fun } y \rightarrow (1 + ((1+0)+0)) (\text{hd } [3;4])) * (\text{length } [3;4]))$ and simplifies as 4. The `costafter` of e_1 in e is the `costafter` of e_1 in e_1 , which is 0, plus the cost of e_2 , plus 1. So the total `costafter` of e_1 in e is 5. Hence, the analyser converts e_1 to `(camap (fun x -> (x+1)) [1;2] (2) (5))`. Similarly, e_2 is converted to `(camap (fun y -> (y-1)) [3;4] (2) (1))`. So the output from the analyser is:

```
(camap (fun x -> (x+1)) [1;2] (2) (5) );
(camap (fun y -> (y-1)) [3;4] (2) (1) )
```

6 Evaluation

We have evaluated six automatically costed programs with single or sequentially composed CAMS against the corresponding AMS programs.

6.1 Single Higher Order Function Examples

The initial hypothesis is if there is only one higher-order functions in the AMP, the performances of the CAMS programs should be the same as the corresponding AMS programs. That is the AMPs should exhibit the same movement behaviour at the same moment and hence have the same execution times. Two single higher-order function AMPs have been tested: matrix multiplication and ray tracing.

Different size matrix multiplication have been executed to see if the CAMS programs have the same execution time as the corresponding AMS programs. The test environment has three locations with CPU speeds 534MHZ(`ncc1710`), 933MHZ(`jove`) and 1894MHZ(`lxtinder`). The loads on these three locations are almost zero, and both the CAMS and AMS programs are started on the first location. Figure 8 shows that the CAMS programs behave the same as the corresponding AMS programs. Similar results are gained for ray tracing AMPs.

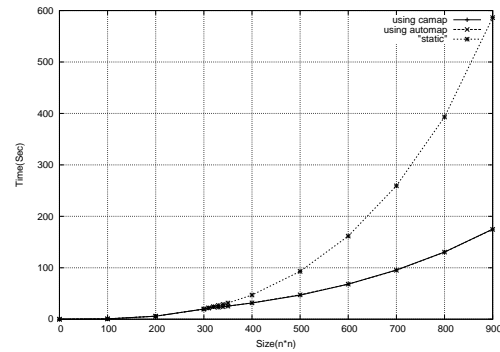


Figure 8: CAMS/AMS Matrix Mult. Exec. Time

6.2 Sequentially Composed Iterations

To investigate the performance of the skeletons when an AMP contains the sequential composition of several higher-order functions, four programs are compared: double matrix multiplication, invertible matrix, double ray tracing, and five matrix multiplications. The test environments in this section are the same as in Section 6.1.

The invertible matrix program takes two matrices and checks if they are invertible to each other. In this program

two matrices multiplication are performed sequentially, so there are two higher-order functions. Figure 9 compares the `camap` and `automap` invertible matrix. The `camap` AMP starts moving at matrix size of 230×230 , but the `automap` AMP starts moving still at 330×330 , because the `camap` considers the total remaining costs of the entire program but `automap` only consider the remaining costs in the function.

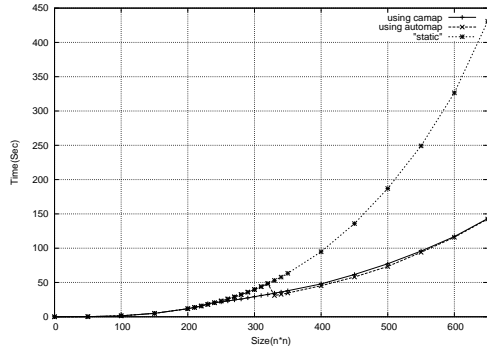


Figure 9: CAMS/AMS Inver Matrix Exec. Time

6.3 Varying Loads

When there is more than one higher-order function in AMPs, the CAMS programs may do more checks than the corresponding AMS programs. This is an advantage for reacting to the change of environment. The experiments in this section test the behaviour of CAMS and AMS programs when the loads of locations in the network are changed. Tests for two AMPs have been done: invertible matrix and five matrix multiplications. The tests are based on four locations, (1)`ncc1710`, (2)`jove`, (3)`lxtrinder`, and (4)`linux81` (2800MHz). At the beginning, the first three location are idle and the fourth location is heavily loaded with relative CPU speed of 56MHz. The AMPs are tested one by one. We start the AMPs on `ncc1710`, and they move to `lxtrinder` as expected. At the same time `linux81` finishes the work and becomes idle.

Figure 10 compares the execution time of CAMS to AMS invertible matrix programs. When the sizes of matrices are larger than 450×450 , the CAMS programs move to the faster location `linux81` again, but the corresponding AMS programs do not, as the `camap` in the CAMS programs do more checks than `automap` in the corresponding AMS programs. In this case, the CAMS programs may finish more quickly than the corresponding AMS programs. When the size of the matrix is larger than 450×450 but smaller than 500×500 , the CAMS programs is slower than the corresponding AMS programs, as the additional coordination time is bigger than the reduced execution time in

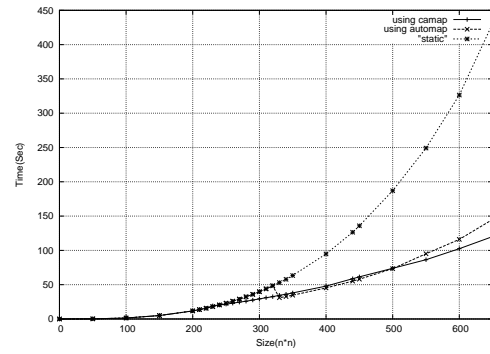


Figure 10: CAMS/AMS Inver. Matrix Exec. Time with Changing Loads

a faster location. When the size of the matrix is larger than 500×500 , the CAMS programs is faster than the corresponding AMS programs.

When the size of matrix is smaller than 450×450 , the CAMS programs do not move, because the cost of moving to another location is too big compared to the reduced execution time in a faster location. In this case, the results are similar to the result in Figure 9. Similar results are also obtained for five matrix multiplication AMPs.

6.4 Discussion

From the results above the following conclusion can be drawn: Firstly, if there is only one high-order function dominating the computation, CAMS programs reproduce the movement of the corresponding AMS programs. Secondly, if there is more than one higher-order functions, CAMS programs move with smaller size data than the corresponding AMS programs. Finally, the CAMS programs react to the change of environment more sensitively than the corresponding AMS programs.

7 Conclusion & Future Work

To deploy autonomous mobility effectively, it is necessary to know the cost of the program continuation. We have developed a cost calculus to estimate the costs for the continuation at arbitrary program points. This calculus includes three parts: indexing the abstract syntax tree, calculating the cost of each expression, calculating the continuation cost i.e. cost of expressions.

We have extended our AMS cost models to be parametrised on the continuation cost. Costed autonomous mobility skeleton (CAMSs) have been built, which not only encapsulate common patterns of autonomous mobility but take additional cost parameters. Measurements show that CAMS programs perform more effectively than AMS programs, because they have more accurate cost information.

Hence a CAMS program may move to a faster location when the corresponding AMS program does not.

For an autonomous mobile program, it is not necessary to calculate the continuation cost of each expression. We are interested in the continuation cost of higher-order functions. An automatic Jocaml cost analyser based on the calculus has been built, which calculates both cost in higher-order functions and the cost after of them, and converts these higher-order functions into CAMSs.

There are a number of areas for future work. Firstly, the current AMP experiments are performed on local area networks. We would like to generalise the AMPs architecture to large scale network e.g. WAN, Grid, etc. Secondly, we aim to build automatic resource driven mobility. We propose to investigate the application of a generic cost-based ethology to autonomous mobile multi-agent systems. Specifically, we would like to use evolved biological foraging strategies to better engineer scalable self-organising resource-location systems in large-scale dynamic networks. Finally, we aim to adapt the calculus to the more mainstream Java language.

References

- [1] J. Abawajy. Autonomic Job Scheduling Policy for Grid Computing. In *LNCS 3516*, pages 213–220, Germany, May 2005. ICCS 2005, Springer.
- [2] L. Cardelli. Abstractions for Mobile Computation. *Secure Internet Programming*, pages 51–94, 1999.
- [3] J. Cohen and C. Zuckerman. Two Languages for Estimating Program Efficiency. *Commun. ACM*, 17(6):301–308, 1974.
- [4] X. Y. Deng, G. Michaelson, and P. Trinder. Autonomous Mobility Skeletons. *Journal of Parallel Computing*, Volume 32, Issues 7-8:Pages 463–478 Algorithmic Skeletons, September 2006.
- [5] X. Y. Deng, G. Michaelson, and P. Trinder. Cost-Driven Autonomous Mobility. Technical Report MACS-TR-0051, School of Mathematical and Computer Sciences: Heriot-Watt University, May 2007.
- [6] X. Y. Deng, P. Trinder, and G. Michaelson. Autonomous Mobile Programs. In *IAT 2006 Main Conference Proceedings*, pages 177–186, Hong Kong, December 2006. IEEE Computer Society.
- [7] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 1992.
- [8] Institut National de Recherche en Informatique et en Automatique. *The JoCaml language beta release: Documentation and user's manual*, January 2001.
- [9] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.
- [10] Z. Kirli. *Mobile Computation with Functions*. PhD thesis, University of Edinburgh, LFCS: Division of Informatics, 2001.
- [11] H.-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, University of Glasgow, April 1998. Department of Computing Science.
- [12] D. Milojevic, F. Douglass, and R. Wheeler. *Mobility: processes, computers, and agents*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.
- [13] L. H. Ramshaw. *Formalizing the Analysis of Algorithms*. PhD thesis, Stanford University Department of Computer Science, 1979.
- [14] Recursion Software, Inc, 2591 North Dallas Parkway, Suite 200, Frisco, TX 75034. *Voyager User Guide*, May 2005.
- [15] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *LFP '94*, pages 65–78. ACM Press New York, 1994.
- [16] M. Rosendahl. Automatic Complexity Analysis. In *FPCA '89*, pages 144–156, Imperial College, London, UK, 1989. ACM Press New York.
- [17] T. Sekiguchi. JavaGo, May 2006. <http://homepage.mac.com/t.sekiguchi/javago/index.html>.
- [18] P. T. Tosic and G. A. Agha. Towards a Hierarchical Taxonomy of Autonomous Agents. In *IEEE SMC'2004*, pages 3421–3426, Hague, The Netherlands, October 2004. IEEE Xplore.
- [19] C. V. Travis Desell, Kaoutar El Maghraoui. Load Balancing of Autonomous Actors over Dynamic Networks. page 90268.1, 2004.
- [20] P. Wadler. Strictness Analysis Aids Time Analysis. In *POPL '88*, pages 119–132, San Diego, California, USA, 1988. ACM Press.
- [21] B. Wegbreit. Mechanical Program Analysis. *Commun. ACM*, 18(9):528–539, 1975.
- [22] M. Wooldridge. Agent-Based Software Engineering. *IEE Proceedings Software Engineering*, 144(1):26–37, 1997.