Communicating Process Architectures 2009
Peter Welch, Herman Roebbers and Tobe Announced (Eds.)
IOS Press, 2009

© 2009 The authors and IOS Press. All rights reserved.

Auto-Mobiles: Optimised Message-Passing

Neil C. C. BROWN

Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, England.

neil@twistedsquare.com

Abstract. Some message-passing concurrent systems, such as occam 2, prohibit aliasing of data objects. Communicated data must thus be copied, which can be time-intensive for large data packets such as video frames. We introduce automatic mobility, a compiler optimisation that performs communications by reference and deduces when these communications can be performed without copying. We discuss bounds for speed-up and memory use, and benchmark the automatic mobility optimisation. We show that in the best case it can transform an operation from being linear with respect to packet size into constant-time.

Keywords. Message-passing, Mobility, Optimisation

Introduction

Aliasing in concurrent systems can lead to data race-hazards when two or more concurrently executing processes are able to modify the same data without restriction. The non-deterministic order of the modifications affects the program's subsequent behaviour. Process-oriented programming eliminates this aliasing of mutable objects in favour of message-passing: self-contained processes communicate data between each other without any sharing.

In implementations of the occam 2 programming language, all communicated data is copied, which can be expensive for large messages (for example, video frames). The concept of mobility was introduced in occam-π [1], which (amongst other things) allows data to be mobile – that is, held by reference [2]. Aliasing is prevented by a transferral or *movement* semantics that guarantees that the reference is never held by more than one variable. Movement semantics state that after an assignment or communication, the source variable becomes undefined, because the reference has *moved* to the destination variable. Communication of mobile data is much more efficient, as only the reference to the data need be communicated. (In this paper we assume a common, uniform memory architecture.)

Data mobility increases the burden on the programmer by requiring the addition of annotations to designate data as mobile, and requiring the use of the simple but unusual movement semantics. Thus, the efficiency of mobile data comes at a price to the programmer and can be a barrier to learning the language for newcomers.

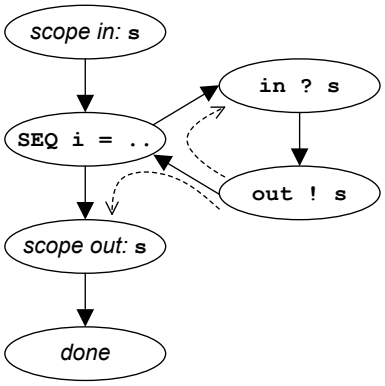
In this paper we present a technique for gaining the efficiency advantages of data mobility without any deviation from the standard copy semantics and unadorned syntax of occam 2. We call this technique *automatic mobility*. Automatic mobility is a compiler optimisation for occam 2 programs that requires no change to the original code and neither syntactic nor semantic changes to the occam 2 language. Automatic mobility also generalises to other message-passing languages; it is not a technique specific to occam 2.

```
PROC id (CHAN LARGE.DATA in,
         CHAN LARGE.DATA out)
  LARGE.DATA s:
  SEQ i = 0 FOR 1000
    SEQ
      in ? s
      out ! s
:
```

(a)

```
PROC id (CHAN MOBILE LARGE.DATA in,
         CHAN MOBILE LARGE.DATA out)
  INITIAL MOBILE LARGE.DATA s IS
            MOBILE LARGE.DATA:
  SEQ i = 0 FOR 1000
    SEQ
      in ? s
      out ! CLONE s
:
```

(b)



(c)

```
PROC id (CHAN MOBILE LARGE.DATA in,
         CHAN MOBILE LARGE.DATA out)
  INITIAL MOBILE LARGE.DATA s IS
            MOBILE LARGE.DATA:
  SEQ i = 0 FOR 1000
    SEQ
      in ? s
      out ! s
:
```

(d)

Figure 1. An example of converting a variant of the occam 2 identity process (a), first into its exact mobile occam-π semantic equivalent (b). The flow graph of the process (c) is then used to turn this into the auto-mobile version (d), based on the observation that the data is not needed inbetween the output and being overwritten. This can be observed by following the dotted lines in the flow graph (c) which lead to an overwrite via an input or to going out of scope.

1. Automatic Mobility

Automatic mobility comprises **two** compiler transformations. The **first *mobilisation* transformation** is to store all large[1] data items in the program's shared heap, rather than the usual occam 2 implementation of storing them in the workspace/stack. If these arrays are allocated at the point of declaration, freed when they go out of scope, and copied on assignment and output, the occam 2 semantics are directly preserved by this transformation. This transformation can be thought of as turning all data mobile, but always `CLONE`ing[2] it rather than moving it. An example of converting a slightly modified identity process is given in figures 1a (the original) and 1b (the transformed version).

The key insight of automatic mobility is that once all the inputting processes are expecting to receive a reference, the outputting process has the option to either allocate a new copy of the data and send that (i.e. to send a `CLONE`), or to send the original reference. If the outputting process will not use the data again after the output, it can therefore send the original reference and discard it. This is the **second transformation: *the copy/move decision***. We continue the earlier example, with the control-flow graph in figure 1c and the final transformed version in figure 1d. The details of the move/copy decision are discussed in section 2.

We will consider in this paper how to mobilise arrays in the most general case. Records can be conceived of as arrays (albeit heterogeneous), and other data (e.g. integers) can be considered to be an array with one element. In occam 2, arrays are the typical way to store large amounts of data.

[1] What we determine to be large will be guided by the benchmarks in section 8.

[2] `CLONE` is the occam-π syntax for making a copy of an item of mobile data.

2. Mobility Rule

Our rule for deciding whether an outputting process should move or copy is simple: **An array can be moved iff no array element can possibly be subsequently read from, before being overwritten or the array going out of scope. Otherwise, it should be copied.** We will first consider implementing the mobility rule for operations that involve the entire array such as reading an array from a channel; operations on individual elements are discussed in section 5.

The first step to implementing the mobility rule is to generate a control-flow graph for the occam PROCedure. This graph is processed to calculate two sets for each node: the **sequentially-later** set of variables, and the **in-parallel** set of variables.

Discovering information about **sequentially-later** uses of a variable involves an iterative data-flow algorithm [3, pp 231]. A set of variables is calculated at each node by taking the union of all the variables read at that node and variable-sets from future-sequential nodes, minus the set of all variables written to at that node. The algorithm iterates to a fixed point.

The occam compiler enforces a CREW (Concurrent-Read, Exclusive-Write) safety rule, but this permits concurrent reads. Therefore it is possible that even though a variable will not be read from by code that is sequentially later in the flow-graph, it may be read from by a node in parallel that happens to execute later. We deal with this by also finding all nodes that are **in parallel** with each other (trivial from the abstract syntax tree of a program) and recording the read-from variables.

Determining whether an array is read-from after it is sent on a channel is a matter of examining the two sets (sequentially-later and in-parallel reads) at the corresponding node. If the variable is in neither set, it can be moved. If it is in either or both sets, it must be copied.

The analysis for the move/copy decision is performed solely by examining the writer process for a particular channel-end. No information is known nor assumed about the reading process. This means that the analysis is robust in the face of separate compilation, providing all compilation units have the automatic mobility feature enabled, as the mobilisation transformation must have been performed on the reader.

3. Allocations

For occam 2 the semantics are that an array is available from the point of its definition, with undefined contents, and continues to be available for reads and writes until the last use. This can be emulated with dynamic arrays by allocating an array at the point of definition and following the automatic mobility rules.

Allocating an array at the point of definition can be inefficient, however. For example, consider:

```
1  [64]INT array:
2  SEQ
3    in ? array
```

The array would be allocated on line 1, and then immediately deallocated when the new array is received from the channel on line 3. To avoid this, we use the control-flow graph. **If an array is not used before it is written-to in its entirety (for example, by reading from a channel), the array does not need to be allocated. In all other cases, it must still be allocated.**

4. Examples

In this section we present several example processes and explain the automatic mobility transformation's effect on the processes.

4.1. Identity Process

The simplest example is that of the identity process that forwards values from one channel to another:

```
 4  PROC id (CHAN [64]INT in, CHAN [64]INT out)
 5    [64]INT s: -- Definition outside the loop,
 6    WHILE TRUE
 7      [64]INT s: -- or inside the loop
 8      SEQ
 9        in ? s
10        out ! s -- becomes a move
11  :
```

The flow analysis is subtly different if the array is defined outside the loop, but the result is the same. With the definition inside the loop, `s` is analysed as never being used again after its output. With the definition outside the loop, `s` is analysed as not being used again before being completely overwritten. Either way, `s` is moved during the output, and will also not be allocated at the point of definition (in both cases, it will be overwritten entirely by the input). The same results apply to transformation processes, such as this amplifier process:

```
12  PROC amp (CHAN [64]INT in, VAL INT factor, CHAN [64]INT out)
13    WHILE TRUE
14      [64]INT s:
15      SEQ
16        in ? s
17        SEQ i = 0 FOR 64
18          s[i] := s[i] * factor
19        out ! s -- becomes a move
20  :
```

Automatic mobility is not just for communications (although that is the most common case), but also works for assignments. Thus if we reimplement the identity process as follows:

```
21  PROC id.2 (CHAN [64]INT in, CHAN [64]INT out)
22    WHILE TRUE
23      [64]INT s, t:
24      SEQ
25        in ? s
26        t := s -- becomes a move
27        out ! t -- becomes a move
28  :
```

Both the assignment and the output become moves, and thus this process is no more expensive than the original identity process, even though there is an extra assignment of the array.

4.2. Delta Process

A delta process is one that reads in data from one channel and sends out the same data on several channels. Its definition is:

```
29  PROC delta (CHAN [64]INT in, []CHAN [64]INT out!)
30    WHILE TRUE
31      [64]INT s:
32      SEQ
33        in ? s
34        PAR i = 0 FOR SIZE out
35          out[i] ! s -- becomes a copy
36  :
```

This will result in cloning for all the outputs, because they are in parallel with each other. Knowing which process is the last to output (and could thus perform a move) is difficult when the outputs are in parallel – but this would be a worthwhile goal, especially if there are only two output channels, as is commonly the case. With two output channels, two clones could become one clone and one move. There are several possible solutions. One is to make the outputs sequential, at which point the compiler could unroll the last loop iteration and turn that into a move. Another solution would be to pull out the copying of the data to outside the PAR[3]:

```
37  PROC delta (CHAN [64]INT in, []CHAN [64]INT out!)
38    [SIZE out][64]INT ss:
39    WHILE TRUE
40      SEQ
41        in ? ss[0]
42        SEQ i = 1 FOR (SIZE out) - 1
43          ss[i] := ss[0]
44        PAR i = 0 FOR SIZE out
45          out[i] ! ss[i]
46  :
```

For the compiler to spot the necessary optimisations, it must be able to handle the array indexing to spot that the assignments (line 43) must be clones, but that the communications (line 45) can be moves. In this case, it is straightforward as the array is not used again (in whole nor in part) after the communications before an overwrite. It is possible that this transformation could be performed by the compiler, where it spots a delta-like pattern in a PROC.

4.3. Merging Process

A merging process is one that reads in data from several channels, and somehow turns them into a single output using a folding operation. For example, this sum process zips together many arrays using addition:

```
47  PROC sum ([]CHAN [64]INT ins, CHAN [64]INT out)
48    WHILE TRUE
49      [64]INT acc, s:
50      SEQ
51        SEQ i = 0 FOR 64
52          acc[i] := 0
53        SEQ j = 0 FOR SIZE ins
54          SEQ
55            ins[j] ? s
56            SEQ i = 0 FOR 64
57              acc[i] := acc[i] + s[i]
58        out ! acc
59  :
```

[3]Note that the dynamic array dimension is not legal occam 2.1, but see the appendix.

This process receives many arrays, and will deallocate them all before the next is read. The new accumulated total will be allocated on each iteration of the loop and sent out with a move (as with the identity process, this is true regardless of whether the accumulator is declared inside or outside the loop).

There is an opportunity to prevent this allocation, by re-using one of the incoming arrays. This can be done as follows (for brevity, we ignore the possibility that `ins` is size zero):

```
60  PROC sum ([]CHAN [64]INT ins, CHAN [64]INT out)
61    WHILE TRUE
62      [64]INT acc, s:
63      SEQ
64        ins[0] ? acc
65        SEQ j = 1 FOR (SIZE ins) - 1
66          SEQ
67            ins[j] ? s
68            SEQ i = 0 FOR 64
69              acc[i] := acc[i] + s[i]
70        out ! acc
71  :
```

Note that this is also more efficient than the original even without automatic mobility, as it avoids the initialisation of `acc` with zeroes, and avoids the first loop execution of additions.

5. Individual Elements

We have so far considered how to implement automatic mobility for entire arrays. For example, we can determine that the output should be a movement in, for example:

```
72  SEQ
73    out ! array.x
74    array.x := some.other.array
```

We will now consider the code:

```
75  SEQ
76    out ! array.x
77    SEQ i = 0 FOR SIZE array.x
78      array.x[i] := some.other.array[i]
```

One option would be to have an optimisation rule to transform lines 77 and 78 into an assignment of the entire array. However, for now we will consider the general principle of what must be done with individual array accesses such as these. There are two options for the rules we could adopt to transform this code with automatic mobility.

5.1. Copying

The code could be transformed by changing the output to a copy when individual elements are written-to after the output:

```
79  SEQ
80    out ! CLONE array.x
81    SEQ i = 0 FOR SIZE array.x
82      array.x[i] := some.other.array[i]
```

This comprises one allocation (during the `CLONE` on line 80), and two copies of the array (one as part of the `CLONE` on line 80, one from the loop on lines 81 and 82).

5.2. Moving and Allocating

If the individual array elements needed to be read afterwards we would have to make the output a copy as explained. However, an alternative rule would be to make the output a move, and then allocate a fresh array if the array elements are only written-to.

```
1  SEQ
2    out ! array.x
3    array.x := MOBILE ARRAY.X.TYPE
4    SEQ i = 0 FOR SIZE array.x
5      array.x[i] := some.other.array[i]
```

This comprises one allocation (line 3), and one copy (lines 4 and 5), and is therefore more efficient than the first option. This rule can only be applied if all the array elements are written-to before being read-from (and thus none of the data present before the output is required after the output). This relies upon the compiler being able to detect that no elements of the array are read from before being written to.

5.3. Dynamic Index Reasoning

Detecting whether array elements are read-from before being written-to in the presence of dynamic indices can be done with the Omega Test [4]. To explain this, we will consider a more subtle problem:

```
1  SEQ
2    out ! array.x
3    SEQ i = 0 FOR (SIZE array.x / 2)
4      array.x[2*i] := some.other.array[2*i]
5    SEQ j = 0 FOR SIZE array.x
6      IF
7        j \ 2 == 1
8          array.x[j] := array.x[j-1]
9        TRUE
10         SKIP
```

The portion of the array written-to by the first loop can be described by the index $2i$ and the compiler-derived inequalities:

$$0 \leq i < \texttt{SIZE array.x / 2}$$

The portion of the array read-from in the second loop can be described by the index $j-1$ and the compiler-derived inequalities:

$$0 \leq j < \texttt{SIZE array.x}$$
$$j \ \texttt{\textbackslash}\ 2 = 1$$

The modulo expression is transformed into further inequalities, and the whole system can then be solved by the Omega Test to show that there is no solution to the equation $2i \neq j-1$ that satisfies the other equations, i.e. that none of the read-from portion in the second loop is not written-to by the the first loop.

6. Efficiency Bounds

We can examine the bounds for the speed-up with automatic mobility. Given the number of communications of same-size data in a process network with consistent communication behaviour, we can determine what proportion, M, are mobile ($0 \leq M \leq 1$).

If we assume that copying data is expensive, but that allocation of memory is potentially cheap enough to be negligible, the maximum speed-up of communications in the mobile version is:

$$\frac{1}{1-M} \tag{1}$$

Therefore, if no communications are mobile the maximum speed-up is $1\times$ (i.e. no speed-up), if $\frac{3}{4}$ of the communications are mobile the maximum speed-up is $4\times$, and if all of the communications are mobile, the maximum speed-up is infinite (i.e. unbounded).

This gives us an optimistic maximum speed-up bound. However, memory allocation is unlikely to be negligible. In fact, the improvement in performance of the automatic mobility technique is reliant on memory allocation being cheaper than copying. We can instead assume that the time taken to allocate a block of memory of size S is $a(S)$ and label the time to copy a block of the same size as $c(S)$. Under automatic mobility, the time for copying operations is $a(S) + c(S)$, while the time for movement operations is a constant V. The speed-up of automatic mobility (all data being size S) can thus be more accurately estimated as:

$$\frac{c(S)}{MV + (1-M)(a(S)+c(S))}$$

The denominator is the time for the moves multiplied by the proportion of moves (MV) plus the time for allocation and copying multiplied by the proportion of copies ($(1-M)(a(S)+c(S))$) – the numerator is the time for copying everything.

If we assume that C (time for communicating a reference) approximates zero, and consider when this speed-up factor will be greater than one, we can rearrange to:

$$\frac{M}{1-M} > \frac{a(S)}{c(S)} \tag{2}$$

So automatic mobility gives a speed-up iff the factor by which the mobile communications outweigh the non-mobile is greater than the saving in time of allocation over copying. If only $\frac{1}{10}$ of communications in a system were mobile, automatic mobility would only be worthwhile if allocating blocks was at least 9 times faster than copying them. If at least half of the communications in a system are mobile, automatic mobility will give a speed-up if allocation is faster than copying.

7. Memory Usage

One problem with occam 2's static allocations, in the absence of automatically growing stacks, is that memory must be allocated in a process's stack/workspace ready for the maximum memory use of a process rather than the current use. Thus an identity process that copies values of 1MB will always have 1MB of stack allocated to it, even if it is idle for most of its lifetime. In contrast, the same process under automatic mobility will have a small stack, and the space for the data is only allocated if the identity process is currently holding a data packet.

More generally, data in occam 2 lives only in the stacks of processes. Data is never held in a channel; the synchronous communications are simply a direct copy from one process's stack to another's. Thus if P processes all have a stack variable of size S, the total memory allocated is slightly larger than PS.

In an automatic mobility setting, all large data lives in the heap, not on the stack. If D items of data of size S are allocated in the heap, the memory use is slightly larger than DS. It can be seen that in a non-deadlocking system with synchronous channels, the number of

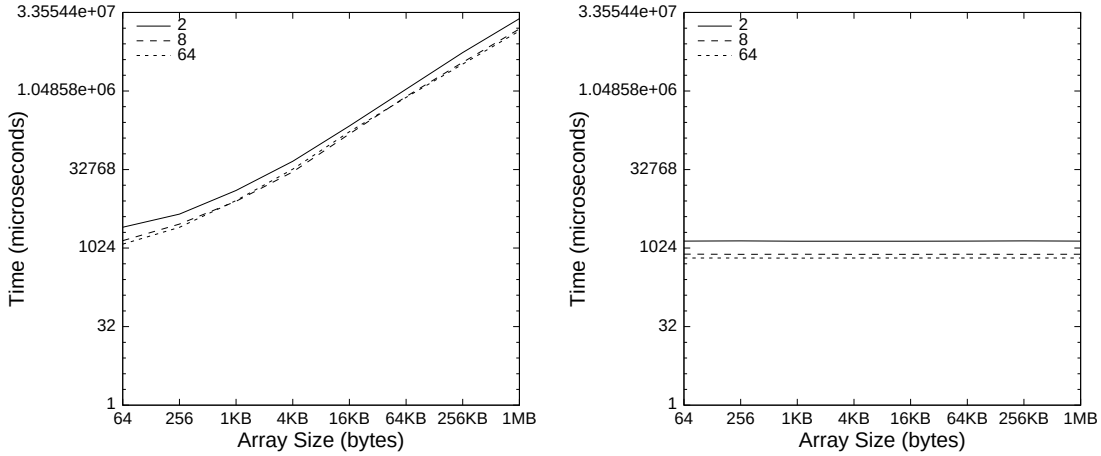


Figure 2. Timings for the ring benchmark, with N communications in a ring. Each line illustrates timings for a different value of N, where the overall number of communications is held constant. The X-axis is the size of each data packet, and the Y-axis is time; both axes are logarithmic. Times are shown for the original copying version (a) and the mobile version (b).

data items allocated, D, must be less than or equal to the number of processes in the system that might be communicating those values, P. Since $D \leq P$ and S is positive, $DS \leq PS$, and thus the system will always have better or equal memory use under automatic mobility than with fixed stacks.

8. Benchmarks

To investigate the speed-up that automatic mobility can provide, we benchmarked several programs on an AMD Athlon 64 3000+ with 512KB cache, using our Tock compiler to generate CCSP code compiled by GCC.

8.1. Ring

To estimate the maximum speed-up that could be achieved through automatic mobility, we first benchmark simple rings. Each pipeline has one prefix and one recorder, connected by $N-2$ identity processes. The prefix process sends out one array and then acts like an identity process. The recorder acts like an identity process, but times the running of the network. In a classic setting, there will be N copy-communications of the data per iteration. In an automatic mobility setting, there will be N move-communications. Thus the difference in times will reveal the difference in cost between the two communication types.

We benchmarked the system with several sizes of data and lengths of pipeline. The results are given in table 1a and depicted in figure 2. It can easily be seen from the graph that the mobile version operates in constant time (w.r.t. to the data size) whereas the copy version is linearly proportional to the amount of bytes being communicated (once the cache size is exceeded). Thus, the speed-up is also linearly proportional to the amount of bytes being communicated. Referring to our earlier measure in equation 1, here $M = 1$, and thus the potential speed-up is unbounded. For this benchmark, automatic mobility provides a benefit even at the lowest size of 64 bytes per data packet.

8.2. Twin Pipeline

To investigate speed-up in a program involving copying and movement, we benchmarked a program with two pipelines. The first pipeline has a producer, followed by N delta processes.

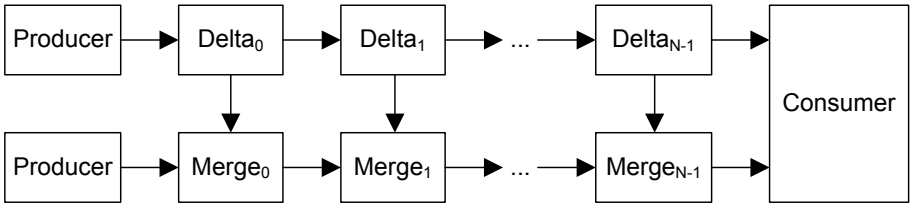


Figure 3. The twin pipeline benchmark: two producers, a dual consumer, and N connected delta and merge processes inbetween.

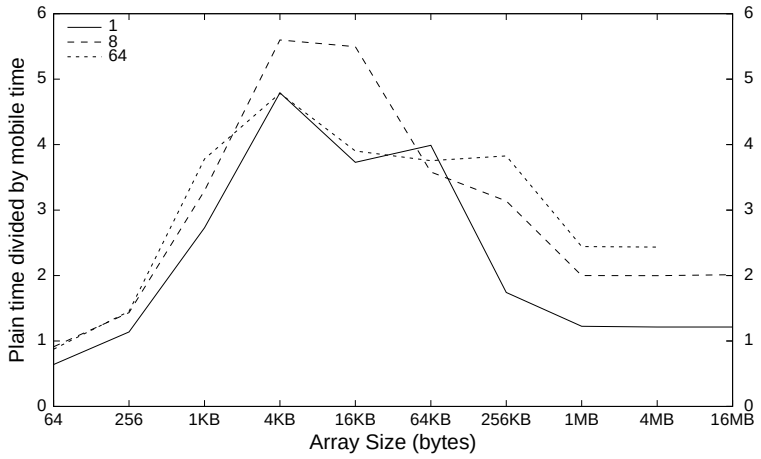


Figure 4. Speed-up factors for the twin pipeline benchmark, depicted in figure 3. Each line illustrates the speed-up for a different value of N, where the overall number of communications is held constant. The logarithmic X-axis is the size of each data packet and the Y-axis is speed-up (a factor greater than one indicates the mobile version is faster). The data point for a pipeline of 64 with 16 megabyte packets could not be carried out due to address-space limitations.

The second pipeline has a producer, followed by N merge processes. The delta processes are connected to the merge processes. The two pipelines both feed into a recorder process responsible for timing. The benchmark is depicted in figure 3.

There will be $2 + 3N$ communications in the benchmark, of which $2 + N$ will be copies and $2N$ will be moves. In the case where $N = 1$ (one delta and one merge process), $M = \frac{2}{5}$ in equation 1, and thus the maximum speed-up is $\frac{5}{3}$. As N increases, the proportion M will tend to $\frac{2}{3}$. Therefore the maximum speed-up for larger pipelines is 3.

We benchmarked the system with several sizes of data and lengths of pipeline. The results are given in table 1b and depicted in figure 4. It can be seen that for packet sizes up to 256KB, the speed-up is slightly chaotic, and larger than our theoretical upper bound! This is due to the processor cache, which can be better taken advantage of in the mobile version, due to the smaller amounts of data allocated in the program. For sizes of 1MB upwards (i.e. that are larger than the cache) the speed-up is stable. In this benchmark, automatic mobility provides a benefit for sizes of 256 bytes and upwards.

8.3. Occam Audio Kit

The occam audio kit (oak) is a library of useful processes for performing audio generation and manipulation in occam 2, written by Adam Sampson. Almost all the processes use channels of blocks of audio data. Aside from common utility processes such as delta and the sum-like mixer process, there are processes for generating sine waves, process networks for performing simple feedback effects and amplifier processes.

The oak library thus represents a real use case of communicating occam arrays, in a stream-processing setting. It is not a purely linear pipeline, and has diverging and merging

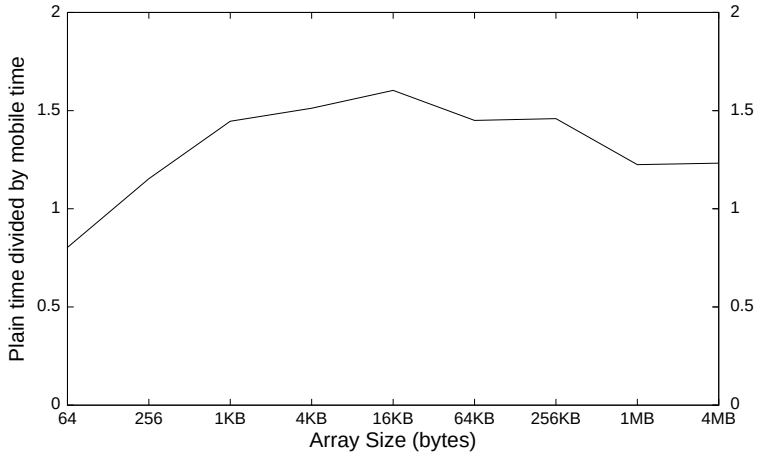


Figure 5. Speed-up factors (Y-axis) for the occam audio kit (oak) benchmark, timing a fixed number of blocks with varying block sizes (the logarithmic X-axis). A speed-up factor greater than one indicates that the mobile version is faster.

processes. We benchmark a music-generating program, with the final passing-to-hardware step replaced with a timing process, and some of the more expensive floating-point operations removed to avoid a confound.

Profiling revealed that each iteration has 55 communications, of which 11 are from producer processes (and thus involve allocations and copies), and a further 10 of which are copy-communications as part of a delta process. The remaining 34 are all move-communications, meaning that M is $\frac{34}{55}$.

The results of the benchmark are given in table 1c and graphed in figure 5. This benchmark, unlike the other two, features a computational component alongside the communication, and as would be expected, the speed-up is fairly low. However, the speed-up is still larger than one; it does improve the speed of the program, for packets of 256 bytes and larger.

9. Alternative Approaches

The analysis is currently performed on individual procedures. This means that pulling out common code into sub-procedures can interfere with the analysis. For example, this modified identity procedure is currently not optimised as the normal identity procedure is:

```
 1  PROC send (CHAN [64]INT out, VAL [64]INT x)
 2    out ! x
 3  :
 4
 5  PROC foo (CHAN [64]INT in, CHAN [64]INT out)
 6    [64]INT x:
 7    WHILE TRUE
 8      SEQ
 9        in ? x
10        send(out, x)
11  :
```

In future it would be better to perform whole-system analysis, which could avoid this problem – and also allow the optimisations to take into account the behaviour of the reader, not just the writer (a more complex topic).

An additional possibility to the approach described here is the use of copy-on-write references. Instead of sending a reference on a channel that the reader owns (as in this paper), some run-time support could be added to support the sending of read-only references. If

the reader needs to write to the data, it must make a modified copy of the data. This could potentially allow even less copies than the current conservative approach, as copies would only be made when actually necessary at run-time.

The cost of copy-on-write references is that co-ordination will also be required in the run-time to destroy data at the right time. Once a read-only reference has been created to a piece of data (in addition to the original reference), there must be some way of ensuring that the data is not destroyed until *all* the references to it have been overwritten or gone out of scope. In a concurrent system, the cost of this coordination can be high as it must involve locks or atomic operations. Thus we have chosen the simplest approach, that relies only on static analysis and not on costly run-time support.

9.1. Linked Lists

This paper has focused on simple blocks of data, such as arrays. An alternative data structure is linked lists. In languages without automatic run-time memory management, linked lists have $O(n)$ preserving concatenation (copying the two lists into a new list, leaving the old lists intact) but $O(1)$ destructive concatenation (transferring the two lists into a new list, and joining the head of one to the tail of the other) if both head and tail pointers are maintained.

Consider this pseudo-code for occam with linked-lists:

```
1  PROC merger (CHAN LIST INT inA, CHAN LIST INT inB, CHAN LIST INT out)
2    WHILE TRUE
3      LIST INT a, b:
4      SEQ
5        PAR
6          inA ? a
7          inB ? b
8       out ! (a ++ b)
9  :
```

With a standard implementation, this process would recieve the two lists, concatenate them in an $O(n)$ operation and send out this new copy on the channel. Using our automatic mobility optimisation, this $O(n)$ concatenation could be transformed into an $O(1)$ destructive concatenation, destroying `a` and `b` because they are not used again after the output. As concatenation is a very common operation on linked lists, it could be that automatic mobility is more beneficial for linked lists than it is for arrays.

10. Conclusions

We have introduced automatic mobility, an optimisation for concurrent message-passing systems. Under automatic mobility, data items of 256 bytes or larger are allocated on the heap. They are communicated by reference wherever possible, and by cloning (allocating a new copy) otherwise. This should typically provide speed-up of 1–2×, but in the ideal case it can turn a program that is linear in the size of data being communicated into one that executes in constant time.

Automatic mobility requires no changes to occam 2 programs, and is simply an optimisation flag in our Tock compiler. Wherever it provides a speed-up, it should be enabled and has no disadvantages. However, automatic mobility is not supported on embedded systems that lack support for dynamically allocating memory.

Based on our results, we expect that for most programs automatic mobility will be faster than the original application. The speed-up is particularly dependent on the speed of the memory allocator, so fast memory allocators would be worth investigating in future. Automatic mobility also has the potential to reduce the overall memory allocation for a program.

Acknowledgements

The author is grateful to the anonymous reviewers for their comments on this paper, and also to Peter Welch, who supported this idea from the outset.

Dynamically-Sized Arrays

The occam language was originally designed for the Transputer hardware. Each process had a statically allocated workspace. The size of the workspace needed for an occam process could be determined at compile-time. This was possible because all array bounds were constant. For embedded applications, this is still a useful feature of occam programs, but for modern desktop or server hardware this is a cumbersome, prohibitive restriction.

The occam-π language allows dynamic arrays through mobiles that are allocated on the heap. With our new automatic mobility transformation that also stores arrays on the heap, we may offer dynamically sized arrays. The problem is no longer one of implementation, but instead one of language design, which is beyond the scope of this paper.

Algorithms Summary

The first step in auto-mobilising a program is to convert all data items beyond a threshold size (we recommend 256 bytes, based on benchmarks) to being mobile. They should (at this point) become allocated at the point of declaration, and all the communications should become clone communications. All the subsequent steps refer solely to these mobilised variables.

The next step is to perform program analysis. A control-flow graph must be derived from the program. This is then used as part of a backwards iterative data flow algorithm [3, pp 231]. Each node begins with an associated empty set of variables. The algorithm then processes each directly connected node pair A and B (where A is followed by B). The new value for A is the union of three sets: its old value, B's current value and all variables read from at B, minus the set of all variables written to at B. The algorithm repeatedly processes all directly connected node pairs until the values for the nodes no longer change. The resulting value for each node is the set of all values that are read afterwards. This is the sequentially-later set; the program is also analysed to form sets of variables used in-parallel (trivial from the abstract syntax tree) for each node. Also generated is a used-before-overwrite set which is almost identical to the sequentially-later set but the union of three things becomes a union of four – it also adds any variables that are partially written to at B.

The next step is to process the declarations. Each declaration is checked against the used-before-overwrite set at that node. If the variable is in the set, it must remain allocated at the point of declaration. If it is not in the set (and thus is entirely overwritten before being accessed) the allocation of memory can be removed in favour of leaving it undefined (i.e. a null reference).

Finally, each output and assignment where the source (right-hand side) is a mobile variable is checked. If that source variable is in either the in-parallel or sequentially-later sets at that node, it remains as a clone. If it is in neither set, the output/assignment is modified to become a movement rather than a clone.

References

[1] Peter H. Welch and Fred R. M. Barnes. Communicating Mobile Processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, 2005.

[2] Fred R. M. Barnes and Peter H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In *Communicating Process Architectures 2001*, pages 243–264. IOS Press, September 2001.

[3] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, 1997.

[4] William Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

Table 1. Times for the (a) ring benchmark, (b) twin benchmark and (c) oak benchmark, showing the means and standard deviations (S.D.) for 30 runs of each condition, measured in milliseconds, given to 4 significant figures, for each number of processes N and packet size. Note: KB=2^{10}, MB=2^{20}.

N	Size (bytes)	Plain Mean	Plain S.D.	Mobile Mean	Mobile S.D.
2	64	2564	94.34	1394	15.15
2	256	4586	20	1405	61.15
2	1KB	13020	108.1	1389	8.678
2	4KB	47190	257.3	1389	8.161
2	16KB	223900	551.8	1389	8.257
2	64KB	1134000	2119	1395	22.26
2	256KB	5722000	61350	1404	29.05
2	1MB	25570000	67420	1391	10.12
8	64	1419	82.62	784.3	26.86
8	256	2948	39.79	778.2	6.344
8	1KB	8066	28.88	780.7	13.21
8	4KB	29600	257.7	778	5.776
8	16KB	157300	509.4	778.2	6.023
8	64KB	818300	6986	780.2	7.383
8	256KB	3716000	18510	778.5	7.154
8	1MB	16190000	31240	779.7	7.203
64	64	1216	40.58	660	7.532
64	256	2554	249.5	659.5	7.907
64	1KB	8341	15.37	658	7.118
64	4KB	33300	24.83	660	6.411
64	16KB	173600	846.9	660.4	7.297
64	64KB	788800	1662	659.3	8.559
64	256KB	3429000	12320	660.5	7.415
64	1MB	14830000	23350	660.9	8.138

(a)

N	Size (bytes)	Plain Mean	Plain S.D.	Mobile Mean	Mobile S.D.
1	64	636.7	27.56	993.3	16.74
1	256	1237	12.68	1087	11.43
1	1KB	4148	15.26	1520	12.75
1	4KB	15310	441.8	3195	29.05
1	16KB	65710	186.7	17620	280.6
1	64KB	325100	689.7	81480	1360
1	256KB	1668000	8667	957900	3764
1	1MB	6830000	13660	5581000	59210
1	4MB	27360000	41890	22540000	254100
1	16MB	109500000	131200	90240000	1039000
8	64	453.3	243.1	497	9.471
8	256	770.6	8.01	537.5	10.1
8	1KB	2385	75.21	723.6	14.9
8	4KB	10180	512.2	1818	8.528
8	16KB	43920	350.3	7991	124.8
8	64KB	277300	4678	77420	1336
8	256KB	1139000	25980	362800	8817
8	1MB	4568000	68190	2285000	64340
8	4MB	18180000	231200	9099000	238800
8	16MB	71930000	234700	35740000	959700
64	64	201.5	6.344	230.5	7.442
64	256	362.9	4.582	250	11.07
64	1KB	1421	32.08	375.8	6.274
64	4KB	5039	53.51	1053	172.2
64	16KB	27550	2302	7058	2462
64	64KB	134500	21980	35830	6002
64	256KB	527100	1843	137700	2227
64	1MB	2113000	4969	865200	15760
64	4MB	8267000	54730	3396000	59160

(b)

Size (bytes)	Plain Mean	Plain S.D.	Mobile Mean	Mobile S.D.
64	1161	114.7	1446	84.64
256	2371	597.3	2056	90.95
1KB	7492	634.6	5182	103.5
4KB	34570	230.4	22860	155.7
16KB	153700	1325	95890	1400
64KB	860000	9057	593000	9443
256KB	3552000	38880	2434000	33080
1MB	14450000	162300	11800000	221300
4MB	57410000	566300	46580000	817900

(c)